8086

MICROPROCESSOR

Scanned with CamScanner

IMPORTANT FEATURES OF 8086:

1) **Buses:**

Address Bus: 8086 has a **20-bit address bus**, hence it can access 2^{20} Byte memory i.e. **1MB**. The **address range** for this memory is **00000H ... FFFFFH.**

Data Bus: 8086 has a **16-bit data bus** i.e. it can access 16 bit data in one operation. Its ALU and internal data registers are also 16-bit.
Hence 8086 is called as a **16-bit μP.**

Control Bus: The control bus carries the signals responsible for performing various operations such as $\overline{\textbf{RD}}$, $\overline{\textbf{WR}}$ etc.

2) 8086 supports **Pipelining.**
It is the process of **"Fetching the next instruction, while executing the current instruction"**.
Pipelining improves performance of the system.

3) 8086 has **2 Operating Modes**.
   i. **Minimum Mode** ... here 8086 is the only processor in the system (uni-processor).
   ii. **Maximum Mode** ... 8086 with other processors like 8087-NDP/8089-IOP etc.
   Maximum mode is intended for multiprocessor configuration.

4) 8086 provides **Memory Banks**.
The entire memory of 1 MB is **divided into 2 banks of 512KB each,** in order to transfer 16-bits in 1 cycle. The banks are called **Lower Bank** (even) **and Higher Bank** (odd).

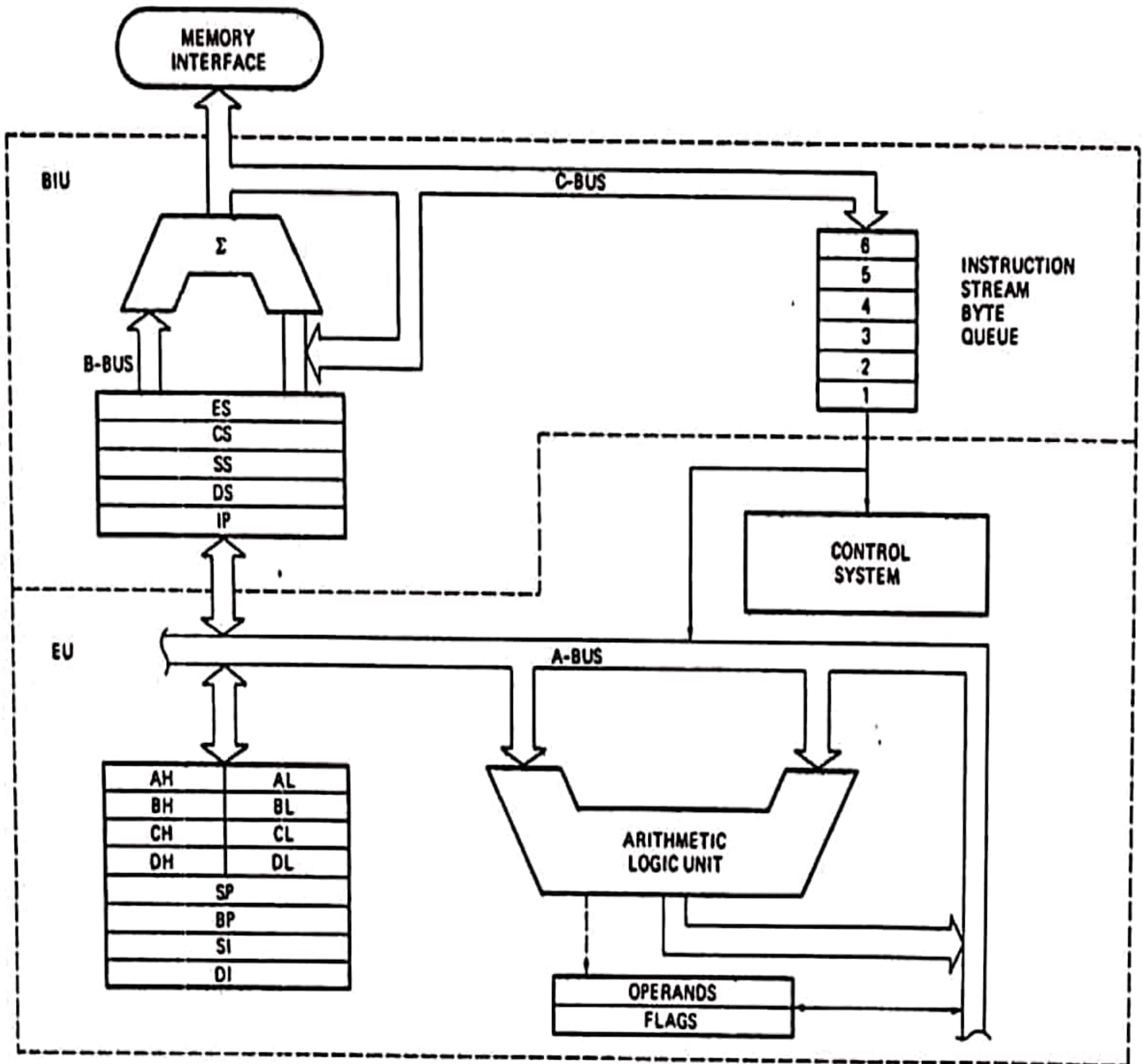
5) 8086 supports **Memory Segmentation.**
Segmentation means dividing the memory into logical components. Here the memory is divided into **4 segments: Code, Stack, Data and Extra Segment.**

6) 8086 has **256 interrupts**.
The ISR addresses for these interrupts are stored in the IVT (Interrupt Vector Table).

7) 8086 has a **16-bit IO address** ∴ it can access **2^{16} IO ports** (2^{16} = 65536 i.e. 64K IO Ports).

Scanned with CamScanner

ARCHITECTURE OF 8086



Scanned with CamScanner

As 8086 does 2-stage pipelining, its architecture is divided into two units:

1. Bus Interface Unit (BIU)
2. Execution Unit (EU)

BUS INTERFACE UNIT (BIU)

1. It provides the **interface** of 8086 **to** other devices.
2. It **operates w.r.t. Bus cycles .**
   This means it performs various machine cycles such as Mem Read, IO Write etc to transfer data with Memory and I/O devices.
3. It performs the following functions:
   a) It **generates** the 20-bit **physical address** for memory access.
   b) **Fetches Instruction** from memory.
   c) **Transfers data** to and from the **memory and IO**.
   d) **Supports Pipelining** using the 6-byte instruction queue.

The main components of the BIU are as follows:

a) SEGMENT REGISTERS:

1) CS Register

CS holds the **base** (Segment) **address** for the **Code Segment**.
All programs are stored in the Code Segment.
It is **multiplied by 10H** (16_d), to give the **20-bit physical address** of the **Code Segment**.
Eg: If **CS = 4321H** then CS × 10H = **43210H → Starting address** of Code Segment.
CS register cannot be modified by executing any instruction except branch instructions

2) DS Register

DS holds the **base** (Segment) **address** for the **Data Segment**.
It is **multiplied by 10H** (16_d), to give the **20-bit physical address** of the **Data Segment**.
Eg: If **DS = 4321H** then DS × 10H = **43210H → Starting address** of Data Segment.

3) SS Register

SS holds the **base** (Segment) **address** for the **Stack Segment**.
It is **multiplied by 10H** (16_d), to give the **20-bit physical address** of the **Stack Segment**.
Eg: If **SS = 4321H** then SS × 10H = **43210H → Starting address** of Stack Segment.

4) ES Register

ES holds the **base** (Segment) **address** for the **Extra Segment**.
It is **multiplied by 10H** (16_d), to give the **20-bit physical address** of the **Extra Segment**.
Eg: If **ES = 4321H** then ES × 10H = **43210H → Starting address** of Extra Segment.

b) Instruction Pointer (IP register)

It is a **16-bit register**.
It **holds offset of** the **next instruction in** the **Code Segment**.

Scanned with CamScanner

Address of the **next instruction** is calculated as **CS x 10H + IP**.
IP is **incremented after every instruction byte is fetched.**
IP gets a new value whenever a branch occurs.

c) Address Generation Circuit

The BIU has a **Physical Address Generation Circuit.** It generates the 20-bit physical address using Segment and Offest addresses using the formula:

Physical address = Segment Address x 10h + Offset Address

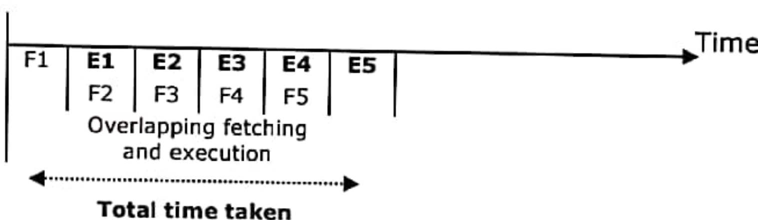
Viva Question: Explain the real procedure to obtain the Physical Address?
The Segment address is left shifted by 4 positions, this multiplies the number by 16 (i.e. 10h) and then the offset address is added.
Eg: If Segment address is 1234h and Offset address is 0005h, then the physical address (12345h) is calculated as follows:
1234h = $(0001\ 0010\ 0011\ 0100)_{binary}$
Left shift by four positions and we get $(0001\ 0010\ 0011\ 0100\ \mathbf{0000})_{binary}$ i.e. 12340h
Now add $(0000\ 0000\ 0000\ 0101)_{binary}$ i.e. 0005h and we get $(0001\ 0010\ 0011\ 0100\ 0101)_{binary}$ i.e. 12345h.

d) 6-Byte Pre-Fetch Queue {Pipelining – 4m}

It is a **6-byte FIFO RAM** used to implement **Pipelining**.
Fetching the next instruction while executing the current instruction is called ***Pipelining***.
BIU fetches the next **"six instruction-bytes"** from the Code Segment and stores it into the queue.
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 the μP.
Pipelining **fails when** a **branch** occurs, as the pre-fetched instructions are no longer useful.
Hence as soon as 8086 detects a branch operation, it clears/discards the entire queue. Now, the next six bytes from the new location (branch address) are fetched and stored in the queue and Pipelining continues.

NON-PIPELINED PROCESSOR EG: 8085

PIPELINED PROCESSOR EG: 8086



Scanned with CamScanner

Execution Unit (EU)

1. It **fetches** instructions **from** the **Queue in BIU**, **decodes** and **executes them**.
2. It performs **arithmetic**, **logic** and **internal data transfer** operations.
3. It sends request signals to the BIU to access the external module.
4. It **operates w.r.t. T-States** (clock cycles). ☺ **For doubts contact Bharat Sir on 98204 08217**

The main components of the EU are as follows:

a) General Purpose Registers

8086 has four 16-bit general-purpose registers **AX**, **BX**, **CX** and **DX**. These are **available** to the programmer, for storing values during programs. Each of these can be **divided** into two **8-bit registers** such as AH, AL; BH, BL; etc. Beside their general use, these registers also have some **specific functions.**

***AX** Register (16-Bits)*

It holds operands and results during **multiplication** and **division** operations.
All IO data transfers using IN and OUT instructions use A reg (AL/AH or AX).
It functions as accumulator during **string operations**.

***BX** Register (16-Bits)*

Holds the **memory address** (offset address), in **Indirect Addressing modes**.

***CX** Register (16-Bits)*

Holds **count** for instructions like: **Loop**, **Rotate**, **Shift** and **String** Operations.

***DX** Register (16-Bits)*

It is used with AX to hold **32 bit** values during **Multiplication** and **Division**.
It is used to **hold** the **address** of the **IO Port** in **indirect IO addressing** mode.

b) Special Purpose Registers

*Stack Pointer (**SP** 16-Bits)*

It is holds **offset address of** the **top of the Stack. Stack is a set of memory locations operating in LIFO manner. Stack is present in the memory in Stack Segment.**
SP is used with the SS Reg to calculate physical address for the Stack Segment. It used during instructions like PUSH, POP, CALL, RET etc. During PUSH instruction, SP is decremented by 2 and during POP it is incremented by 2.

*Base Pointer (**BP** 16-Bits)*

BP can hold **offset address of** any location in the **stack segment.**
It is used to access random locations of the stack. #Please refer Bharat Sir's Lecture Notes for this ...

*Source Index (**SI** 16-Bits)*

It is normally used to hold the **offset address** for **Data segment** but can also be used for other segments using Segment Overriding. It holds **offset address** of **source data** in Data Seg, during **String Operations.**

Scanned with CamScanner

*Destination Index (**DI** 16-Bits)*

It is normally used to hold the **offset address** for **Extra segment** but can also be used for other segments using Segment Overriding. It holds **offset address** of **destination** in Extra Seg, during **String Operations.**

c) ALU (16-Bits)

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

d) Operand Register

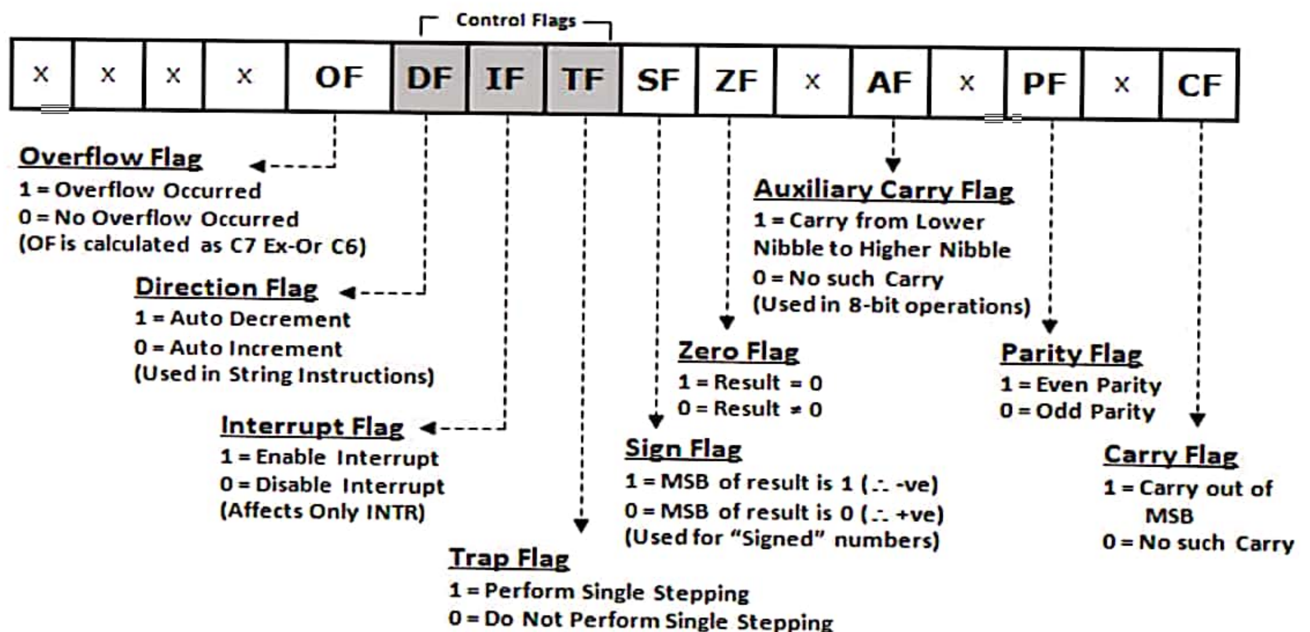
It is a 16-bit register used by the control register to hold the operands temporarily.
It is **not available** to the Programmer.

e) Instruction Register and Instruction Decoder (Present inside the Control Unit)

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

f) Flag Register (16-Bits)

It has **9 Flags**.
These flags are of two types: **6-Status** (Condition) Flags and **3-Control** Flags.
Status flags are affected by the ALU, after every arithmetic or logic operation. They give the **status of the current result**.
The **Control flags** are used to control certain operations.
They are changed by the programmer.



Scanned with CamScanner

Status Flags

1) *Carry flag (**CY**)*
   It is **set** whenever there is a **carry** {or borrow} out of the MSB of a the result (D7 bit for an 8-bit operation D15 bit for a 16-bit operation)

2) *Parity Flag (**PF**)*
   It is **set** if the result has **even parity**.

3) *Auxiliary Carry Flag (**AC**)*
   It is **set** if a carry is generated out of the **Lower Nibble**.
   It is used only in 8-bit operations like DAA and DAS.

4) *Zero Flag (**ZF**)*
   It is **set** if the result is **zero**.

5) *Sign Flag (**SF**)*
   It is **set** if the **MSB** of the result is **1**.
   For **signed** operations, such a number is treated as **–ve**.

6) *Overflow Flag (**OF**)*
   It will be set if the **result of** a **signed operation** is **too large to fit** in the number of bits available to represent it. It can be **checked using** the **instruction INTO** (Interrupt on Overflow). #Please refer Bharat Sir's Lecture Notes for this ...

Control Flags

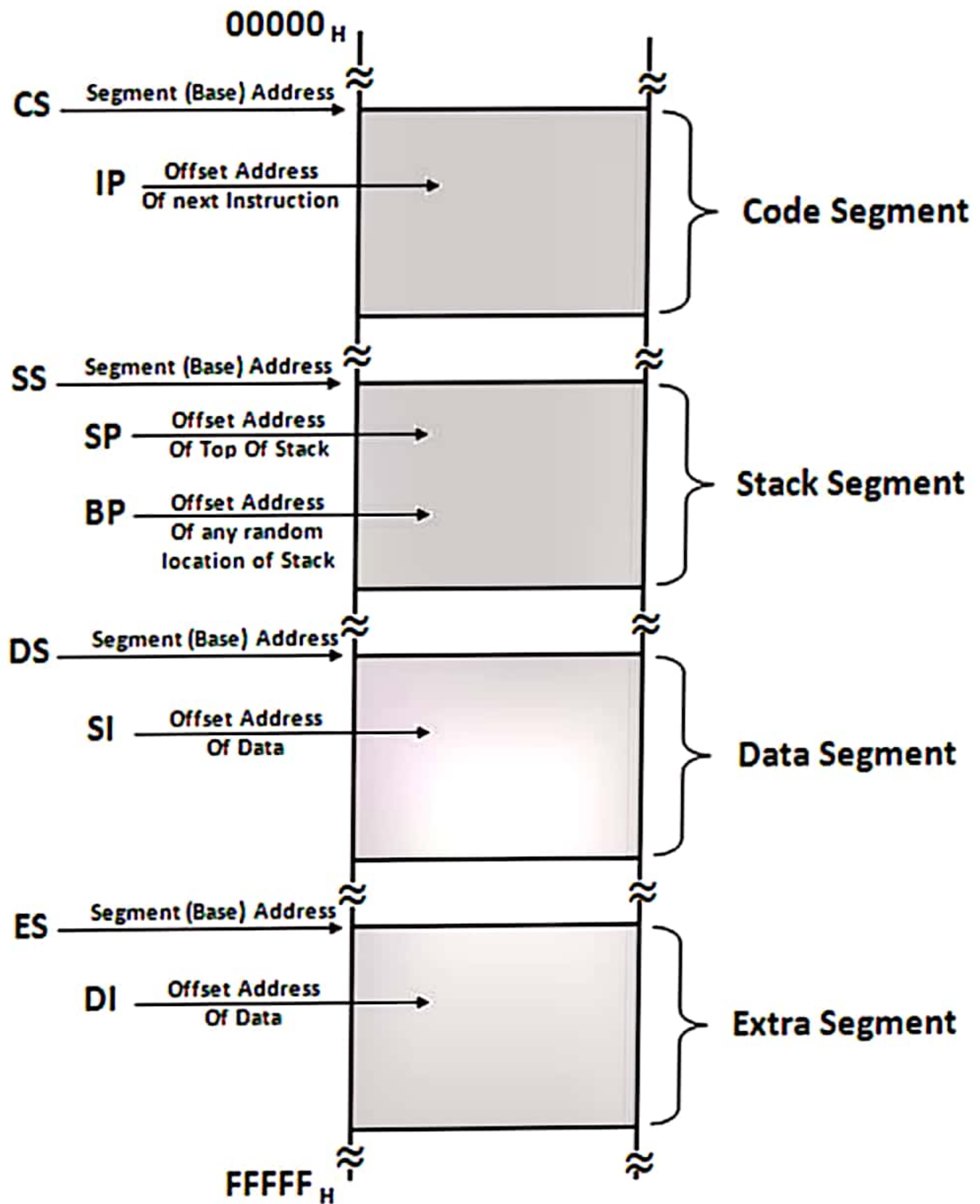
1) *Trap Flag (**TF**)*
   It is used to **set** the Trace Mode i.e. start **Single Stepping Mode**.
   Here the μP is **interrupted after every instruction** so that, the **program** can be **debugged**.

2) *Interrupt Enable Flag (**IF**)*
   It is used to mask (disable) or unmask (enable) the INTR interrupt.

3) *Direction Flag (**DF**)*
   If this flag is **set, SI** and **DI** are in **auto-decrementing** mode in **String Operations**.

Scanned with CamScanner

MEMORY SEGMENTATION IN 8086



Scanned with CamScanner

NEED FOR SEGMENTATION / CONCEPT OF SEGMENTATION

1) Segmentation means **dividing** the memory into **logically different parts called segments**.
2) 8086 has a **20-bit address bus**, hence it can access 2^{20} Bytes i.e. **1MB** memory.
3) But this also means that **Physical address** will now be **20 bit**.
4) It is **not possible** to work with a **20 bit address** as it is **not** a **byte compatible** number. (20 bits is two and a half bytes).
5) To avoid working with this incompatible number, we **create a virtual model** of the memory.
6) Here the memory is **divided into 4 segments**: Code, Stack Data and Extra.
7) The **max size** of a segment is **64KB** and the **minimum size** is **16 bytes**.
8) Now programmer can access each location with a **VIRTUAL ADDRESS**.
9) The Virtual Address is a **combination** of **Segment Address and Offset Address**.
10) **Segment Address indicates where the segment is located in the memory (base address)**
11) **Offset Address gives the offset of the target location within the segment.**
12) Since both, Segment Address and Offset Address are **16 bits each**, they both are **compatible numbers** and can be easily used by the programmer.
13) Moreover, **Segment Address is given only in the beginning** of the program, to initialize the segment. Thereafter, we **only give offset address**.
14) **Hence we can access 1 MB memory using only a 16 bit offset address for most part of the program. This is the advantage of segmentation.**
15) Moreover, dividing Code, stack and Data into different segments, makes the memory **more organized and prevents accidental overwrites** between them.
16) The **Maximum Size** of a segment is **64KB because offset addresses are of 16 bits**. 2^{16} = 64KB.
17) As max size of a segment is 64KB, programmer can create **multiple Code/Stack/Data segments** till the entire 1 MB is utilized, but **only one of each** type will be **currently active**.
18) The physical address is calculated by the microprocessor, using the formula:

    PHYSICAL ADDRESS = SEGMENT ADDRESS X 10H + OFFSET ADDRESS

19) Ex: if Segment Address = 1234H and Offset Address is 0005H then
    Physical Address = 1234H x 10H + 0005H = 12345H
20) This formula automatically ensures that the **minimum size of a segment is 10H bytes** (10H = 16 Bytes).

Scanned with CamScanner

Code Segment

This segment is used to hold the **program** to be executed.
Instruction 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.

Data Segment

This segment is used to hold **general data**.
This segment also holds the **source** operands during **string** operations.
DS register holds the 16-bit **base** address for this segment.
BX register is used to hold the 16-bit **offset** for this segment.
SI register (Source Index) holds the 16-bit **offset** address during String Operations.

Stack Segment

This segment holds the **Stack** memory, which operates in LIFO manner.
SS holds its **Base** address.
SP (Stack Pointer) holds the 16-bit **offset** address of the **Top** of the Stack.
BP (Base Pointer) holds the 16-bit **offset** address during **Random Access**.

Extra Segment

This segment is used to hold **general data**
Additionally, this segment is used as the **destination** during **String Operations**.
ES holds the **Base** Address.
DI holds the **offset** address during string operations.

Advantages of Segmentation:

1) It permits the programmer to access 1MB **using only 16-bit address**.
2) Its **divides** the **memory logically** to store Instructions, Data and Stack separately.

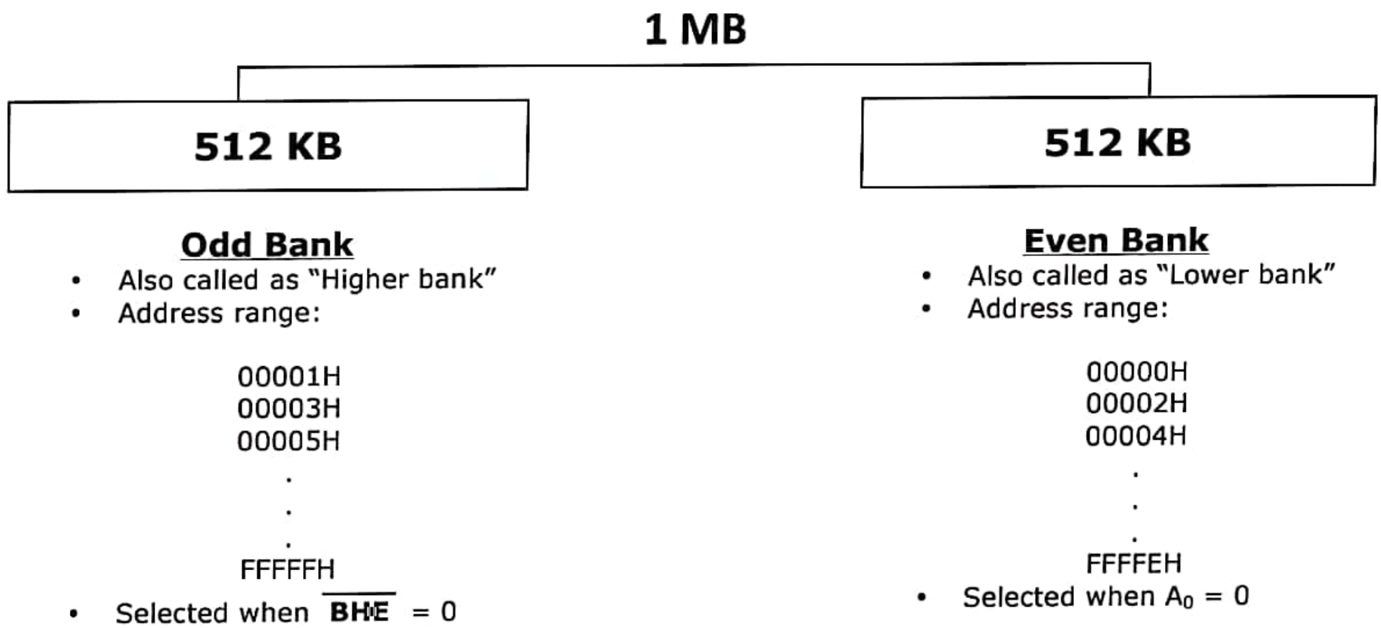
Disadvantage of Segmentation:

1) Although the total memory is 16*64 KB, **at a time only 4*64 KB memory can be accessed**.

Scanned with CamScanner

MEMORY BANKING IN 8086

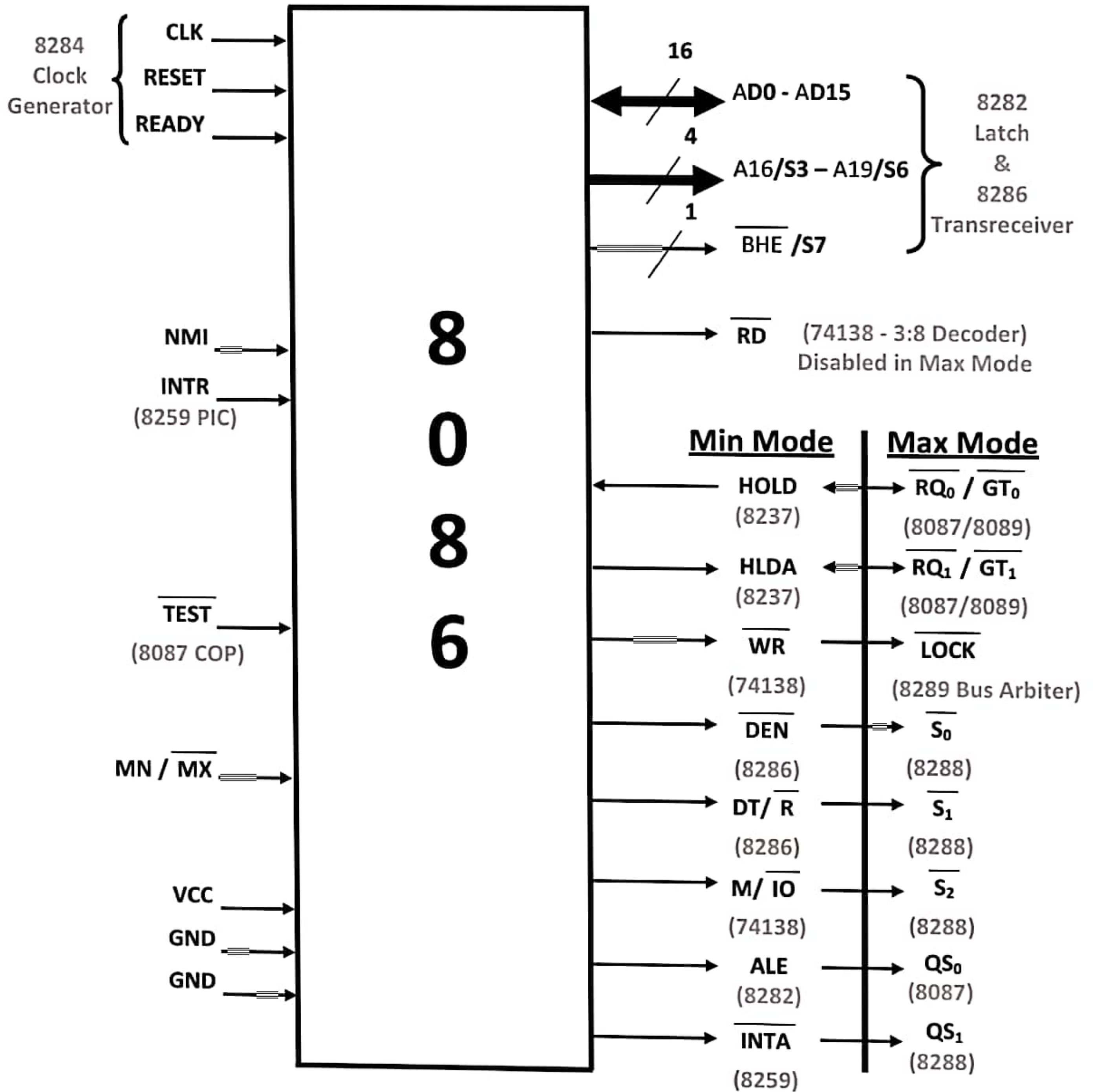
- As 8086 has a 16-bit data bus, it should be able to access 16-bit data **in one cycle**.
- To do so it needs to read from **2 memory locations**, as one memory location carries only one byte. 16-bit data is stored in two consecutive memory locations.
- However, if both these memory locations are in the same memory chip then they cannot be accessed at the same time, as the address bus of the chip cannot contain two address simultaneously.
- Hence, the memory of 8086 is divided into two banks each bank provides 8-bits.
- The division is done in 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 the **"Even bank"**, while the other is called **"Odd bank"** containing all odd addresses. ☺ For doubts contact Bharat Sir on 98204 08217
- Generally for any 16-bit operation, the Even bank provides the lower byte and the ODD bank provides the higher byte. Hence the **Even bank** is also called the **Lower bank** and the **Odd bank** is also called the **Higher bank**.



$\overline{BHE}$	A_0	OPERATION
0	0	R/W **16-bit** from both banks
0	1	R/W 8-bit from higher bank
1	0	R/W 8-bit from lower bank
1	1	No Operation (Idle).

Scanned with CamScanner

PIN DIAGRAM OF 8086



Scanned with CamScanner

Machine Cycles

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	Bus Cycle / Machine Cycle
0	0	0	INTA Cycle
0	0	1	I/O Read
0	1	0	I/O Write
0	1	1	Halt
1	0	0	Opcode Fetch
1	0	1	Memory Read
1	1	0	Memory Write
1	1	1	Inactive

Segment Selection

S_4	S_3	Segment. Selected
0	0	Extra Segment
0	1	Stack Segment
1	0	CS/No Segment Selected
1	1	Data Segment

Queue Synchronization

QS_1	QS_0	Queue Operation
0	0	NOP
0	1	Opcode Fetch from queue
1	0	Queue is Cleared
1	1	Fetch remaining instruction bytes form queue

Scanned with CamScanner

PIN DESCRIPTIONS

- **CLK**

  This is the clock-input line.
  An external clock generator (8284) provides the clock signal.
  8086 required single phase, 33% duty cycle, TTL clock signal.

- **RESET**

  This is the reset input signal. The 8284 Clock generator provides it.
  It **Clears** the **Flag** register and the Instruction **Queue**.
  It also **Clears** the **DS, SS, ES and IP** registers and ***Sets*** the bits of ***CS*** register.
  Hence the **reset vector address** of 8086 is **FFFFOH**
  (as CS = FFFFH and IP = 0000H).

- **READY**

  This signal is used to synchronize the µP with **slower peripherals**.
  Devices inform the µP whether they are ready or not.
  µP **samples** the READY input **during T3 state** of a Machine Cycle
  If device is Ready it send a "1" on the Ready pin else send a "0".
  If Ready pin is 0, µP inserts **wait-states** between T3 and T4 and will only come out of Wait state when Ready becomes 1 thereby ensuring that the Device is ready.

- **$\overline{\text{TEST}}$**

  It is an active low **input** line dedicated for 8087 Co-processor.
  In **Maximum Mode** whenever the **Co-Processor** is **busy** it makes this pin HIGH.
  µP **samples** the **$\overline{\text{TEST}}$** input only when it encounters the WAIT instruction.
  If the **$\overline{\text{TEST}}$** pin is **high**, the **µP enters wait state**, till **$\overline{\text{TEST}}$** pin becomes low i.e. 8087 is free.
  In minimum mode it is not used and is connected to ground (VIVA Q).

- **MN/ $\overline{\text{MX}}$**

  This is an **input** signal to 8086.
  If this signal is **HIGH,** 8086 is in **Minimum** mode i.e. Uni-Processor system.
  If this signal is **Low,** 8086 is in **Maximum** mode i.e. Multiprocessor system.

- **NMI**

  This is a **non-maskable, edge** triggered, **high priority** interrupt.
  On receiving an interrupt on NMI line, the µP executes **INT 2** i.e. and takes control to location 2 x 4 = 00008H in the Interrupt Vector Table (IVT), to get the value for CS ad IP.

Scanned with CamScanner

- **INTR**

  This is a **maskable, level** triggered, **low priority** interrupt.

  On receiving an interrupt on INTR line, the μP executes **2** $\overline{\textbf{INTA}}$ cycles.

  On FIRST $\overline{\textbf{INTA}}$ pulse, the interrupting device (8259) prepares **to send a vector number "N"**.

  On SECOND $\overline{\textbf{INTA}}$ pulse, the interrupting device (8259 PIC) **sends vector number "N"** to μP.

  Now μP will multiply N x 4 and go to the IVT to obtain the ISR address i.e. values for IP and CS.

- $\overline{\textbf{RD}}$

  It is an active low output signal. When it is low 8086 **reads** from memory or IO.

- **VCC and GND**

  Used for power supply. Two grounds are due to the two internal layers in μP.

- **AD15 - AD0**

  It carries $\mathbf{A_{15}}$ **-** $\mathbf{A_0}$ **(address)** during T1 of a Machine Cycle when **ALE = 1**.
  It carries $\mathbf{D_{15}}$ **-** $\mathbf{D_0}$ **(data)** for remaining T-States of a Machine Cycle when **ALE = 0**.

- **A16/S3 - A19/S6**

  These lines carry ($\mathbf{A_{16}}$ **...** $\mathbf{A_{19}}$) during T1 of every M/C Cycle.
  T2 onwards these lines carry the Status signals $\mathbf{S_3}$ **...** $\mathbf{S_6}$.
  $\mathbf{S_3}$ and $\mathbf{S_4}$ indicate the memory segment currently accessed. $\mathbf{S_5}$ gives the status of the **Interrupt Enable Flag**. $\mathbf{S_6}$ goes **low** when **8086 controls the system bus**.

- $\overline{\textbf{BHE}}$ **/S7**

  It carries $\overline{\textbf{BHE}}$ during T1. $\overline{\textbf{BHE}}$ is used to enable the higher bank.

  T2 onwards it carries S7, which is reserved for "*further development*" ☺.

Scanned with CamScanner

MIN Mode / *Max Mode* Signals (10m question --- Important)

- **HOLD --- $\overline{RQ_0} / \overline{GT_0}$**

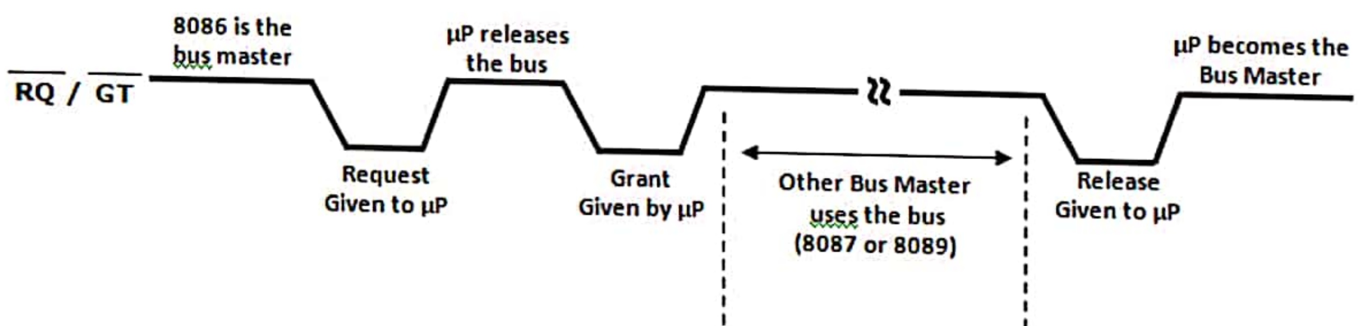
In **Minimum** Mode this line carries the **HOLD** input signal.
The **DMA Controller issues** the HOLD signal to request for the system bus.
In response 8086 completes the current bus cycle and releases the system bus.

*In **Maximum** Mode it carries the bi-directional $\overline{RQ_0} / \overline{GT_0}$ signal (Request/Grant).The **external bus master** (eg: 8087) **sends an active low pulse** to request for the sys bus.*
*In **response** the **8086** completes the current bus cycle, releases the system bus and **sends an active low Grant pulse** on the same line to the external bus controller.*
***8086 gets back** the system bus only after external **bus master sends an active low release pulse** on the same line.*

- **HLDA --- $\overline{RQ_1} / \overline{GT_1}$**

In **Minimum** Mode this line carries the **HLDA** signal.
This signal is issued by 8086 after releasing the system bus.

*In **Maximum** Mode it functions as $\overline{RQ_1} / \overline{GT_1}$, which is the same as $\overline{RQ_0} / \overline{GT_0}$ but is of **lower priority**.*



Scanned with CamScanner

- $\overline{\textbf{WR}}$ --- $\overline{\textbf{LOCK}}$

In **Minimum** Mode this line carries the $\overline{\textbf{WR}}$ signal.

It is used with **M/** $\overline{\textbf{IO}}$ to write to Memory or IO Device.

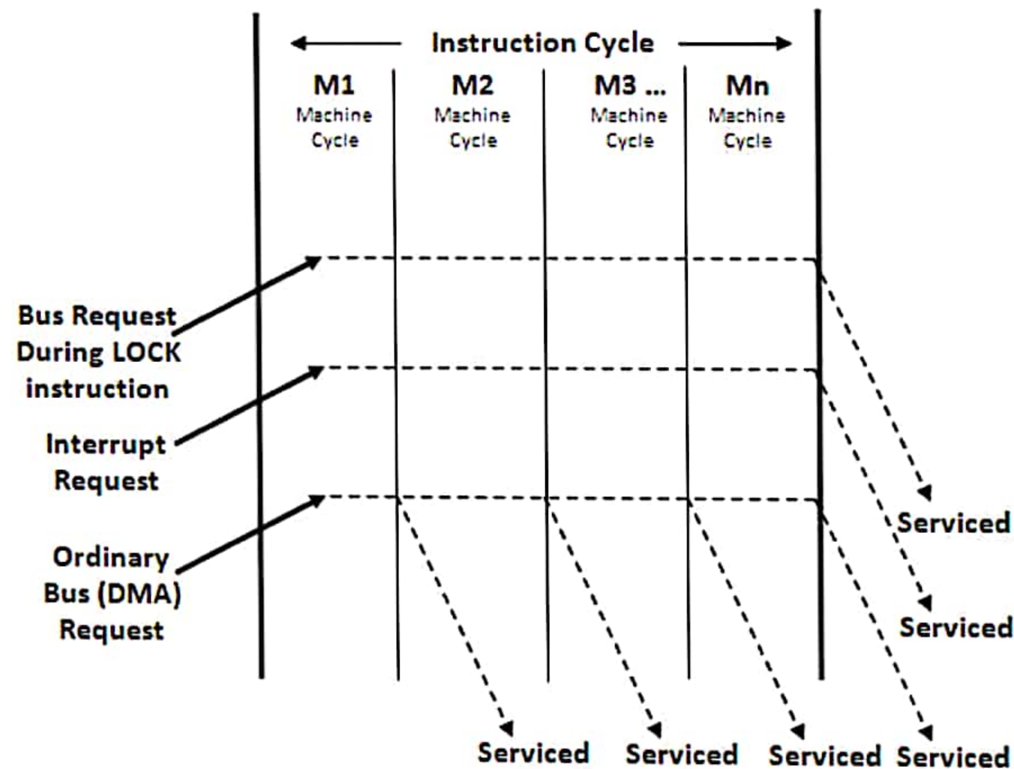
*In **Maximum Mode** it functions as the* $\overline{\textbf{LOCK}}$ *output line.*

***When** this signal is **active** (i.e. low) the **external bus master cannot take control of the system bus**. It is activated when 8086 executes an **instruction** with the* $\overline{\textbf{LOCK}}$ *prefix, and remains active till next instruction.*

***LOCK Prefix**: Normally a bus request is serviced after the current machine cycle and an interrupt request is serviced after the current instruction cycle.*

But if we write LOCK prefix before any instruction, then even if there is a bus request, the bus will be released only after the current instruction. Hence the bus is said to be locked during the instruction.

µP will maintain $\overline{\textbf{LOCK}}$ *signal low throughout the instruction to indicate that it is performing an instruction with LOCK prefix.* $\overline{\textbf{LOCK}}$ *signal is given to 8289 Bus Arbiter in Loosely Coupled Systems, to prevent 8289 from releasing the system bus to other bus masters.*



Scanned with CamScanner

- **$\overline{\text{DEN}}$ --- $\overline{\text{S0}}$**

  In **Minimum** Mode it carries the **$\overline{\text{DEN}}$** signal
  It is used to **enable** the data **transceivers** (bi-directional buffers - **IC 8286**).
  *In **Maximum** Mode it carries the **$\overline{S_0}$** signal.*
  In Maximum Mode, Bus Controller (IC 8288) gives the DEN signal.

- **DT/ $\overline{\text{R}}$ --- $\overline{\text{S1}}$**

  In **Minimum** Mode it carries the **DT/ $\overline{\text{R}}$** signal
  This signal goes **low** for a **Read** operation and high for a write operation.
  *In **Maximum** Mode it carries the **$\overline{S_1}$** signal.*

  In Maximum Mode, Bus Controller gives the DT/ $\overline{\text{R}}$ signal.

$\overline{\text{DEN}}$	DT/ $\overline{\text{R}}$	Action
1	X	Transreceiver is disabled
0	0	Receive data
0	1	Transmit data

- **M/ $\overline{\text{IO}}$ --- $\overline{\text{S2}}$**

  In **Minimum** Mode it carries the **M/ $\overline{\text{IO}}$** signal, to distinguish between Memory and IO access.
  *In **Maximum** Mode it carries the **$\overline{S_2}$** signal.*
  In Maximum Mode $\overline{S_2}$, $\overline{S_1}$ and $\overline{S_0}$ are used to generate the appropriate control signal.

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

- **ALE --- QS0**

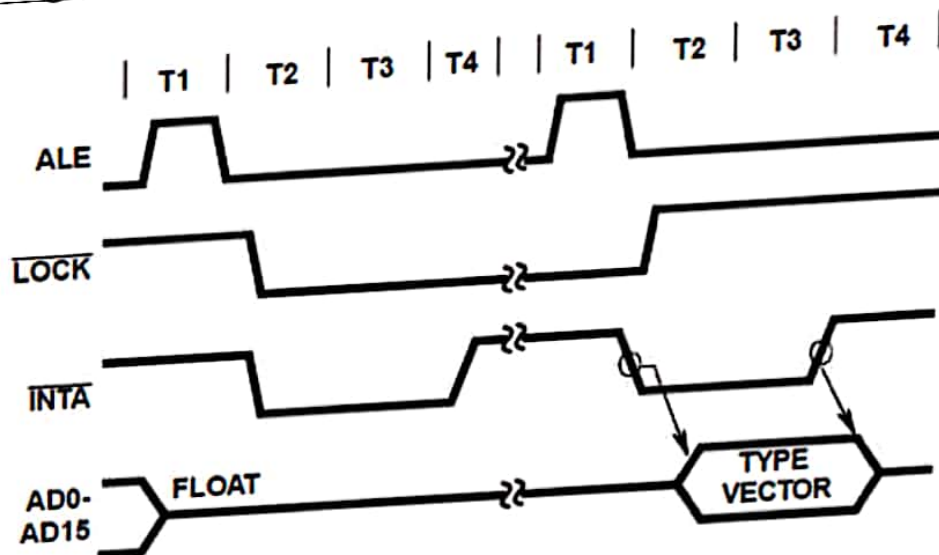
  In **Minimum** Mode it carries the **ALE** signal, which is used to latch the address.
  *In **Maximum** Mode it carries the **QS_0** signal.*
  *It is used with **QS_1** to indicate the Instruction Queue Status.*
  In Maximum Mode, Bus Controller gives the ALE signal.

Scanned with CamScanner

- **$\overline{\text{INTA}}$ --- QS1**

In **Minimum** Mode it carries the $\overline{\textbf{INTA}}$ signal
It is issued in response to an interrupt on the INTR line.
It is used to read the vector number form the interrupting device.
In ***Maximum*** *Mode it carries the* ***QS_1*** *signal.* ☺ **For doubts contact Bharat Sir on 98204 08217**
In Maximum Mode, Bus Controller gives the $\overline{\textbf{INTA}}$ ***signal.***

Timing diagram for 2 back-to-back $\overline{\text{INTA}}$ cycles



- As shown above there are two $\overline{\textbf{INTA}}$ cycles.
- Each $\overline{\textbf{INTA}}$ cycle is of 4 T-states
- In the 1st $\overline{\textbf{INTA}}$ cycle, the interrupting device (8259) **starts preparing** the vector number **"N"**.
- In the 2nd $\overline{\textbf{INTA}}$ cycle, 8259 **sends the vector number (Type Number) "N"**, to the microprocessor, through the multiplexed address data bus.
- The microprocessor then **multiplies the number by 4** and **goes to** the corresponding location in the **IVT** (Interrupt Vector Table).
- From there it **obtains the values of Segment Address and Offset Address** for the ISR of the corresponding interrupt, and hence **executes the ISR**.
- $\overline{\textbf{LOCK}}$ signal is **held low between** the two $\overline{\textbf{INTA}}$ cycles, **so that the bus is not released** in between this process.

Scanned with CamScanner

ADDRESSING MODES OF 8086

8086 provides different addressing modes for Data, Program and Stack Memory.

ADDRESSING MODES FOR DATA MEMORY {IMP}

I IMMEDIATE ADDRESSING MODE

In this mode the **operand** is specified in the **instruction** itself.
Instructions are **longer** but the **operands** are **easily identified**.

Eg:	**MOV CL, 12H**	*; Moves 12 immediately into CL register*
	MOV BX, 1234H	*; Moves 1234 immediately into BX register*

II REGISTER ADDRESSING MODE

In this mode **operands** are specified using **registers**.
Instructions are **shorter** but **operands cant** be **identified** by looking at the instruction.

Eg:	**MOV CL, DL**	*; Moves data of DL register into CL register*
	MOV AX, BX	*; Moves data of BX register into AX register*

III DIRECT ADDRESSING MODE

In this mode **address** of the operand is directly specified **in the instruction**.
Here **only** the **offset address is specified**, the segment being indicated by the instruction.

Eg: **MOV CL, [4321H]**

; Moves data from location 4321H in the data
; segment into CL
; The physical address is calculated as
; $\mathbf{DS * 10_H + 4321}$
; Assume DS = 5000H
; ∴ P A= 50000 + 4321 = 54321H
; ∴ CL ← [54321H]

Eg: **MOV CX, [4320H]**

; Moves data from location 4320H and 4321H
; in the data segment into CL and CH resp.

Scanned with CamScanner

IV INDIRECT ADDRESSING MODES

REGISTER INDIRECT ADDRESSING MODE

In this mode the µP uses any of the 2 **base registers** BP, BX or any of the two index registers SI, DI to provide the offset **address** for the data byte.

The segment is indicated by the Base Registers:
BX -- Data Segment, BP --- Stack Segment

Eg: **MOV CL, [BX]** *; Moves a byte from the address pointed by BX in Data*
; Segment into CL.
*; Physical Address calculated as DS * 10_H + BX*

Eg: MOV [BP], CL *; Moves a byte from CL into the location pointed by BP in*
; Stack Segment.
*; Physical Address calculated as SS * 10_H + BP*

REGISTER RELATIVE ADDRESSING MODE

In this mode the operand address is calculated using one of the **base registers** and a **8-bit** or a **16-bit displacement.**

Eg:MOV CL, [BX+4] *; Moves a byte from the address pointed by BX+4 in*
; Data Seg to CL.
*; Physical Address: DS * 10_H + BX + 4H*

Eg: MOV 12H [BP], CL *; Moves a byte from CL to location pointed by BP+12H in*
; the Stack Seg.
*; Physical Address: SS * 10_H + BP + 12H*

BASE INDEXED ADDRESSING MODE

Here, operand address is calculated as **Base register plus** an **Index** register.

Eg: **MOV CL, [BX+SI]** *; Moves a byte from the address pointed by BX+SI*
; in Data Segment to CL.
*; Physical Address: DS * 10_H + BX + SI*

Eg: MOV [BP+DI], CL *; Moves a byte from CL into the address pointed by*
; BP+DI in Stack Segment.
*; Physical Address: SS * 10_H + BP + DI*

Scanned with CamScanner

Base Relative Plus Index Addressing Mode

In this mode the address of the operand is calculated as **Base** register **plus Index** register **plus 8-bit** or **16-bit displacement**.

Eg: **MOV CL, [BX+DI+20]** *; Moves a byte from the address pointed by*
; BX+SI+20H in Data Segment to CL.
*; Physical Address: DS * 10_H + BX + SI+ 20H*

Eg: MOV [BP+SI+2000], CL *; Moves a byte from CL into the location pointed by*
; BP+SI+2000H in Stack Segment.
*; Physical Address: SS * 10_H + BP+SI+2000H*

V Implied Addressing Mode

In this addressing mode the operands are implied and are hence not specified in the instruction.
#Please refer Bharat Sir's Lecture Notes for this ...

Eg: **STC** *; Sets the Carry Flag.*

Eg: CLD *; Clears the Direction Flag.*

Important points for understanding addressing modes...

1) *Anything given in square brackets will be an Offset Address also called Effective Address.*
2) *MOV instruction by default operates on the Data Segment; unless specified otherwise.*
3) *BX and BP are called Base Registers.*
   BX holds Offset Address for Data Segment.
   BP holds Offset Address for Stack Segment.
4) *SI and DI are called Index Registers*
5) *The Segment to be operated is decided by the Base Register and NOT by the Index Register.*

Scanned with CamScanner

ADDRESSING MODES FOR PROGRAM MEMORY

(optional --- to be written only if asked)

This addressing mode is required for instructions that **cause a branch** in the program. If the Branch is **within the same segment**, it is called as an **Intra-Segment Branch** or a **Near** Branch. If the Branch is **in a different segment**, it is called as an **Inter-Segment Branch** or a **Far** Branch.

INTRA SEGMENT DIRECT ADDRESSING MODE

Address is **specified directly** in the instruction as an **8-bit (or 16-bit) displacement.**
The **effective address** is thus calculated by **adding** the **displacement to** current value of **IP**.
As it is intra-segment, ONLY IP changes, CS does not change.
If the **displacement** is **8-bit** it is called as a **Short Branch**.
This addressing mode is also called as **relative addressing mode**.

Eg: **Code SEGMENT**

```
Code      SEGMENT
Prev:     .........
Current:  JMP Prev        ; IP ← Offset address of "Prev"
Code      ENDS
```

Or

```
Code      SEGMENT
Current:  JMP Next
Next:     ............    ; IP ← Offset address of "Next"
Code      ENDS
```

INTER SEGMENT DIRECT ADDRESSING MODE

The new Branch location is **specified directly** in the instruction
Both **CS and IP get new values**, as this is an inter-segment branch.
Eg:

```
Code_1    SEGMENT
Current:  JMP NextSeg     ; CS ← Segment address of "NextSeg"
                          ; IP ← Offset address of "NextSeg"
Code_1    ENDS
Code_2    SEGMENT
NextSeg:  ............
Code_2    ENDS
```

Scanned with CamScanner

Intra Segment Indirect Addressing Mode

Address is **specified indirectly** through a **register** or a **memory location** (in DS only).
The value in the IP is **replaced** with the new value.
As it is intra-segment, ONLY IP changes, CS does not change.

Eg: **JMP WORD PTR [BX]** *; IP ← {DS:[BX], DS:[BX+1]}*

Inter Segment Indirect Addressing Mode

The new Branch location is **specified indirectly** through a **register** or a **memory location** (in DS only). #Please refer Bharat Sir's Lecture Notes for this ...
Both CS and IP get new values, as this is an inter-segment branch.

Eg: **JMP DWORD PTR [BX]** *; IP ← {DS:[BX], DS:[BX+1]},*
; CS ← {DS:[BX+2], DS:[BX+3]}

Addressing Modes for Stack Memory (optional, to be written only if asked)

Register Addressing Mode

Here the **operands** are specified **in registers** (ONLY 16-bit registers).

EG: PUSH BX *; Transfers BH at location pointed by SP-1 and BL at location*
; pointed by SP-2 in the Stack segment. Also SP ← SP - 2.

Register Indirect Addressing Mode

Here the **address** of the operand is specified **in** the **registers**.

Eg: PUSH [BX] *; Transfers a word from location pointed by BX and BX+1 in*
; data segment to SP-1 and SP-2 in Stack Segment.

1) Flag Addressing Mode

Here the contents of Flag register are transferred to and from the Stack.

Eg: PUSHF *; Transfers higher byte of Flag register to SP-1*
; and lower byte to SP-2 in the Stack Segment.

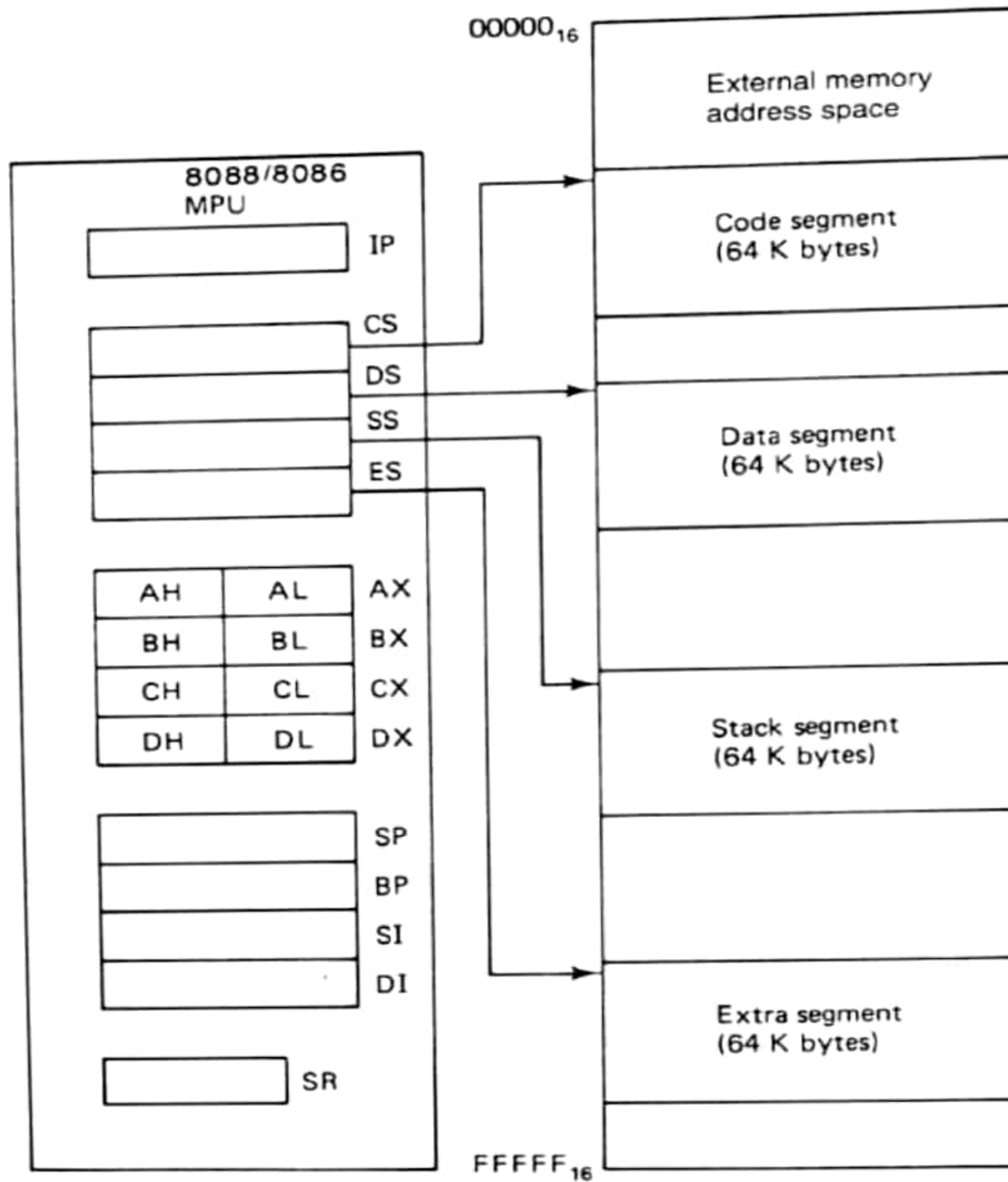
2) Segment Register Addressing Mode

Here the segment registers (except CS) are transferred to and from the Stack.

Eg: PUSH DS *; Transfers higher byte of DS register to location*
; SP - 1 and lower byte to SP-2 in the Stack Segment.

Scanned with CamScanner

8086 SOFTWARE MODEL



Scanned with CamScanner

8086 Instruction Set

Classification of Instruction Set of 8086

1) **Data Transfer Instructions**
   E.g.:: MOV, PUSH, POP
2) **Arithmetic Instructions**
   E.g.:: ADD, SUB, MUL
3) **Logic Instructions (Bit Manipulation Instructions)**
   E.g.:: AND, OR, XOR
4) **Shift Instructions & Rotate Instructions**
   E.g.:: ROL, RCL, ROR, SHL
5) **Program Execution and Transfer Instructions (Branch Instructions)**
   E.g.:: JMP, CALL, JC
6) **Iteration Control Instructions (Loop Instructions)**
   E.g.:: LOOP, LOOPZ, LOOPNE
7) **Processor Control Instructions (Instructions operating on Flags)**
   E.g.:: STC, CLC, CMC
8) **External Hardware Synchronization Instructions**
   E.g.:: LOCK, ESC, WAIT
9) **Interrupt Control Instructions**
   E.g.:: INT n, IRET, INTO (Interrupt on overflow)
10) **String Instructions**
   E.g.:: MOVSB, LODSB, STOSB

Scanned with CamScanner

Data Transfer Instructions

1) **MOV Destination, Source**
Moves a byte/word **from** the **source to** the **destination** specified in the instruction.
Source: *Register, Memory Location, Immediate Number*
Destination: Register, Memory Location
Both, source and destination cannot be memory locations.

```
Eg: MOV CX, 0037H        ; CX ← 0037H
    MOV BL, [4000H]      ; BL ← DS:[4000H]
    MOV AX, BX           ; AX ← BX
    MOV DL, [BX]         ; DL ← DS:[BX]
    MOV DS, BX           ; DS ← BX
```

2) **PUSH Source**
Push the **source** (word) **into** the **stack** and decrement the stack pointer by two.
The source MUST be a **WORD** (**16 bits**).
Source: *Register, Memory Location*

```
Eg: PUSH CX              ; SS:[SP-1] ← CH, SS:[SP-2] ← CL
                         ; SP ← SP – 2
    PUSH DS              ; SS:[SP-1, SP-2] ← DS
                         ; SP ← SP – 2
```

3) **POP Destination**
POP a **word from** the **stack** into the given destination and increment the Stack Pointer by 2. The destination MUST be a **WORD** (16 bits).
Destination: *Register [EXCEPT CS], Memory Location*

```
Eg: POP CX               ; CH ← SS:[SP], CL ← SS:[SP+1]
                         ; SP ← SP + 2
    POP DS               ; DS ← SS:[SP, SP+1]
                         ; SP ← SP + 2
```

***Please Note*: MOV, PUSH, POP** are the ONLY instructions that use the Segment Registers as operands {except CS}.

4) **PUSHF**
Push value of **Flag Register into stack** and decrement the stack pointer by 2.
Eg: **PUSHF** ; $SS:[SP-1] \leftarrow Flag_H$, $SS:[SP-2] \leftarrow Flag_L$, $SP \leftarrow SP - 2$

5) **POPF**
POP a **word from** the **stack into** the **Flag register**.
Eg: **POPF** ; $Flag_L \leftarrow SS:[SP]$, $Flag_H \leftarrow SS:[SP+1]$, $SP \leftarrow SP + 2$

6) **XCHG Destination, Source**
Exchanges a byte/word between the **source and** the **destination** specified in the instruction.
Source: *Register, Memory Location*
Destination: Register, Memory Location
Even here, both operands cannot be memory locations.

```
Eg: XCHG CX, BX          ; CX ←→ BX
    XCHG BL, CH          ; BL ←→ CH
```

Scanned with CamScanner

7) **XLATB / XLAT** (very important)

Move into AL, the **contents of** the **memory location** in Data Segment, **whose** effective **address** is **formed by** the **sum of BX and AL**.

Eg: **XLAT** ; *AL ← DS:[BX + AL]*

; i.e. if DS = 1000H; BX = 0200H; AL = 03H

; ∴ 10000 ... DS × 16

; + 0200 ... BX

; + 03 ... AL

; =**10203** ∴ **AL ← [10203H]**

Note: the difference between XLAT and XLATB

In XLATB there is no operand in the instruction.

E.g.:: XLATB

It works in an implied mode and does exactly what is shown above.

In XLAT, we can specify the name of the look up table in the instruction

E.g.:: XLAT SevenSeg

This will do the translation form the look up table called SevenSeg.

In any case, the base address of the look up table must be given by BX.

8) LAHF

Loads AH with **lower byte** of the **Flag** Register.

9) SAHF

Stores the contents of **AH into** the **lower byte** of the **Flag** Register.

10)LEA register, source

Loads Effective Address (offset address) **of** the **source into** the **given register**.

Eg: **LEA BX, Total** ; *BX ← offset address of Total in Data Segment.*

11)LDS destination register, source

Loads the **destination register and DS** register **with offset** address **and segment address** specified by the **source**.

Eg: **LDS BX, Total** ; *BX ← {DS:[Total], DS:[Total + 1]},*

; *DS← {DS: [Total + 2], DS:[Total + 3]}*

12)LES destination register, source

Loads the **destination register and ES** register with the **offset address and** the **segment address** indirectly specified by the **source**.

Eg: **LES BX, Total** ; *BX ← {DS:[Total], DS:[Total + 1]},*

; *ES← {DS: [Total + 2], DS:[Total + 3]}*

Scanned with CamScanner

I/O ADDRESSING MODES OF 8086 (5m – Important Question)

I/O addresses in 8086 can be either 8–bit or 16-bit

Direct Addressing Mode:

If we use **8-bit I/O address** we get a **range of 00H... FFH**.
This gives a total of **256 I/O ports.**
Here we use Direct addressing Mode, that is, the **I/O address is specified in the instruction.**

E.g.:: IN AL, 80H **; AL gets data from I/O port address 80H.**

This is also called **Fixed Port Addressing**.

Indirect Addressing Mode:

If we use **16-bit I/O address** we get a **range of 0000H... FFFFH**.
This gives a total of **65536 I/O ports.**
Here we use Indirect addressing Mode, that is, the **I/O address is specified by DX register.**

E.g.::
```
MOV DX, 2000H
IN AL, DX            ; AL gets data from I/O port address 2000H given by DX.
```

This is also called **Variable Port Addressing**.

13)IN destination register, source port
Loads the **destination register with** the contents of the **I/O port** specified by the source.

Source: It is an I/O port address.
If the address is 8-bit it will be given in the instruction by **Direct addressing mode.**
If it is a 16 bit address it will be given by DX register using **Indirect addressing mode.**

Destination: It has to be some form of "A" register, in which we will get data from the I/O device.
If we are getting 8-bit data, it will be AL or AH register.
If we are getting 16-bit data, it will be AX register.

Eg:
```
IN AL, 80H      ; AL gets 8-bit data from I/O port address 80H
IN AX, 80H      ; AX gets 16-bit data from I/O port address 80H
IN AL, DX       ; AL gets 8-bit data from I/O port address given by DX.
IN AX, DX       ; AX gets 16-bit data from I/O port address given by DX.
```

14)OUT destination port, source register
Loads the destination I/O port with the contents of the source register.

Eg:
```
OUT 80H, AL     ; I/O port 80H gets 8-bit data from AL
OUT 80H, AX     ; I/O port 80H gets 16-bit data from AX
OUT DX, AL      ; I/O port whose address is given by DX gets 8-bit data from AL
OUT DX, AX      ; I/O port whose address is given by DX gets 16-bit data from AX
```

Scanned with CamScanner

<u>Segment Overriding</u>

In every instruction, a particular segment register is accessed for the base address.

Eg: **MOV CL, [5000H]** *; CL ← DS:[5000H] as Data Seg is accessed by default*

However, we can also **override the segment** as follows:

Eg: **MOV CL, CS:[5000H]** *; Here CL ← CS:[5000H], this is* ***Segment Overriding****.*

By default, the address 5000H would have been an offset for the data segment, BUT here we **override** it with the Code segment as shown above.

<u>Another example</u>:

```
MOV BL, [BP]        ; BL ← SS:[BP] ... Normal
MOV BL, DS:[BP]     ; BL ← DS:[BP] ... Overriding
```

Scanned with CamScanner

Arithmetic Instructions

1) **ADD/ADC destination, source**
Adds the source to the destination and stores the **result** back **in** the **destination**.
Source: Register, Memory Location, Immediate Number
Destination: Register
Both, source and destination have to be of the same size.
ADC also adds the carry into the result.

```
Eg: ADD AL, 25H        ; AL ← AL + 25H
    ADD BL, CL         ; BL ← BL + CL
    ADD BX, CX         ; BX ← BX + CX
    ADC BX, CX         ; BX ← BX + CX + Carry Flag
```

2) **SUB/SBB destination, source**
It is similar to ADD/ADC except that it does subtraction.

3) **INC destination**
Adds "1" to the specified **destination**.
Destination: Register, Memory Location
Note: Carry Flag is NOT affected.

```
Eg: INC AX                ; AX ← AX + 1
    INC BL                ; BL ← BL + 1
    INC BYTE PTR [BX]     ; Increment the byte pointed by BX in the Data Segment
                          ; i.e. DS:[BX] ← DS;[BX] + 1
    INC WORD PTR [BX]     ; Increment word pointed by BX in the Data Segment
                          ; {DS:[BX], DS:[BX+1]}← {DS:[BX], DS:[BX+1]}+1
```

4) **DEC destination**
It is similar to INC. Here also Carry Flag is NOT affected.

5) **MUL source**(**unsigned** 8/16-bit register)
If the **source** is **8-bit**, it is **multiplied with AL** and the **result** is stored in **AX** (AH–higher byte, AL–lower byte)
If the **source** is **16-bit**, it is **multiplied** with **AX** and the **result** is stored in **DX-AX** (DX–higher byte, AX–lower byte)
Source: Register, Memory Location
MUL affects AF, PF, SF and ZF.

```
Eg:MUL BL                 ; AX ← AL × BL
   MUL BX                 ; DX-AX ← AX × BX
   MUL BYTE PTR [BX]      ; AX ← AL × DS:[BX]
```

6) **IMUL source**(**signed** 8/16-bit register)
Same as MUL except that the source is a SIGNED number.

7) **DIV source**(**unsigned** 8/16-bit register – divisor)
This instruction is used for **UNSIGNED** division.
Divides a **WORD by a BYTE**, OR a **DOUBLE WORD by a WORD.**
If the **divisor is 8-bit** then the **dividend is in AX** register.
After division, the **quotient is in AL** and the **Remainder in AH**.
If the **divisor is 16-bit** then the **dividend is in DX-AX** registers.
After division, the **quotient is in AX** and the **Remainder in DX**.

Scanned with CamScanner

Source: Register, Memory Location ☺ **For doubts contact Bharat Sir on 98204 08217**
ALL flags are **undefined** after DIV instruction.

Eg: **DIV BL** ; *AX ÷ BL :- AL ← Quotient; AH ← Remainder*
DIV BX ; *{DX,AX} ÷ BX :- AX ← Quotient; DX ← Remainder*

Please Note: If the divisor is 0 or the result is too large to fit in AL (or AX for 16-bit divisor), them 8086 does a Type 0 interrupt (Divide Error).

8) IDIV source(signed 8/16-bit register – divisor)
Same as DIV except that it is used for **SIGNED** division.

9) NEG destination
This instruction forms the **2's complement** of the destination, and stores it back in the destination.
Destination: Register, Memory Location
ALL condition flags are **updated**.
Eg: **Assume** AL= 0011 0101 = 35 H then
NEG AL ;*AL ← 1100 1011 = CBH. i.e. AL ← 2's Complement (AL)*

10)CMP destination, source
This instruction **compares the source with the destination**.
The source and the destination must be of the same size.
Comparison is **done by internally SUBTRACTING** the **SOURCE form DESTINATION**.
The result of this subtraction is NOT stored anywhere, instead the Flag bits are affected.
Source: Register, Memory Location, Immediate Value
Destination: Register, Memory Location
ALL condition flags are **updated**.
Eg: **CMP BL, 55H** ; *BL compared with 55H i.e. BL – 55H.*
CMP CX, BX ; *CX compared with BX i.e. CX – BX.*

11)CBW [Convert signed BYTE to signed WORD]
This instruction **copies sign of** the byte in **AL into** all the bits of **AH**.
AH is then called *sign extension of AL.*
No Flags affected.

Eg: Assume
AX = XXXX XXXX **1**001 0001
Then **CBW** gives
AX = **1111 1111 1**001 0001

12)CWD [Convert signed WORD to signed DOUBLE WORD]
This instruction **copies sign of** the **WORD** in **AX into** all the bits of **DX**.
DX is then called *sign extension of AX.*
No Flags affected.

Eg: Assume
AX = **1**000 0000 1001 0001
DX = XXXX XXXX XXXX XXXX
Then **CWD** gives
AX = **1**000 0000 1001 0001
DX = **1111 1111 1111 1111**

Note: Both CBW and CWD are used for Signed Numbers.

Scanned with CamScanner

Decimal Adjust Instructions

13) DAA [Decimal Adjust for Addition]

It makes the **result** in **packed BCD** form **after** BCD **addition** is performed.
It works **ONLY** on **AL** register.
All Flags are updated; **OF** becomes **undefined** after this instruction.
For AL register ONLY
If $D_3 - D_0 > 9$ OR Auxiliary Carry Flag is set => ADD 06H to AL.
If $D_7 - D_4 > 9$ OR Carry Flag is set => ADD 60H to AL.

Assume AL = 14H
CL = 28H
Then **ADD AL, CL** gives
AL = 3CH
Now **DAA** gives
AL = 42 (06 is added to AL as C > 9)
If you notice, $(14)_{10} + (28)_{10} = (42)_{10}$

14) DAS [Decimal Adjust for Subtraction]

It makes the **result** in **packed BCD** form **after** BCD **subtraction** is performed.
It works **ONLY** on **AL** register.
All Flags are updated; **OF** becomes **undefined** after this instruction.
For AL register ONLY
If $D_3 - D_0 > 9$ OR Auxiliary Carry Flag is set => Subtract 06H from AL.
If $D_7 - D_4 > 9$ OR Carry Flag is set => Subtract 60H from AL.

Assume AL = 86H
CL = 57H
Then **SUB AL, CL** gives
AL = 2FH
Now **DAS** gives
AL = 29 (06 is subtracted from AL as F > 9)
If you notice, $(86)_{10} - (57)_{10} = (29)_{10}$

ASCII Adjust Instructions (for the AX register ONLY)

15) AAA [ASCII Adjust for Addition]

It makes the **result** in **unpacked BCD form**.
In **ASCII** Codes, **0 … 9** are represented as **30 … 39**.
When we **add ASCII Codes**, we need to **mask** the **higher byte** (Eg: 3 of 39).
This can be **avoided** if we **use AAA** instruction **after the addition** is performed.
AAA updates the **AF** and the **CF**; But **OF, PF, SF, ZF** are **undefined** after the instruction.
Eg: Assume
AL = 0011 0100 … ASCII 4.
CL = 0011 1000 … ASCII 8.
Then **ADD AL, CL** gives
AL = 01101100
i.e. AL = 6CH … it is the Incorrect temporary Result

Scanned with CamScanner

Now **AAA** gives

AL = 0000 0010 ... Unpacked BCD for 2.
Carry = 1 ... this indicates that the answer is 12.

16)AAS [ASCII Adjust for Subtraction]

It makes the **result** in **unpacked BCD form**.
In **ASCII** Codes, **0 ... 9** are represented as **30 ... 39**.
When we **subtract ASCII Codes**, we need to **mask** the **higher byte** (Eg: 3 of 39).
This can be **avoided** if we **use AAS** instruction **after the subtraction** is performed.
AAS updates the **AF** and the **CF**; But **OF, PF, SF, ZF** are **undefined** after the instruction.

Eg: Assume

AL = 0011 1001 ... ASCII 9.
CL = 0011 0101 ... ASCII 5.

Then **SUB AL, CL** gives

AL = 0000 0100
i.e. AL = 04H

Now **AAS** gives

AL = 0000 0100 ... Unpacked BCD for 4.
Carry = 0 ... this indicates that the answer is 04.

17)AAM [BCD Adjust After Multiplication]

Before we multiply two ASCII digits, we mask their upper 4 bits.
Thus we have two unpacked BCD operands.
After the two unpacked BCD operands are multiplied, the AAM instruction converts this result into unpacked BCD form in the AX register.
AAS updates PF, SF ZF; But **OF, AF, CF** are **undefined** after the instruction.

Eg: Assume

AL = 0000 1001 ... unpacked BCD 9.
CL = 0000 0101 ... unpacked BCD 5.

Then **MUL CL** gives

AX = 0000 0000 0010 1101 = 002DH.

Now **AAM** gives

AX = 0000 0100 0000 0101 = 0405H.
This is 45 in the unpacked BCD form.

18)AAD [Binary Adjust before Division]

This instruction converts the unpacked BCD digits in AH and Al into a Packed BCD in AL.
AAD updates PF, SF ZF; But **OF, AF, CF** are **undefined** after the instruction.

Eg: Assume

CL = 07H.
AH = 04.
AL = 03.
∴ AX = 0403H ... unpacked BCD for $(43)_{10}$

Then **AAD** gives

AX = 002BH ... i.e. $(43)_{10}$

Now **DIV CL** gives (divide AX by unpacked BCD in CL)

AL = Quotient = 06 ... unpacked BCD
AH = Remainder = 01 ... unpacked BCD

Scanned with CamScanner

Logical Instructions [Bit Manipulation instructions]

1) **NOT destination**

This instruction forms the **1's complement** of the destination, and stores it back in the destination.
Destination: Register, Memory Location. **No Flags affected**.
Eg: **Assume** AL= 0011 0101
NOT AL ; *AL ← 1100 1010 ... i.e. AL = 1's Complement (AL)*

2) **AND destination, source**

This instruction **logically ANDs** the source with the destination and stores the **result in the destination**. Source and destination have to be of the same size.
Source: Register, Memory Location, Immediate Value
Destination: Register, Memory Location
PF, **SF**, **ZF affected**; **CF**, **OF ← 0**; **AF** becomes **undefined**.
Eg: **AND BL, CL** ; *BL ← BL AND CL*

3) **OR destination, source**

This instruction **logically Ors** the source with the destination and stores the **result in the destination**. Source and destination have to be of the same size.
Source: Register, Memory Location, Immediate Value
Destination: Register, Memory Location
PF, **SF**, **ZF affected**; **CF**, **OF ← 0**; **AF** becomes **undefined**.
Eg: **OR BL, CL** ; *BL ← BL OR CL*

4) **XOR destination, source**

This instruction **logically X-Ors** the source with the destination and stores the **result in the destination**. Source and destination have to be of the same size.
Source: Register, Memory Location, Immediate Value
Destination: Register, Memory Location
PF, **SF**, **ZF affected**; **CF**, **OF ← 0**; **AF** becomes **undefined**.
Eg: **XOR BL, CL** ; *BL ← BL XOR CL*

5) **TEST destination, source**

This instruction **logically ANDs** the source with the destination **BUT** the **RESULT** is **NOT STORED ANYWHERE**. **ONLY** the **FLAG** bits are **AFFECTED**.
Source: Register, Memory Location, Immediate Value
Destination: Register, Memory Location
PF, **SF**, **ZF affected**; **CF**, **OF ← 0**; **AF** becomes **undefined**.
Eg: **TEST BL, CL** ; *BL AND CL; result not stored; Flags affected.*
*Note: Don't forget this instruction because it will be used later in **multiprocessor systems**!*

Scanned with CamScanner

SHIFT INSTRUCTIONS

1) SAL/SHL destination, count
LEFT-Shifts the **bits of destination**.
MSB shifted **into** the **CARRY**.
LSB gets a **0**.

Bits are shifted 'count' number of times.
If count = 1, it is directly specified in the instruction.
If count > 1, it has to be given using CL Register.

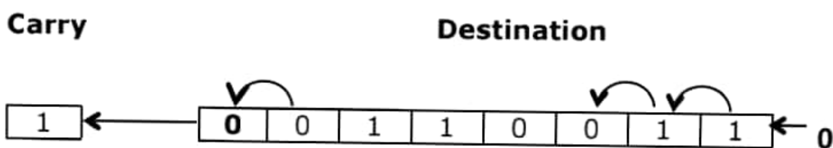
Destination: Register, Memory Location. #Please refer Bharat Sir's Lecture Notes for this ...

Eg: **SAL BL, 1** *; Left-Shift BL bits, once.*

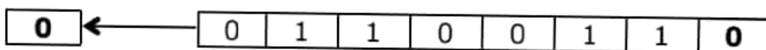
Assume:

Before Operation: BL = 0011 0011 and CF = 1

Carry **Destination**



After Operation: BL = 0110 0110 and CF = 0



More examples:
MOV CL, 05H *; Load number of shifts in CL register.*
SAL BL, CL *; Left-Shift BL bits CL (5) number of times.*

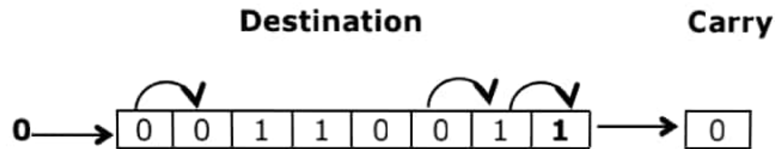
2) SHR destination, count
RIGHT-Shifts the bits of destination.
MSB gets a **0** (∴ Sign is lost).
LSB shifted **into** the **CARRY**.

Bits are shifted 'count' number of times.
If count is 1, it is directly specified in the instruction.
If count > 1, it has to be given using CL register.
Eg: **SHR BL, 1** *; Right-Shift BL bits, once.*

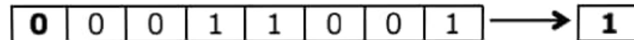
Scanned with CamScanner

Assume:

Before Operation: BL = 0011 0011 and CF = 0



After Operation: BL = 00011 1001 and CF = 1

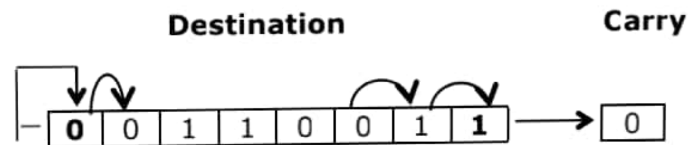


3) **SAR destination, count**
RIGHT-Shifts the bits of destination.
MSB placed **in MSB itself** (∴ Sign is preserved).
LSB shifted **into** the **CARRY**.

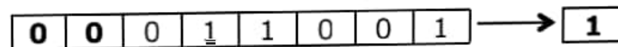
Bits are shifted 'count' number of times.
If count is 1, it is directly specified in the instruction.
If count > 1 it has to be given using CL register. ☺ For doubts contact Bharat Sir on 98204 08217
Destination: Register, Memory Location

Eg: **SAR BL, 1** *; Right-Shift BL bits, once.*

Assume:
Before Operation: BL = 0011 0011 and CF = 0



After Operation: BL = 0001 1001 and CF = 1

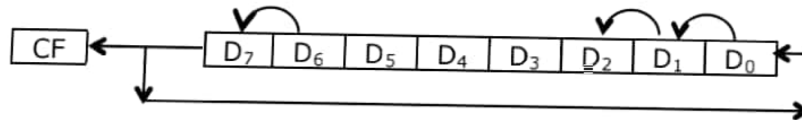


Scanned with CamScanner

ROTATE INSTRUCTIONS

1) ROL destination, count

LEFT-Shifts the bits of destination.
MSB shifted **into** the **CARRY**.
MSB also goes **to LSB**.
Bits are shifted 'count' number of times.
If count = 1, it is directly specified in the instruction.
If count > 1, it has to be loaded in the CL register, and CL gives the count in the instruction.
Destination: Register, Memory Location
Eg: **ROL BL, 1** *; Left-Shift BL bits once.*

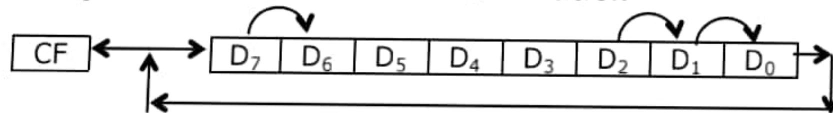


More examples:

MOV CL, 05H *; Load number of shifts in CL register.*
ROL BL, CL *; Left-Shift BL bits CL (5) number of times.*

2) ROR destination, count

RIGHT-Shifts the bits of destination.
LSB shifted **into** the **CARRY**.
LSB also goes **to MSB**.
Bits are shifted 'count' number of times.
If count = 1, it is directly specified in the instruction.
If count > 1, it has to be loaded in the CL register, and CL gives the count in the instruction.
Eg:
ROR BL, 1 *; Right-Shift BL bits once.*



Scanned with CamScanner

3) RCL destination, count

LEFT-Shifts the bits of destination.

MSB shifted **into** the Carry Flag **(CF)**.

CF goes **to LSB**.

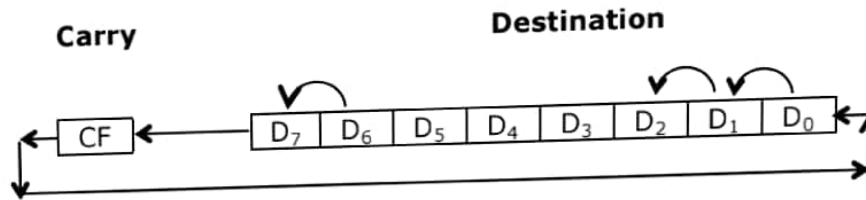
Bits are shifted 'count' number of times.

If count = 1, it is directly specified in the instruction.

If count > 1, it has to be loaded in the CL register, and CL is specified as the count in the instruction.

Destination: Register, Memory Location

Eg: **RCL BL, 1** *; Left-Shift BL bits once.*



4) RCR destination, count

RIGHT-Shifts the bits of destination.

LSB shifted **into** the **CF**.

CF goes **to MSB**.

Bits are shifted 'count' number of times.

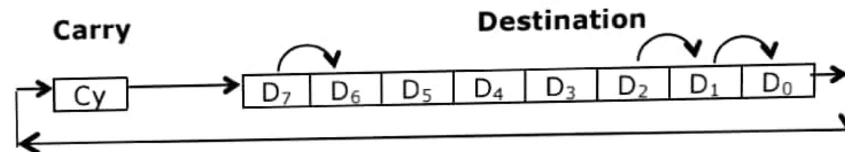
If count = 1, it is directly specified in the instruction.

If count > 1, it has to be loaded in the CL register, and CL is specified as the count in the instruction.

Destination: Register, Memory Location

Eg:

RCR BL, 1 *; Right-Shift BL bits once.*



More examples:

MOV CL, 05H *; Load number of shifts in CL register.*

RCR BL, CL *; Right-Shift BL bits CL (5) number of times.*

Scanned with CamScanner

PROGRAM EXECUTION AND TRANSFER INSTRUCTIONS

These instructions cause a branch in the program sequence.

There are 2 main types of branching:

i. Near branch
ii. Far Branch

i. **Near Branch**
This is an **Intra-Segment Branch** i.e. the branch is to a new location within the current segment only.
Thus, **only** the value of **IP needs to be changed**.
If the Near Branch is in the **range of –128 to 127**, then it is called as a **Short Branch**.

ii. **Far Branch**
This is an **Inter-Segment Branch** i.e. the branch is to a new location in a different segment.
Thus, the values of **CS and IP need to be changed**.

JMP (Unconditional Jump)

INTRA-Segment (NEAR) JUMP

The Jump address is specified in two ways:

1) INTRA-Segment Direct Jump

The new Branch location is specified directly in the instruction
The new address is calculated by **adding** the 8 or16-bit **displacement** to the IP.
The CS does not change.
A +ve displacement means that the Jump is ahead (forward) in the program.
A -ve displacement means that the Jump is behind (backward) in the program.
It is also called as *Relative Jump*.

Eg: **JMP Prev** ; *IP ← offset address of "Prev".*
JMP Next ; *IP ← offset address of "Next".*

2) INTRA-Segment Indirect Jump

The New Branch address is specified indirectly through a **register** or a **memory location**.
The value in the IP is **replaced** with the new value.
The CS does not change.

Eg: **JMP WORD PTR [BX]** ; *IP ← {DS:[BX], DS: [BX+1]}*

INTER-Segment (FAR) JUMP

The Jump address is specified in two ways:

3) INTER-Segment Direct Jump

The new Branch location is **specified directly** in the instruction
Both **CS and IP get new values**, as this is an inter-segment jump.

Eg: **Assume NextSeg** is a label pointing to an instruction in a **different segment.**
JMP NextSeg ; *CS and IP get the value from the label NextSeg.*

4) INTER-Segment Indirect Jump

The new Branch location is **specified indirectly** through a **register** or a **memory location.**
Both **CS and IP get new values**, as this is an inter-segment jump.

Eg:**JMP DWORD PTR [BX]** ; *IP ← {DS:[BX], DS: [BX+1]},*
; *CS ← {DS:[BX+2], DS:[BX+3]}*

Scanned with CamScanner

JCondition (Conditional Jump)

This is a conditional branch instruction.
If condition is **TRUE**, then it is **similar to** an **INTRA-Segment Direct Jump.**
If condition is FALSE, then branch does not take place and the next sequential instruction is executed.
The destination must be in the range of –128 to 127 from the address of the instruction (i.e. **ONLY SHORT Jump**).
Eg: **JNC Next** *; Jump to Next If Carry Flag is not set (CF = 0).*
The various conditional jump instructions are as follows:

Mnemonic	Description	Jump Condition
	Common Operations	
JC	Carry	**CF = 1**
JNC	Not Carry	CF = 0
JE/JZ	Equal or Zero	**ZF = 1**
JNE/JNZ	Not Equal or Not Zero	ZF = 0
JP/JPE	Parity or Parity Even	**PF = 1**
JNP/JPO	Not Parity or Parity Odd	PF = 0
	Signed Operations	
JO	Overflow	**OF = 1**
JNO	Not Overflow	OF = 0
JS	Sign	**SF = 1**
JNS	Not Sign	SF = 0
JL/JNGE	Less	**(SF Ex-Or OF) = 1**
JGE/JNL	Greater or Equal	(SF Ex-Or OF) = 0
JLE/JNG	Less or Equal	**((SF Ex-Or OF) + ZF) = 1**
JG/JNLE	Greater	((SF Ex-Or OF) + ZF) = 0
	Unsigned Operations	
JB/JNAE	Below	**CF = 1**
JAE/JNB	Above or Equal	CF = 0
JBE/JNA	Below or Equal	**(CF Ex-Or ZF) = 1**
JA/JNBE	Above	(CF Ex-Or ZF) = 0

CALL (Unconditional CALL)

CALL is an instruction that transfers the program control to a sub-routine, with the intention of coming back to the main program.
Thus, in CALL 8086 **saves the address of the next instruction into the stack** before branching to the sub-routine.
At the end of the subroutine, control transfers back to the main program using the return address from the stack.
There are two types of CALL: Near CALL and Far CALL.

INTRA-Segment (NEAR) CALL

The **new subroutine** called must be **in the same segment** (hence intra-segment).
The CALL **address** can be **specified directly** in the instruction **OR indirectly** through Registers or Memory Locations.
The following sequence is executed for a NEAR CALL:

1. i. 8086 will **PUSH Current IP** into the Stack.
2. ii. **Decrement SP by 2**.
3. iii. **New value** loaded **into IP**.

Scanned with CamScanner

iv. **Control transferred** to a subroutine within the same segment.

Eg: **CALL subAdd** *; {SS:[SP-1], SS:[SP-2]} ← IP, SP ← SP – 2,*
; IP ← New Offset Address of subAdd.

INTER-Segment (FAR) CALL

The **new subroutine** called is in **another segment** (hence inter-segment).
Here CS and IP both get new values.
The CALL address can be specified directly OR through Registers or Memory Locations.
The following sequence is executed for a Far CALL:

i. **PUSH CS** into the Stack.
ii. **Decrement SP** by 2.
iii. **PUSH IP** into the Stack.
iv. **Decrement SP** by 2.
v. **Load CS** with new segment address.
vi. **Load IP** with new offset address.
vii. **Control transferred** to a subroutine in the new segment.

Eg: **CALL subAdd** *; {SS:[SP-1], SS:[SP-2]} ← CS, SP ← SP – 2,*
; {SS:[SP-1], SS:[SP-2]} ← CS, SP ← SP – 2,
; CS ← New Segment Address of subAdd,
; IP ← New Offset Address of subAdd.

There is **NO PROVISION for Conditional CALL**.

RET --- Return instruction

RET instruction causes the control to return to the main program from the subroutine.

Intrasegment-RET

Eg: RET ; IP ← SS:[SP], SS:[SP+1]
; SP ← SP + 2

RET n ; IP ← SS:[SP], SS:[SP+1]
; SP ← SP + 2 + n

Intersegment-RET

Eg: RET ; IP ← SS:[SP], SS:[SP+1]
; CS ← SS:[SP+2], SS:[SP+3]
; SP ← SP + 4

RET n ; IP ← SS:[SP], SS:[SP+1]
; CS ← SS:[SP+2], SS:[SP+3]
; SP ← SP + 4 + n

Please Note: The programmer writes the intra-seg and Inter-seg RET instructions in the same way. It is the assembler, which distinguishes between the two and puts the right opcode.
#Please refer Bharat Sir's Lecture Notes for this ...

Scanned with CamScanner

Differentiate between

	JMP INSTRUCTION	**CALL INSTRUCTION**
1	JMP instruction is used to **jump to a new location** in the program and continue	Call instruction is used to **invoke a subroutine, execute it and then return** to the main program.
2	A jump simply **puts the branch address into IP**.	A call **first stores the return address into the stack** and then loads the branch address into IP.
3	In 8086 Jumps can be either **unconditional or conditional**.	In 8086, Calls are only **unconditional**.
4	Does **not use the stack**	**Uses the stack**
5	Does **not need a RET** instruction.	**Needs a RET** instruction to return back to main program.

Differentiate between

	PROCEDURE (FUNCTION)	**MACRO**
1	A procedure (Subroutine/ Function) is a set of instruction needed repeatedly by the program. It is **stored as a subroutine and invoked from several places by the main program.**	A Macro is similar to a procedure but is not invoked by the main program. Instead, the **Macro code is pasted into the main program wherever the macro name is written in the main program.**
2	A subroutine is **invoked by a CALL** instruction and control returns by a RET instruction.	A Macro is simply accessed by **writing its name**. The entire macro code is pasted at the location by the assembler.
3	**Reduces the size** of the program	**Increases the size** of the program
4	**Executes slower** as time is wasted to push and pop the return address in the stack.	**Executes faster** as return address is not needed to be stored into the stack, hence push and pop is not needed.
5	**Depends on the stack**	**Does not depend on the stack**

Scanned with CamScanner

Type 1) **Iteration Control Instructions**

These instructions **cause** a series of **instructions to be executed repeatedly**.
The **number of iterations** is loaded **in CX** register.
CX is **decremented by 1**, after every iteration. Iterations occur **until CX = 0**.
The **maximum difference between** the **address** of the instruction and the address of the Jump **can be 127**.

1) LOOP Label

Jump to specified label if CX not equal to 0; and decrement CX.

Eg:

```
        MOV CX, 40H
BACK:   MOV AL, BL
        ADD AL, BL
        .
        .
        .
        MOV BL, AL
        LOOP BACK       ; Do CX ← CX – 1.
                        ; Go to BACK if CX not equal to 0.
```

2) LOOPE/LOOPZ Label (Loop on Equal / Loop on Zero)

Same as above except that looping occurs ONLY if Zero Flag is set (i.e. ZF = 1)

Eg:

```
        MOV CX, 40H
BACK:   MOV AL, BL
        ADD AL, BL
        .
        .
        .
        MOV BL, AL
        LOOPZ BACK      ; Do CX ← CX – 1.
                        ; Go to BACK if CX not equal to 0 and ZF = 1.
```

3) LOOPNE/LOOPNZ Label (Loop on NOT Equal / Loop on NO Zero)

Same as above except that looping occurs ONLY if Zero Flag is reset (i.e. ZF = 0)

Eg:

```
        MOV CX, 40H
BACK:   MOV AL, BL
        ADD AL, BL
        .
        .
        .
        MOV BL, AL
        LOOPZ BACK      ; Do CX ← CX – 1.
                        ; Go to BACK if CX not equal to 0 and ZF = 0.
```

Scanned with CamScanner

Type 2) **Processor Control / Machine Control Instructions**

(these are instructions that directly operate on Flag Reg)

In the exam first explain the following instructions: **PUSHF, POPF, LAHF and SAHF**

For Carry Flag

1) STC
This instruction **sets** the **Carry Flag**. No Other Flags are affected.

2) CLC
This instruction **clears** the **Carry Flag**. No Other Flags are affected.

3) CMC
This instruction **complements** the **Carry Flag**. No Other Flags are affected.

For Direction Flag

4) STD
This instruction **sets** the **Direction Flag**. No Other Flags are affected.

5) CLD
This instruction **clears** the **Direction Flag**. No Other Flags are affected.

For Interrupt Enable Flag

6) STI
This instruction **sets** the **Interrupt Enable Flag**. No Other Flags are affected.

7) CLI
This instruction **clears** the **Interrupt Enable Flag**. No Other Flags are affected.

Note: There is no direct way to alter TF. It can be altered through program as follows:

To set TF:

```
PUSHF           ; push contents of Flag register into the stack
POP BX          ; pop contents of flag reg from the stack-top into BX
OR BH, 01H      ; set the bit corresponding to TF, in the BH register
PUSH BX         ; push the modified BX register into the stack
POPF            ; pop the modified contents into flag register.
```

To reset TF:

```
PUSHF           ; push contents of Flag register into the stack
POP BX          ; pop contents of flag reg from the stack-top into BX
AND BH, FEH     ; reset the bit corresponding to TF, in the BH register
PUSH BX         ; push the modified BX register into the stack
POPF            ; pop the modified contents into flag register.
```

Scanned with CamScanner

Type 3) **External Hardware Synchronization Instructions**

1) **ESC**

This is an 8086 **instruction-prefix** used to **indicate that the current instruction is for the 8087 NDP**.
We write a *homogeneous program* for the two processors 8086 and 8087.
Instructions are fetched by 8086 into its queue.
8087 duplicates the instruction queue of 8086 and monitors this queue.
When an instruction with **ESC prefix** (binary code 11011) is **encountered**, **8087 is activated**, and hence **it executes the instruction**.
8086 treats the instruction as NOP.
ESC has to be written before each 8087 instruction.

2) **WAIT**

This instruction is used to synchronize 8086 with the 8087 Co-Processor via the $\overline{\textbf{TEST}}$ input pin of 8086. Whenever 8087 is busy it puts a "1" on its BUSY o/p line connected to the $\overline{\textbf{TEST}}$ i/p of the μP.

The WAIT instruction makes the μP check the $\overline{\textbf{TEST}}$ pin.

If the μP checks the $\overline{\textbf{TEST}}$ pin and finds a "1" on it, 8086 understands that 8087 is busy and so it enters wait state. Here it does no processing.

It can **come out** of this idle state **in 2 ways**:

i. **$\overline{\textbf{TEST}}$ input is made low**

i.e. 8087 is no longer busy.
This takes 8086 completely out of the IDLE state.

ii. **Valid Interrupt on INTR or NMI**

In this case 8086 **exits wait state**, **executes the ISR** for the interrupt, and then **re-enters the WAIT state**. (*This is because the address of the WAIT instruction is what was pushed into the stack before executing the ISR.)*

Thus if **we write a WAIT instruction before every 8087 instruction**, we can ensure that 8087 is ready for executing its own instruction whenever it arrives.

WAIT can also be written before an 8086 instruction that requires the result of a previous 8087 operation.

Scanned with CamScanner

3) LOCK

This is an 8086 **instruction prefix.**

It **prevents any external bus master from taking control of the system** bus during execution of the instruction, which has a **LOCK** prefix.

It causes 8086 to activate the $\overline{\textbf{LOCK}}$ signal so that no other bus master takes control of the system bus. ☺ **For doubts contact Bharat Sir on 98204 08217**

4) NOP

There is **no operation performed** while executing this instruction.

8086 requires 3 T-States for this instruction.

It is mainly used to insert time delays, and can also be used while debugging.

5) HLT

This instruction causes 8086 to **stop fetching any more instructions**.

8086 enters **Halt state**.

8086 can **come out** of this halt state only if there is a **valid hardware interrupt** (NMI or INTR) or **by reset**.

Scanned with CamScanner

Type 4) **Interrupt Control Instructions**

1) INT Type

This instruction causes an interrupt of the given type. The **'Type'** can be a number between **0 ... 255**.
The following action takes place:

i. **PUSH Flag** Register onto the Stack. SP decremented by 2.
ii. **IF** and **TF** are **cleared**. No other flags are affected.
iii. **PUSH CS** onto the Stack. SP decremented by 2.
iv. **PUSH IP** onto the Stack. SP decremented by 2. ∴ In all **SP decremented by 6**.
v. **New value of IP** taken from location **type × 4**.
   Eg: INT 1 ; IP ← {[00004] and [00005]} (as 1 × 4 = 00004H)
vi. **New value of CS** taken from location **(type × 4) + 2**.
   Eg: INT 1 ; CS ← {[00006] and [00007]}
   Execution of ISR begins from the address formed by new values of CS and IP.

2) INTO (Interrupt on Overflow)

This instruction causes an interrupt of **type 4**, **ONLY if Overflow Flag (OF) is set**.
The above sequence is followed and the control is transferred to the location pointed by 00010H.
Eg: **INTO** ; *If OF = 1 then execute INT 4.*
Please Note:- This is INTO (O for Overflow) and NOT INT 0 (i.e. Type 0 ==> Zero Divide Interrupt).

3) IRET (Return from ISR)

This instruction causes the 8086 to return to the main program from an ISR.
The following action takes place:

i. **POP IP** from the Stack.
   SP incremented by 2.
ii. **POP CS** from the Stack.
   SP incremented by 2.
iii. **POP Flag Register** from the Stack.
   SP incremented by 2.
   ∴ In all **SP incremented by 6**.

Execution of the **Main Program** continues from the address formed values of CS and IP restored from the stack.

Please Note:- The original value of TF and IF are restored from the Stack. Also note that to come back from an ISR, the programmer must use the IRET instruction and not the normal RET instruction as the RET instruction will not POP back the Flag.

Scanned with CamScanner

Type 5) **String Instructions of 8086** *(Very Important ※ 10m)*

A **String** is a **series of bytes** stored sequentially in the memory. String Instructions operate on such "Strings".
The **Source String is at a location pointed by SI in the Data Segment.**
The **Destination String is at a location pointed by DI in the Extra Segment.**
The Count for String operations is always given by CX.
Since CX is a 16-bit register we can transfer max 64 KB using a string instruction.
SI and/or DI are **incremented/decremented** after each operation depending upon the direction flag **"DF"** in the flag register.
If **DF = 0**, it is **auto increment**. This is done by **CLD instruction**.
If **DF = 1**, it is **auto decrement**. This is done by **STD instruction**.

1) MOVS: MOVSB/MOVSW (*Move String*)

It is used to **transfer** a word/byte **from data segment to extra segment**.
The offset of the source in data segment is in SI.
The offset of the destination in extra segment is in DI.
SI and DI are incremented / decremented depending upon the direction flag.

Eg: **MOVSB** ; *ES:[DI] ← DS:[SI] ... byte transfer*
; *SI ← SI ± 1 ... depending upon DF*
; *DI ← DI ± 1 ... depending upon DF*

MOVSW ; *{ES:[DI], ES:[DI + 1]} ← {DS:[SI], DS:[SI + 1]}*
; *SI ← SI ± 2*
; *DI ← DI ± 2*

2) LODS: LODSB/LODSW (*Load String*)

It is used to **Load AL** (or AX) register with a byte (or word) **from data segment**.
The offset of the source in data segment is in SI.
SI is incremented / decremented depending upon the direction flag (DF).

Eg: **LODSB** ; *AL ← DS:[SI] ... byte transfer*
; *SI ← SI ± 1 ... depending upon DF*

LODSW ; *AL ← DS:[SI]; AH ← DS:[SI + 1]*
; *SI ← SI ± 2*

Scanned with CamScanner

3) STOS: STOSB/STOSW (*Store String*)

It is used to **Store AL** (or AX) **into** a byte (or word) in the **extra segment**.
The offset of the source in extra segment is in DI.
DI is incremented / decremented depending upon the direction flag (DF).

Eg: **STOSB** ; *ES:[DI] ← AL ... byte transfer*
; *DI ← DI ± 1 ... depending upon DF*

STOSW ; *ES:[DI] ← AL; ES:[DI+1] ← AH ... word transfer*
; *DI ← DI ± 2 ... depending upon DF*

4) CMPS: CPMSB/CMPSW (*Compare String*)

It is used to **compare** a **byte** (or word) **in** the **data segment with** a **byte** (or word) **in** the **extra segment**.
The offset of the byte (or word) in data segment is in SI. The offset of the byte (or word) in extra segment is in DI.
SI and DI are incremented / decremented depending upon the direction flag.
Comparison is done by subtracting the byte (or word) from extra segment from the byte (or word) from Data segment.
The Flag bits are affected, but the result is not stored anywhere.

Eg :**CMPSB** ; *Compare DS:[SI] with ES:[DI] ... byte operation*
; *SI ← SI ± 1 ... depending upon DF*
; *DI ← DI ± 1 ... depending upon DF*

CMPSW ; *Compare {DS:[SI], DS:[SI+1]}*
; *with {ES:[DI], ES:[DI+1]}*
; *SI ← SI ± 2 ... depending upon DF*
; *DI ← DI ± 2 ... depending upon DF*

5) SCAS: SCASB/SCASW (*Scan String*)

It is used to **compare** the contents of **AL** (or AX) **with** a **byte** (or word) **in** the **extra segment**.
The offset of the byte (or word) in extra segment is in DI.
DI is incremented / decremented depending upon the direction flag (DF). Comparison is done by subtracting a byte (or word) from extra segment from AL (or AX). The Flag bits are affected, but the result is not stored anywhere.

Eg: **SCASB** ; *Compare AL with ES:[DI] ... byte operation*
; *DI ← DI ± 1 ... depending upon DF*

SCASW ; *Compare {AX} with {ES:[DI], ES:[DI+1]}*
; *DI ← DI ± 1 ... depending upon DF*

Scanned with CamScanner

REP (*Repeat prefix used for string instructions*)

This is an **instruction prefix**, which can be used in string instructions.
It can be **used with string instructions only**.
It **causes** the **instruction** to be **repeated CX number** of times.
After each execution, the **SI** and **DI** registers are **incremented/decremented** based on the **DF** (Direction Flag) in the Flag register **and CX is decremented.**
i.e. **DF = 1; SI, DI decrements.** #Please refer Bharat Sir's Lecture Notes for this ...

Thus, it is important that before we use the REP instruction prefix the following steps must be carried out:
CX must be initialized to the Count value. If **auto decrementing** is required, **DF** must be **set using STD** instruction **else cleared** using **CLD** instruction.

```
EG:       MOV CX, 0023H
          CLD
    REP   MOVSB
```

The above section of a program will cause the following string operation

ES:[DI] ← DS:[SI], SI ← SI + 1, DI ← DI + 1, CX ← CX – 1

to be executed 23H times (as CX = 23H) in auto incrementing mode (as DF is cleared).

6) REPZ/REPE (*Repeat on Zero/Equal*)
It is a conditional repeat instruction prefix.It behaves the same as a REP instruction **provided** the Zero Flag is set (i.e. **ZF = 1**).It is used with CMPS instruction.

☺ **For doubts contact Bharat Sir on 98204 08217**

7) REPNZ/REPNE (*Repeat on No Zero/Not Equal*)
It is a conditional repeat instruction prefix.It behaves the same as a REP instruction **provided** the Zero Flag is reset (i.e. **ZF = 0**).It is used with SCAS instruction.

Please Note: 8086 instruction set has only 3 instruction prefixes :

1) **ESC** *(to identify 8087 instructions)*
2) **LOCK** *(to lock the system bus during an instruction)*
3) **REP** *(to repeatedly execute string instructions)*

For a question on instruction prefixes (asked repeatedly), explain the above in detail.

Scanned with CamScanner

INSTRUCTION FORMAT TEMPLATE OF 8086

Instructions in 8086 can be of size 1 byte to 6 bytes.
The distribution of the bytes is as follows

byte	7	6	5	4	3	2	1	0	
1	opcode						d	w	**Opcode byte**
2	mod		reg			r/m			**Addressing mode byte**
3	[optional]								**low disp, addr, or data**
4	[optional]								**high disp, addr, or data**
5	[optional]								**low data**
6	[optional]								**high data**

Opcode Byte

The first byte is called the "opcode byte".
It has a 6-bit opcode that indicates the operation to be performed.
It has two more bits "d" and "w"

d: direction
1 = data moves from operand specified by r/m to operand specified by reg.
0 = data moves from operand specified by reg to operand specified by r/m.

w: word/ byte
1: data is a word: 16-bits
0: data is a byte: 8-bits

Addressing Mode Byte

mod (2 bits):
These are called "mode" bits. They decide how r/m is interpreted.

00: r/m is a memory operand, but no displacement
01: r/m is a memory operand, with 8-bit displacement
10: r/m is a memory operand, with 16-bit displacement
11: r/m is a register operand

Scanned with CamScanner

reg (3 bits):

This specifies the register used as the first operand, which may act as source or destination depending upon the "d"(direction) bit.

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

r/m (3 bits):

This specifies the second operand which may either be a register or a memory location depending upon the "mod" bits.

R/M	MOD				
	00	01	10	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	16-bit address	[BP]+d8	[BP]+d16	DH	SI
111	[BX]	[BX]+d8	[BX]+d16	BH	DI

Scanned with CamScanner

ASSEMBLER DIRECTIVES / PSEUDO OPCODES

Assembly language has 2 types of statements:

1. **Executable**: Instructions that are translated into Machine Code by the assembler.
2. **Assembler Directives**:
   Statements that direct the assembler to do some special task.
   No M/C language code is produced for these statements.
   Their main task is to inform the assembler about the start/end of a segment, procedure or program, to reserve appropriate space for data storage etc.
   Some of the assembler directives are listed below

1. **DB** (Define Byte) ; Used to define a Byte type variable.
   Eg: SUM DB 0 ; Assembler reserves 1 Byte of memory for the variable ; named SUM and initialize it to 0.
2. **DW** (Define Word) ; Used to define a Word type variable (2 Bytes).
3. **DD** (Double Word) ; Used to define a Double Word type variable (4 Bytes).
4. **DQ** (Quad Word) ; Used to define a Quad Word type variable (8 Bytes).
5. **DT** (Ten Bytes) ; Used to define 10 Bytes to a variable (10 Bytes).
6. **DUP()** ; Copies the contents of the bracket followed by this ; keyword into the memory location specified before it.
   Eg: LIST DB 10 DUP (0) ; Stores LIST as a series of 10 bytes initialized to Zero.
7. **SEGMENT** ; Used to indicate the beginning of a segment.
8. **ENDS** ; Used to indicate the end of a segment.
9. **ASSUME** ; Associates a logical segment with a processor segment.
   Eg: Assume CS:Code ; Makes the segment "Code" the actual Code Segment.
10. **PROC** ; Used to indicate the beginning of a procedure.
11. **ENDP** ; Used to indicate the end of a procedure.
12. **END** ; Used to indicate the end of a program.
13. **EQU** ; Defines a constant
    E.g.:: AREA EQU 25H ; Creates a constant by the name AREA with a value 25H
    Do remember, in the class, you have been clearly made to understand the difference between using a variable and using a constant.

Scanned with CamScanner

14. EVEN / ALIGN ; Ensures that the data will be stored by the assembler in the memory in an aligned form. Aligned data works faster as it can be accessed in One cycle. Misaligned data, though is valid, requires two cycles to be accessed hence works slower.

15. OFFSET ; Can be used to tell the assembler to simply substitute the offset address of any variable.
E.g.:: MOV Si, OFFSET String1 ; SI gets the offset address of String1

16. Start ; It's the label from where the microprocessor to start executing the program

17. Model Directives

.MODEL SMALL **;** All Data Fits in one 64 KB segment.
All Code fits in one 64 KB Segment

.MODEL MEDIUM **;** All Data Fits in one 64 KB segment.
Code may be greater than 64 KB

.MODEL LARGE **;** Both Data and Code may be greater than 64 KB

Combined Example:

```
Data    SEGMENT
        LIST DB 10 DUP (0); Stores LIST as a series of 10 bytes initialized to zero    ...
Data    ENDS

Code    SEGMENT
        Assume CS: Code, DS: Data          ; Makes Code → Code Segment
                                           ; and Data → Data Segment.
Start:          ...
                ...
                ...

Code    ENDS
        END Start
```

There are many more assembler directives
Please refer class notes and VIVA booklet for the same.

Scanned with CamScanner

INT 21H (DOS Interrupt)
Important for College Practicals and Viva

1) DOS provides **various internal interrupts** which are used by the system programmer. The **most commonly used** interrupt is **INT 21H**. **#For doubts contact Bharat Sir on 98204 08217**
2) It **invokes inbuilt DOS functions** which can be used to perform tasks such as reading a user input char from the screen, displaying result on the screen, exiting the program etc.
3) While calling the INT21H Dos interrupt, we must **first assign a correct value in AH** register.
4) The value in the AH register **selects the INT 21H function** which is required by the user.
5) The most commonly used INT 21H functions are as shown:

Task	Method	Comment
How to **input** a **character** from the screen	**Mov AH, 01H** **INT 21H**	Takes the user input character from the screen. Returns the ASCII value of the character in AL register. If AL=0, then a control key was pressed.
How to **input** a **string** from the screen	**Mov AH, 0AH** **LEA DX, string** **INT 21H**	0AH is the parameter for the input string function. The string will be stored from the offset address given by DX.
How to **display** a **character** on the screen	**Mov AH, 02H** **Mov DL, char** **INT 21H**	02H is the parameter for the display char function. DL should contain the char to be displayed.
How to **display** a **string** on the screen	**Mov AH, 09H** **LEA DX, string** **INT 21H**	09H is the parameter for the display string function. DX should contain the offset address of the output string.
How to **terminate** the program	**Mov AH, 4CH** **Mov AL, 00H** **INT 21H**	4CH is the parameter for the terminate function. The return code is placed by the system in AL register. If AL is 00h then the program terminated without an error.

Scanned with CamScanner

PARAMETER PASSING TECHNIQUES USED IN ASSEMBLY LANGUAGE PROGRAMMING

A Parameter is any value passed by the main program to the subroutine.
Passing parameters makes the subroutine **more flexible** and can tremendously **enhance the usage** of the subroutine.
E.g.:: If a subroutine always finds factorial of 10, it is rigid. But if it can find factorial of "N" and "N" can be any number passed by the main program, then the same subroutine can find factorial of any number and hence becomes more usable.
There are 4 popular methods of passing parameters to subroutines.

1. USING REGISTERS

Here, the main program **stores the parameter into a register** like DL, and Calls the Subroutine.
Now **Subroutine takes the parameter value form DL register** and works on it.

```
Main:
MOV    DL, 25H        ; (parameter value 25H stored in DL)
Call   Sub
...

Sub:
MOV    AL, DL         ; Subroutine takes parameter value from DL Register
...
RET                   ; Return to the main Program
```

Advantage: **Simple** to use
Drawback: **Can not be used** if there are **multiple parameters** as there are very **few registers**.

2. USING MEMORY LOCATIONS DIRECTLY

Here, the main program **stores the parameter into a memory location** like [4000H], and Calls the Subroutine.
Now **Subroutine takes the parameter value form memory location** [4000H] and works on it.

```
Main:
MOV    [4000H], 25H;  (parameter value 25H stored at location 4000H)
Call   Sub
...

Sub:
MOV    AL, [4000H]  ; Subroutine takes parameter value from location [4000H]
...
RET                 ; Return to the main Program
```

Advantage: Can pass **many parameters** as there is **abundant memory.**
Drawback: Uses a **fixed memory location hence rigid**.

Scanned with CamScanner

3. **<u>USING MEMORY LOCATIONS INDIRECTLY</u>**

Here, the main program **stores the parameter into a memory location pointed indirectly by a register** like SI. The interesting point to note is that the **location can be any location** chosen by the programmer instead of being pre-determined by the subroutine.
The Subroutine will **take the parameter value from the memory location pointed by SI**.

```
Main:
MOV   SI, 4000H     ; We are choosing location 4000H to pass the parameter.
                    ; Could have been any other location as well.
MOV   [SI], 25H     ; (parameter value 25H stored at location pointed by SI)
Call  Sub
...

Sub:
MOV   AL, [SI]      ; Subroutine takes parameter value from any location pointed by SI
...
RET                 ; Return to the main Program
```

Advantage: **Can pass many parameters**. Moreover, the location can be chosen by the person calling the subroutine, instead of being pre-decided by the subroutine.
Hence **more flexible** than direct addressing.
Drawback: **More complex** than direct addressing.

4. **<u>USING STACK</u>**

Here, the main program **Pushes the parameter into the stack** and Calls the subroutine.
The interesting point to note is, **during "CALL", microprocessor pushes the return address into the stack**. This will be placed above the parameter, which we stored in the Stack.
Hence, the **subroutine will have to first POP the return address** into some register.
Thereafter the subroutine will POP the parameter and use it.
Before returning, the **subroutine must first Push the return address into the stack** and only then execute the RET instruction. (Don't worry, its much simpler than it sounds ☺☺☺)

```
Main:
MOV   BX, 1234H     ; Put Parameter 1234H into BX.
PUSH  BX            ; (parameter value 1234H Pushed into top of stack)
Call  Sub           ; Now microprocessor pushes return address above the parameter into the stack
...

Sub:
POP   CX            ; Pop Return address into CX
POP   AX            ; Pop parameter into AX and use it
...
PUSH  CX            ; Push back return address into stack
RET                 ; Return to main program
```

Advantage: Can **pass many parameters** as stack can be very large.
Drawback: **Most Complex method**.

Scanned with CamScanner

8086 INTERRUPTS

- An interrupt is a special condition that arises during the working of a μP.
- The μP services it by executing a subroutine called Interrupt Service Routine (ISR).
- There are 3 sources of interrupts for 8086:

External Signal (Hardware Interrupts):

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

Special instructions (Software Interrupts):

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

Condition Produced by the Program (Internally Generated Interrupts):

8086 is interrupted when some special conditions occur while executing certain instructions in the program.
Eg: **An error in division** automatically causes the INT 0 interrupt.

INTERRUPT VECTOR TABLE (IVT) {10M --- IMPORTANT }

The **IVT contains ISR address** for the 256 interrupts.
Each ISR **address** is stored as **CS and IP**.
As each ISR address is of 4 bytes (2–CS and 2-IP), each ISR address requires 4 locations to be stored.
There are **256 interrupts**: INT 0 ... INT 255 ∴ the **total size of** the **IVT** is 256 x 4 **= 1KB**.
The first 1KB of memory, address 00000 H ... 003FF H, are reserved for the IVT.
Whenever an interrupt **INT N occurs, μP does N x 4 to get** values of **IP and CS from the IVT and hence perform the ISR**.

Scanned with CamScanner

1 KB (256 * 4)

Address	Content	Interrupt
00000 H	IP Lower	INT 0 --- Divide error
00001 H	IP Higher	
00002 H	CS Lower	
00003 H	CS Higher	
00004 H . . . 00007 H		INT 1 --- Single Stepping
00008 H . . . 0000B H		INT 2 --- NMI
0000C H . . . 0000F H		INT 3 --- Breakpoint
00010 H . . . 00013 H		INT 4 --- Interrupt on Overflow
00014 H . . . 0007F H		INT 5 . . . INT 31 --- Reserved
00080 H . . . 003FF H		INT 32 . . . INT 255 --- User Defined

Dedicated Interrupts (INT 0 – INT 4)

Scanned with CamScanner

DEDICATED INTERRUPTS (INT 0 ... INT 4)

1) **INT 0** (Divide Error)
This interrupt occurs whenever there is **division error**
i.e. when the result of a division is too large to be stored.
This condition normally occurs when the divisor is very small as compared to the dividend or the divisor is zero. #Refer example from Bharat Sir's lecture notes...
Its ISR address is stored at location 0 x 4 = 00000H in the IVT.

2) **INT 1** (Single Step)
The µP executes this interrupt **after every instruction if the TF is set**.
It puts µP in **Single Stepping** Mode i.e. the µP pauses after executing every instruction.
This is very useful during **debugging**. #Refer example from Bharat Sir's lecture notes...
Its ISR generally displays contents of all registers.
Its ISR address is stored at location 1 x 4 = 00004H in the IVT.

3) **INT 2** (Non Maskable Interrupt)
The µP executes this ISR in **response to** an interrupt on the **NMI** line.
Its ISR address is stored at location 2 x 4 = 00008H in the IVT.

4) **INT 3** (Breakpoint Interrupt)
This interrupt is used to cause **Breakpoints** in the program.
It is caused by writing the instruction INT 03H or simply INT.
It is useful in **debugging large programs** where Single Stepping is inefficient.
Its ISR is used to **display** the **contents of all registers** on the screen.
Its ISR address is stored at location 3 x 4 = 0000CH in the IVT.

5) **INT 4** (Overflow Interrupt)
This interrupt occurs if the **Overflow Flag is set AND** the µP executes the **INTO** instruction (Interrupt on overflow). #Show example from Bharat Sir's lecture notes...
It is used to detect overflow error in **signed arithmetic** operations.
Its ISR address is stored at location 4 x 4 = 00010H in the IVT.

Please Note: INT 0 ... INT 4 are called as dedicated interrupts as these interrupts are dedicated for the above-mentioned special conditions.

Scanned with CamScanner

Reserved Interrupts

INT 5 … INT 31

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

User defined Interrupts

INT 32 … INT 255

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

Hardware Interrupts

1) NMI (Non Maskable Interrupt)

This is a **non-maskable**, **edge** triggered, **high priority** interrupt.
On receiving an interrupt on NMI line, the µP executes **INT 2.**
µP obtains the ISR address from location 2 x 4 = 00008H from the IVT.
It reads 4 locations starting from this address to get the values for IP and CS, to execute the ISR.

☺ For doubts contact Bharat Sir on 98204 08217

2) INTR

This is a **maskable**, **level** triggered, **low priority** interrupt.

On receiving an interrupt on INTR line, the µP executes **2** $\overline{\textbf{INTA}}$ pulses.

1st $\overline{\textbf{INTA}}$ pulse --- the interrupting device **calculates** (prepares to send) the **vector number**.

2nd $\overline{\textbf{INTA}}$ pulse --- the interrupting device **sends** the **vector number "N"** to the µP.

Now µP multiplies N x 4 and goes to the corresponding location in the IVT to obtain the ISR address.
INTR is a maskable interrupt.
It is masked by making IF = 0 by software through **CLI** instruction.
It is unmasked by making IF = 1 by software through **STI** instruction.

Scanned with CamScanner

Response to any interrupt --- INT N

i) The µP will **PUSH Flag** register into the Stack.
SS:[SP−1], SS:[SP-2] ← Flag
SP ← SP − 2

ii) **Clear IF and TF** in the Flag register and thus disables INTR interrupt.
IF ← 0, TF ← 0

iii) **PUSH CS** into the Stack.
SS:[SP−1], SS:[SP-2] ← CS
SP ← SP − 2

iv) **PUSH IP** into the Stack.
SS:[SP−1], SS:[SP-2] ← IP
SP ← SP − 2

v) **Load new IP** from the IVT
IP ← [N x 4], [N x 4 + 1]

vi) **Load new CS** from the IVT
IP ← [N x 4 + 2], [N x 4 + 3]

Since CS and IP get new values, control shifts to the address of the ISR and the ISR thus begins. At the end of the ISR the µP encounters the IRET instruction and returns to the main program in the following steps.

Response to IRET instruction

i) The µP will **restore IP from the stack**
IP ← SS:[SP], SS:[SP+1]
SP ← SP + 2

ii) The µP will **restore CS from the stack**
CS ← SS:[SP], SS:[SP+1]
SP ← SP + 2

iii) The µP will **restore FLAG register from the stack**
Flag ← SS:[SP], SS:[SP+1]
SP ← SP + 2

Scanned with CamScanner

Interrupt Priorities

Interrupt	Priority	
	(Simultaneous occurrence)	(To interrupt another ISR)
Divide Error, INT n, INTO	1 (**Highest**)	Can interrupt any ISR
NMI	2	
INTR	3	Cannot interrupt an ISR (IF, TF ← 0)
Single Stepping	4(**Lowest**)	

Priority in 8086 interrupts is of two types:

1. Simultaneous Occurrence:

When more than one interrupts occur simultaneously then, **all s/w interrupts except single stepping**, get the **highest priority**.
This is followed by **NMI**. Next is **INTR**. Finally, the **lowest priority** is of the **single stepping** interrupt.

Eg: Assume the μP is executing a **DIV** instruction that causes a **division error** and **simultaneously INTR** occurs.
Here **INT 0** (Division error) will be **serviced first** i.e. its ISR will be executed, as it has higher priority, and **then INTR** will be **serviced.** #Please refer Bharat Sir's Lecture Notes for this ...

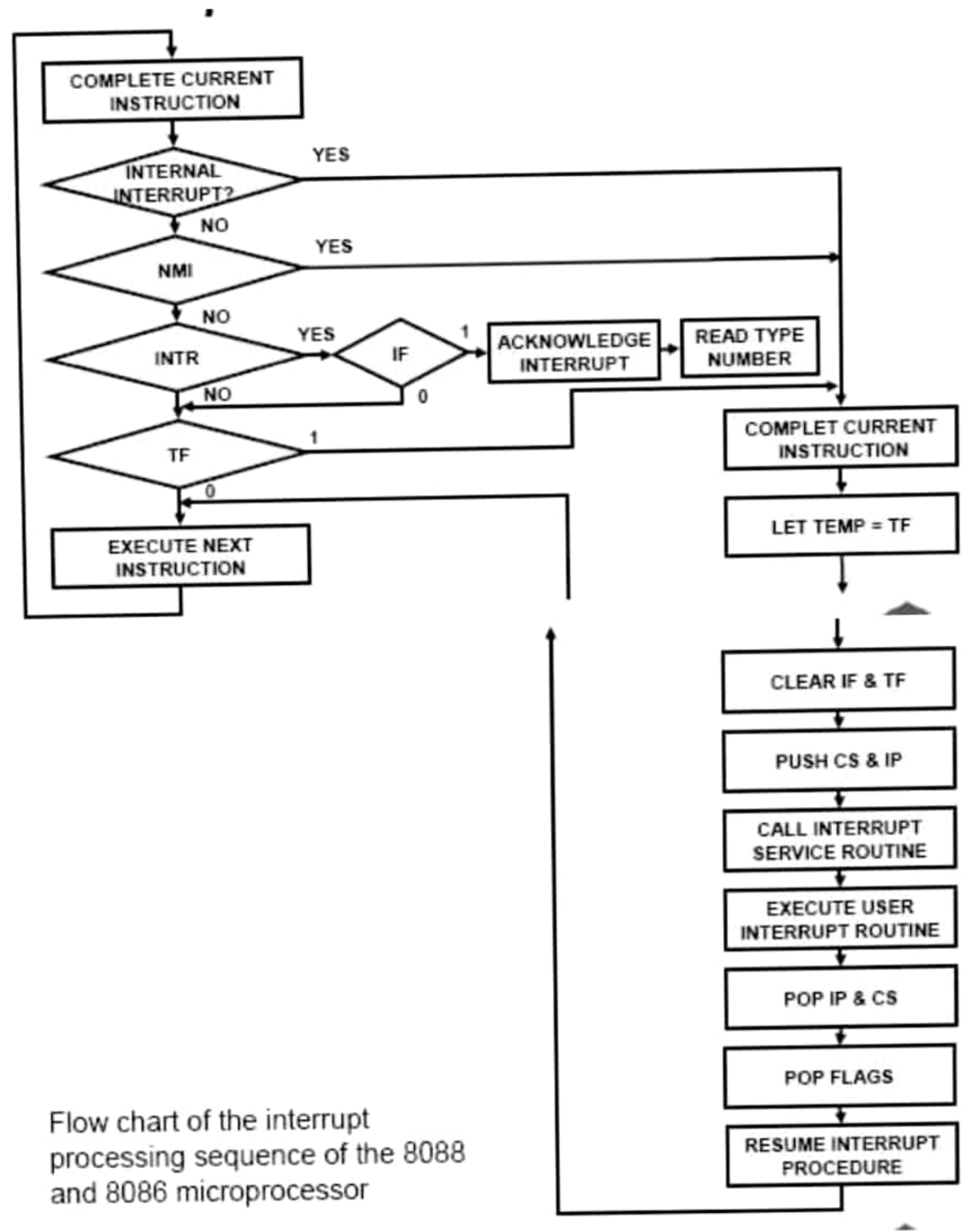
2. Ability to interrupt another ISR:

Since software interrupts (**INT N**) are **non-maskable**, they **can interrupt** any ISR.
NMI is also **non-maskable** hence it **can** also **interrupt** any ISR.
But **INTR** and **Single stepping cannot interrupt** another ISR **as** both are **disabled** before μP enters an ISR by **IF ← 0 and TF ← 0**.

Eg: Assume the μP executes DIV instruction that causes a **division error**. So μP gets the **INT 0** interrupt and now **μP enters the ISR for INT 0**. **During** the execution of **this ISR, NMI and INTR occur**.
Here **μP will branch out** from the ISR of INT 0 and **service NMI** (as **NMI is non-maskable**).
After completing the ISR of NMI **μP** will **return to** the **ISR for INT 0**.
INTR is **still pending but** the **μP will not service INTR** during the ISR of INT 0 (as **IF ← 0**).
μP will first **finish** the **INT 0 ISR** and **only then service INTR**.
Thus INTR and Single stepping cannot interrupt an existing ISR.

Scanned with CamScanner

Interrupt priority Flowchart {Optional – Only for reference}



Flow chart of the interrupt processing sequence of the 8088 and 8086 microprocessor

Scanned with CamScanner

8086 CONFIGURATIONS

Some common devices used in 8086 circuits for Minimum Mode or Maximum Mode are:

8282 – 8-bit (Octal) Latch

1) 8282 is an **8-bit latch.**
2) In 8086, the **address bus** is **multiplexed with** the **data bus** and status bits.
3) 8282 is **used to latch the address** from this bus.
4) The ALE signal is connected to STB of 8282.
5) When **STB (ALE) is high**, the **input** is latched and **transferred to** the **output**.
   Hence address is latched.
6) When **STB (ALE) is low**, the **input is discarded.**
   Hence, data is not latched. The previously latched address remains at the output.
7) As totally 21 bits are to be latched (A_{19}-A_0 and $\overline{\textbf{BHE}}$), **3 latches are required**, each latch being 8-bit.

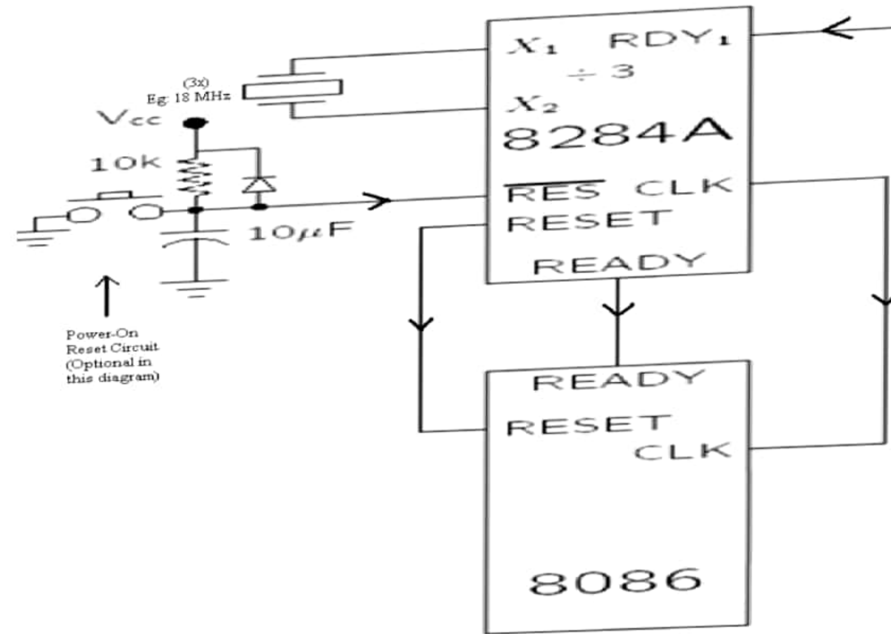
8286 – 8-bit Data Trans-receiver

1) 8286 is an **8-bit Trans-receiver.**
2) It acts as a **bi-directional buffer,** and **increases** the **driving capacity** of the data bus.
3) It is **enabled** when $\overline{\textbf{OE}}$ **is low**.
4) **T controls** the **direction** of data.
   If **T = 1**: **data** is **transmitted**.
   If T = 0: data is received.
5) As the **data bus** is **16-bits**, **2** trans-receivers are **required**.
6) Its main function is to **prevent address and allow data** to be transferred on the data bus.
7) In the 1st T-State when the bus contains address, $\overline{\textbf{OE}}$ is high hence the transreceiver is disabled.

   Thereafter when the bus contains data $\overline{\textbf{OE}}$ is low and the transreceiver is enabled. Thus it **only allows data to pass**.

Scanned with CamScanner

8284 – Clock Generator



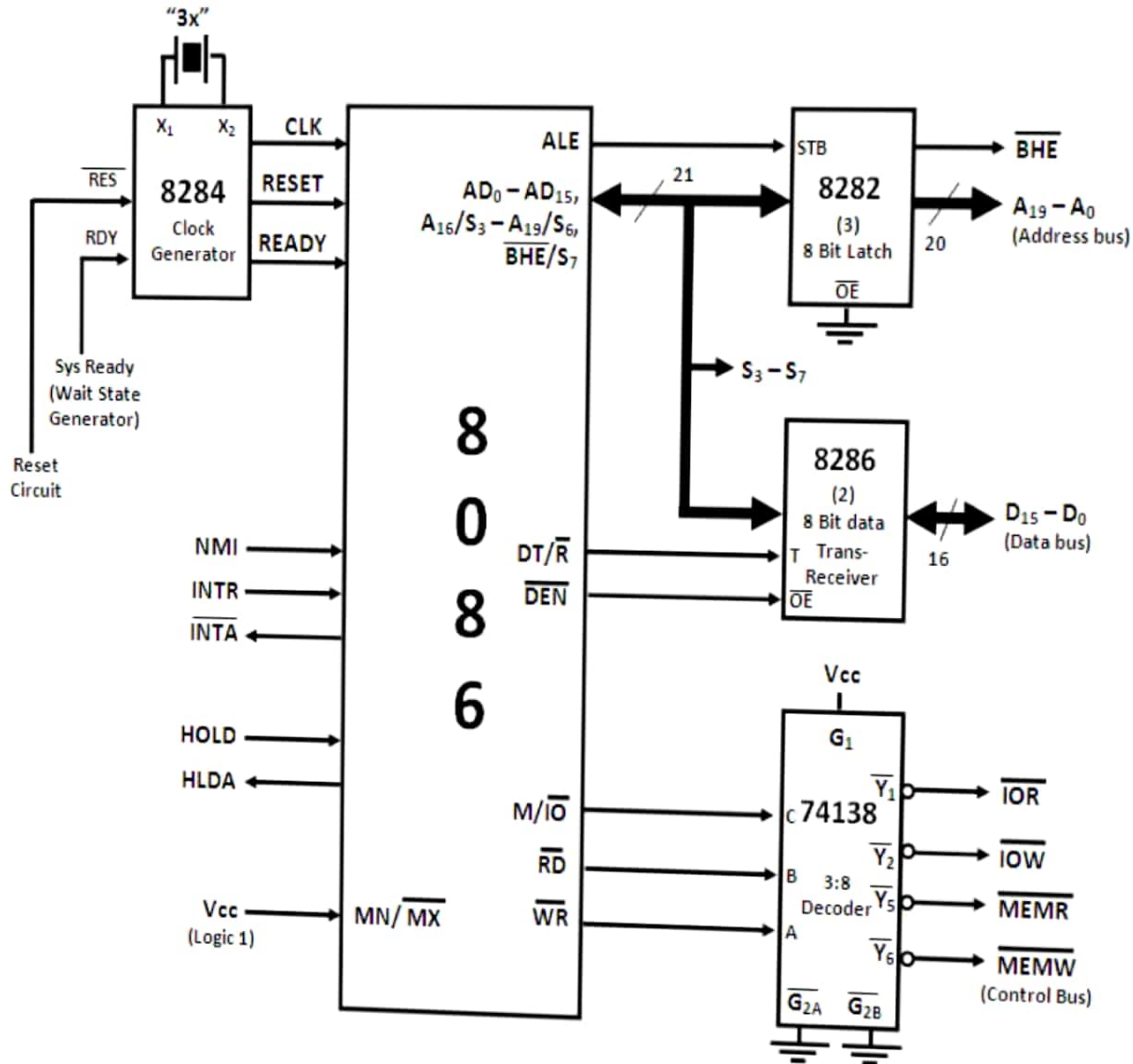
1) 8284 is a *Clock Generator* IC.
2) It provides the C **LOCK** (CLK) signal, a train of pulses at a constant freq, to the entire circuit.
3) It synchronizes the READY (**RD** Y) signal which indicates that an interface is ready for data.
4) It also synchronizes the RESET (RST) signal which is used to initialize the system.
5) There are 2 ways of providing the frequency input to 8284.
   1) Through EFI (External Frequency Input)
   A "**Pulse Generator**" circuit can be connected to the EFI pin, to provide an external freq.
   2) Through X1, X2 (Oscillator Clock Inputs)
   An **Oscillator** can be connected across the X1, X2 lines to provide constant clock signal.
6) In both the cases the Output Clock frequency = 1/3rd of the Input Clock frequency **to produce a 33% duty cycle** required by the Microprocessor. **For doubts contact Bharat Sir on 98204 08217**
7) Clock Selection is done by the **F/ $\overline{C}$** pin.

F/ $\overline{C}$ = 1 → Input Clock given **through EFI** pin.

F/ $\overline{C}$ = 0 → Input Clock given **through** Oscillator inputs **X1, X2** pins.

Scanned with CamScanner

8086 Minimum Mode Configuration



Scanned with CamScanner

1) **8086 works in Minimum Mode, when MN/ $\overline{\text{MX}}$ = 1.**

2) **In Minimum Mode, 8086 is the ONLY processor in the system.**
The Minimum Mode circuit of 8086 is as shown above.

3) Clock is provided by the 8284 Clock Generator.

4) Address from the address bus is latched into 8282 8-bit latch.
Three such latches are needed, as address bus is 20-bit.
The ALE of 8086 is connected to STB of the latch.
The ALE for this latch is given by 8086 itself. #Please refer Bharat Sir's Lecture Notes for this ...

5) The data bus is driven through 8286 8-bit transreceiver.
Two such transreceivers are needed, as the data bus is 16-bit.
The transreceivers are enabled through the **$\overline{\text{DEN}}$** signal, while the direction of data is controlled by the **DT/ $\overline{\text{R}}$** signal. **$\overline{\text{DEN}}$** is connected to **$\overline{\text{OE}}$** and **DT/ $\overline{\text{R}}$** is connected to T. **Both $\overline{\text{DEN}}$ and DT/ $\overline{\text{R}}$ are given by 8086 itself.**

$\overline{\text{DEN}}$	DT/ $\overline{\text{R}}$	Action
1	X	Transreceiver is disabled
0	0	Receive data
0	1	Transmit data

6) **Control signals for all operations are generated by decoding M/ $\overline{\text{IO}}$, $\overline{\text{RD}}$ and $\overline{\text{WR}}$ signals.**

? **For doubts contact Bharat Sir on 98204 08217**

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

7) **M/ $\overline{\text{IO}}$, $\overline{\text{RD}}$, $\overline{\text{WR}}$** are **decoded by a 3:8 decoder like IC 74138.**

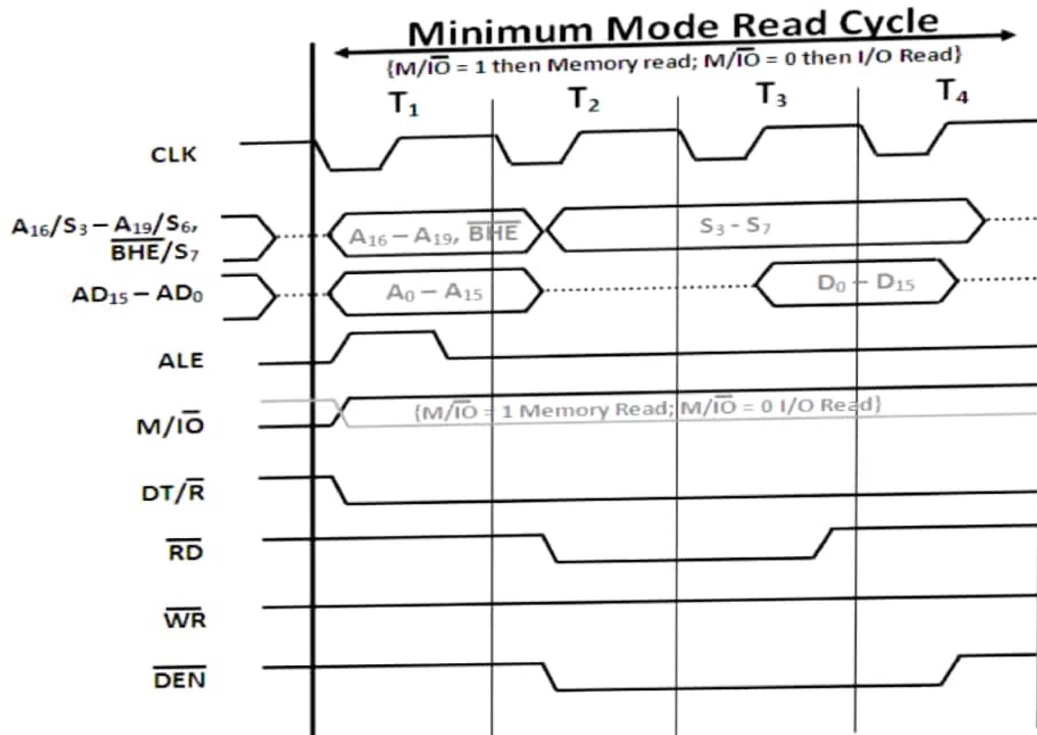
8) **Bus Request (DMA) is done using the HOLD and HLDA signals.**

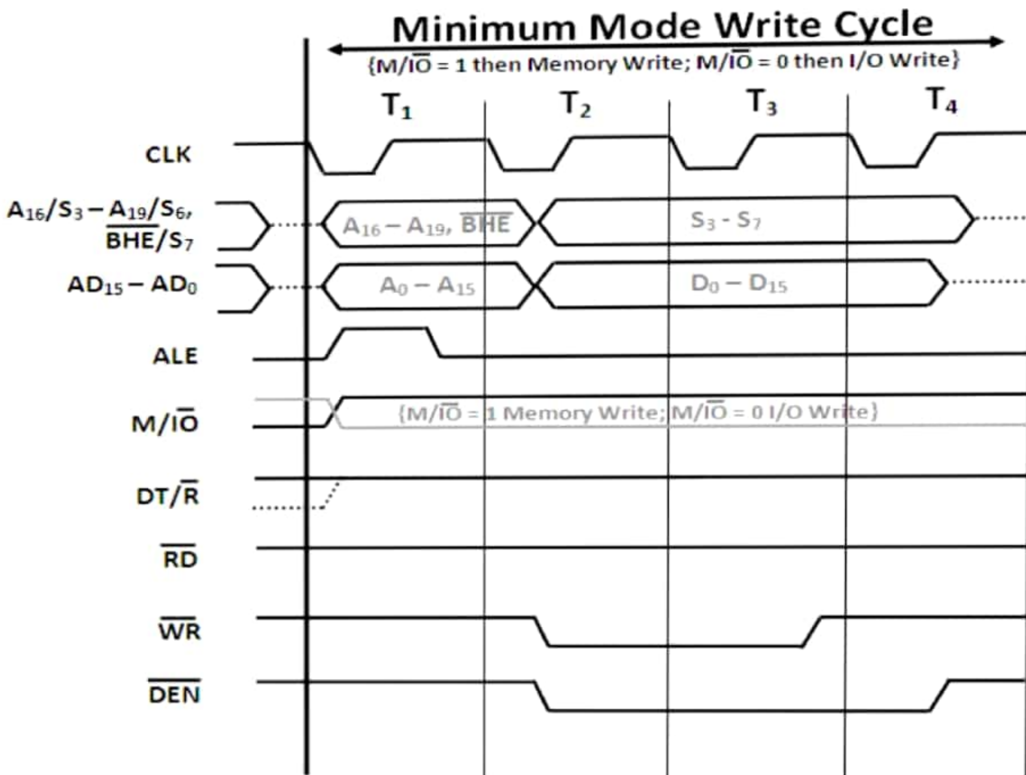
9) **$\overline{\text{INTA}}$ is given by 8086, in response to an interrupt on INTR line.**

10) **The Circuit is simpler than Maximum Mode but does not support multiprocessing.**

Scanned with CamScanner

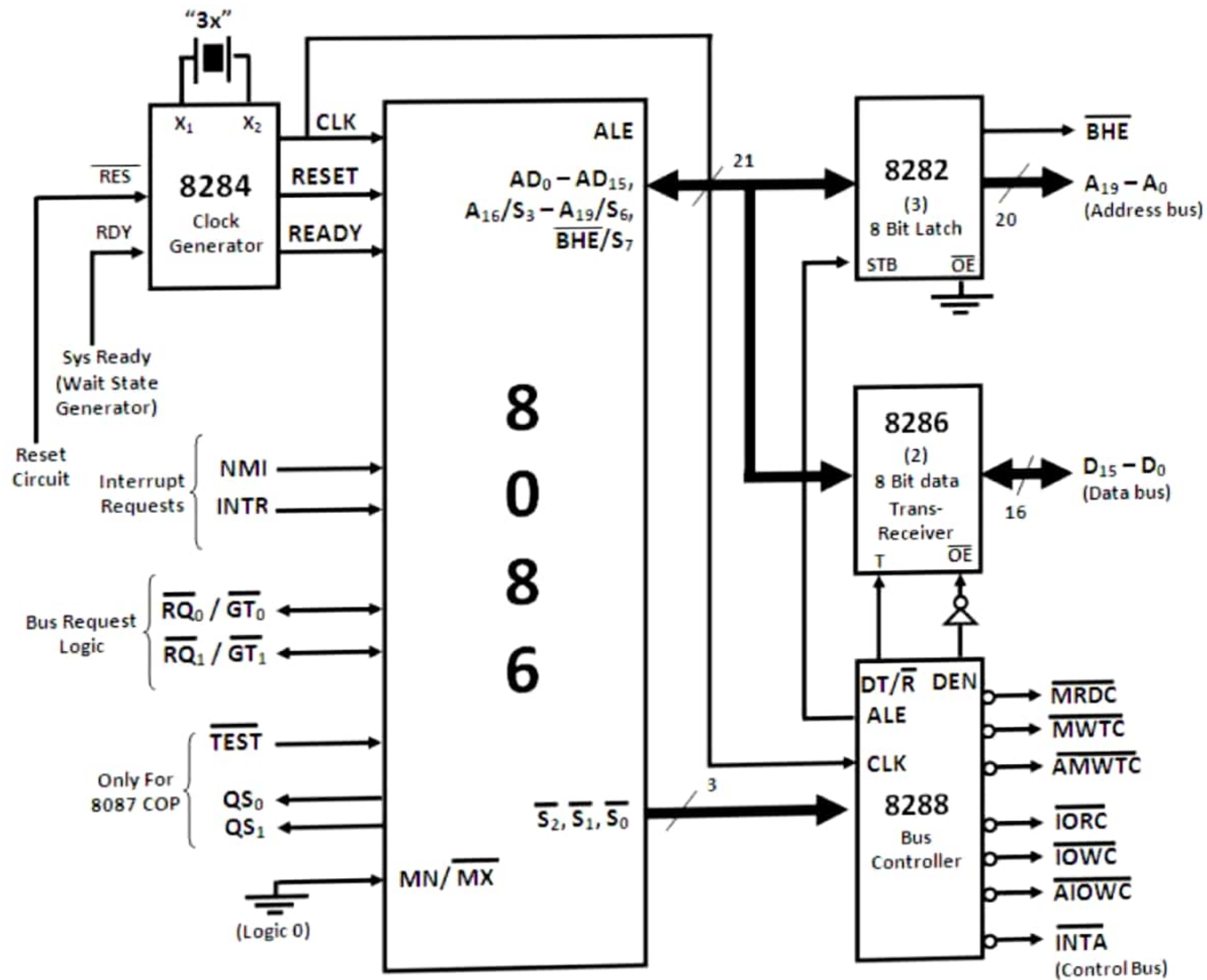
Timing Diagrams:





Scanned with CamScanner

8086 Maximum Mode Configuration



Scanned with CamScanner

1) **8086 works in Maximum Mode, when MN/ $\overline{MX}$ = 0.**

2) **In Maximum Mode, we can connect more processors to 8086 (8087/8089).**
The Maximum Mode circuit of 8086 is as shown above.

3) Clock is provided by the 8284 Clock Generator.

4) The most significant part of the Maximum Mode circuit is the 8288 Bus Controller.
Instead of 8086, the Bus Controller provides the various control signals as explained below.

5) Address form the address bus is latched into 8282 8-bit latch.
Three such latches are needed, as address bus is 20-bit.
This ALE is connected to STB of the latch.
The ALE for this latch is given by 8288 Bus Controller.

6) The data bus is driven through 8286 8-bit transreceiver.
Two such transreceivers are needed, as the data bus is 16-bit.
The transreceivers are enabled through the DEN signal, while the direction of data is controlled by the **DT/ $\overline{R}$** signal.

DEN is connected to $\overline{OE}$ and **DT/ $\overline{R}$** is connected to T.

Both DEN and DT/ $\overline{R}$ are given by 8288 Bus Controller.

DEN (Of 8288)	**DT/ $\overline{R}$**	**Action**
0	X	Transreceiver is disabled
1	0	Receive data
1	1	Transmit data

7) **Control signals for all operations are generated by decoding $\overline{S_2}$, $\overline{S_1}$ and $\overline{S_0}$ signals.** For doubts contact Bharat Sir on 98204 08217

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	**Processor State (What the µP wants to do)**	**8288 Active Output (What Control signal should 8288 generate)**
0	0	0	Int. Acknowledge	$\overline{\textbf{INTA}}$
0	0	1	Read I/O Port	$\overline{\textbf{IORC}}$
0	1	0	Write I/O Port	$\overline{\textbf{IOWC}}$ and $\overline{\textbf{AIOWC}}$
0	1	1	Halt	None
1	0	0	Instruction Fetch	$\overline{\textbf{MRDC}}$
1	0	1	Memory Read	$\overline{\textbf{MRDC}}$
1	1	0	Memory Write	$\overline{\textbf{MWTC}}$ and $\overline{\textbf{AMWTC}}$
1	1	1	Inactive	None

Scanned with CamScanner

8) **$\overline{S_2}$, $\overline{S_1}$ and $\overline{S_0}$ are decoded using 8288 bus controller.**

9) **Bus request is done using $\overline{RQ}$ / $\overline{GT}$ lines interfaced with 8086.**

RQ_0/GT_0 has higher priority than RQ_1/GT_1. ☺ For doubts contact Bharat Sir on 98204 08217

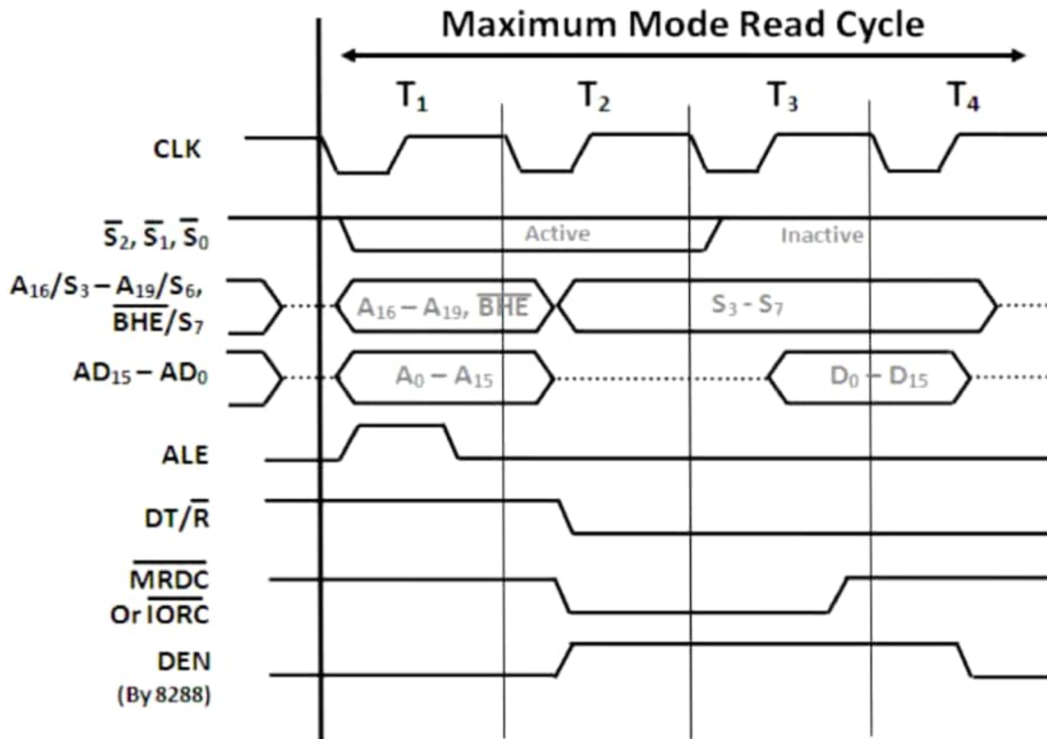
10) **$\overline{INTA}$ is given by 8288 Bus Controller, in response to an int. on INTR line of 8086.**

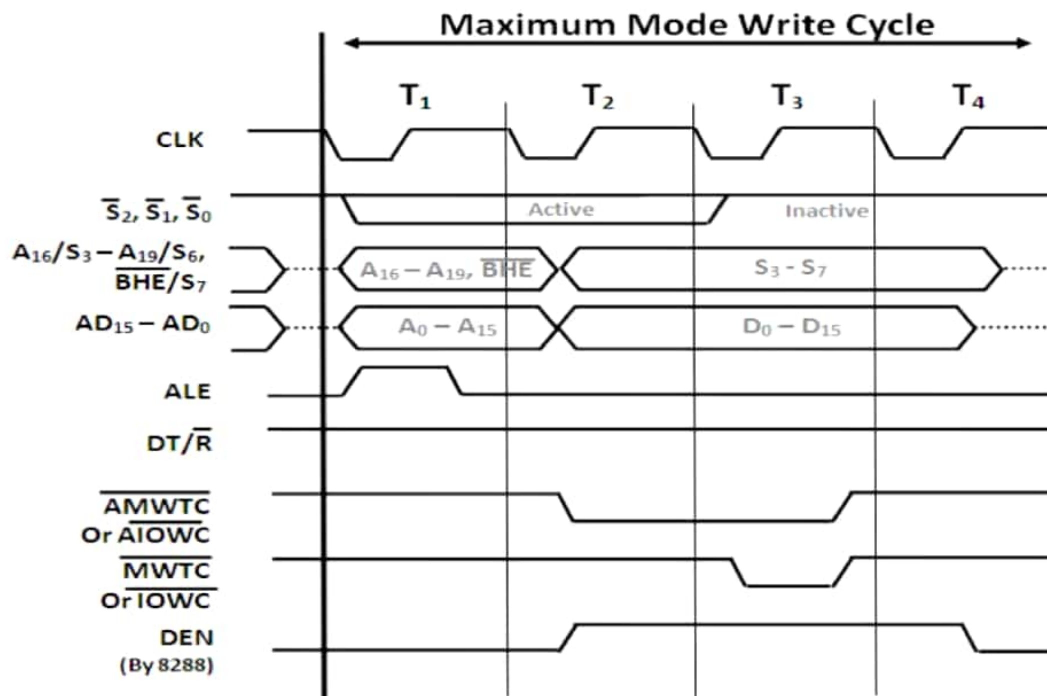
11) **Max mode circuit is more complex than Min mode but supports multiprocessing hence gives better performance.**

12) In max mode, the advanced write signals get activated one T-State in advance as compared to normal write signals. This gives slower devices more time to get ready to accept the data (as µP is writing), and hence reduces the number of "wait states".

Scanned with CamScanner

TIMING DIAGRAMS





Scanned with CamScanner

Differentiate between

	MIN MODE	MAX MODE
1	It is a **uniprocessor mode**. 8086 is the only processor in the circuit.	It is a **multiprocessor mode**. Along with 8086, there can be other processors like 8087 and 8089 in the circuit.
2	Here **MN/ $\overline{MX}$** is connected to **Vcc.**	Here **MN/ $\overline{MX}$** is connected to **Ground.**
3	**ALE** for the latch is **given by 8086** itself.	As there are multiple processors, **ALE** for the latch is **given by 8288 bus controller**.
4	**$\overline{DEN}$** and **DT/ $\overline{R}$** for the transreceivers **are given by 8086 itself.**	As there are multiple processors, **DEN** and **DT/ $\overline{R}$** for the transreceivers is **given by 8288 bus controller**.
5	Direct control signals like **M/ $\overline{IO}$, $\overline{RD}$ and $\overline{WR}$ are produced by 8086 itself.**	Instead of control signals, all processors produce status signals **$\overline{S_2}$, $\overline{S_1}$ and $\overline{S_0}$**
6	Control signals **M/ $\overline{IO}$, $\overline{RD}$ and $\overline{WR}$ are decocded by a 3:8 decoder IC 74138.**	Status signals **$\overline{S_2}$, $\overline{S_1}$ and $\overline{S_0}$ require special decoding** are decoded by **8288 bus controller**.
7	**$\overline{INTA}$** for interrupt acknowledgement **is produced by 8086.**	**$\overline{INTA}$** for interrupt acknowledgement **is produced by 8288 Bus Controller.**
8	**Bus request are grant** is handled using **HOLD and HLDA** signals.	**Bus request are grant** is handled using **$\overline{RQ}$ / $\overline{GT}$** signals.
9	Since **74138** does not independently generate any signals, it **does not need a CLK**.	Since **8288** independently generates control signals, it **needs a CLK from 8284** clock generator.
10	The circuit is **simpler but does not support multiprocessing**.	The circuit is **more complex but supports multiprocessing**.

Scanned with CamScanner

Differentiate between

	8085	8086
1	**8-bit processor** with: **8-bit ALU** and **8-bit data bus**.	**16-bit processor** with: **16-bit ALU** and **16-bit data bus**.
2	Memory **banking not needed**.	**Memory** is divided into **two banks.**
3	**16-bit address bus.**	**20-bit address bus.**
4	Accesses **64 KB Memory.**	Accesses **1 MB Memory.**
5	**Segmentation not performed**.	**Segmentation is performed.**
6	Has **5 status flags**.	Has **6 status flags** and **3 control flags.**
7	**Pipelining is not performed.**	**2 stage Pipelining is performed.**
8	**Has 5 hardware interrupts.**	**Has 2 hardware interrupts.**
9	**Does not support multiprocessing.**	**Supports multiprocessing** in Max Mode.
10	**ALU cannot perform** powerful arithmetic like **MUL and DIV**.	**ALU can perform** powerful arithmetic like **MUL and DIV**.

Scanned with CamScanner

Differentiate between

	8088	8086
1	**16-bit processor** with: **16-bit ALU** and **8-bit data bus**.	**16-bit processor** with: **16-bit ALU** and **16-bit data bus**.
2	Memory **banking not needed**. Hence **circuit is simpler**.	**Memory** is divided into **two banks.** Hence **circuit is more complex**.
3	Since **data bus is 8-bits**, it can transfer **1 byte in 1 cycle**. Hence is **slower**.	Since **data bus is 16-bits**, it can transfer **2 bytes in 1 cycle**. Hence is **faster**.
4	$\overline{\textbf{BHE}}$ is **not needed**. Instead, has a signal called SSO used for Single Stepping.	$\overline{\textbf{BHE}}$ **is needed** to enable the higher bank**.**
5	**Prefetch queue is of 4 bytes**.	**Prefetch queue is of 6 bytes.**
6	Uses **IO/** $\overline{\textbf{M}}$ compatible with 8085.	Uses **M/** $\overline{\textbf{IO}}$ to differentiate between memory and I/O operations.

Scanned with CamScanner