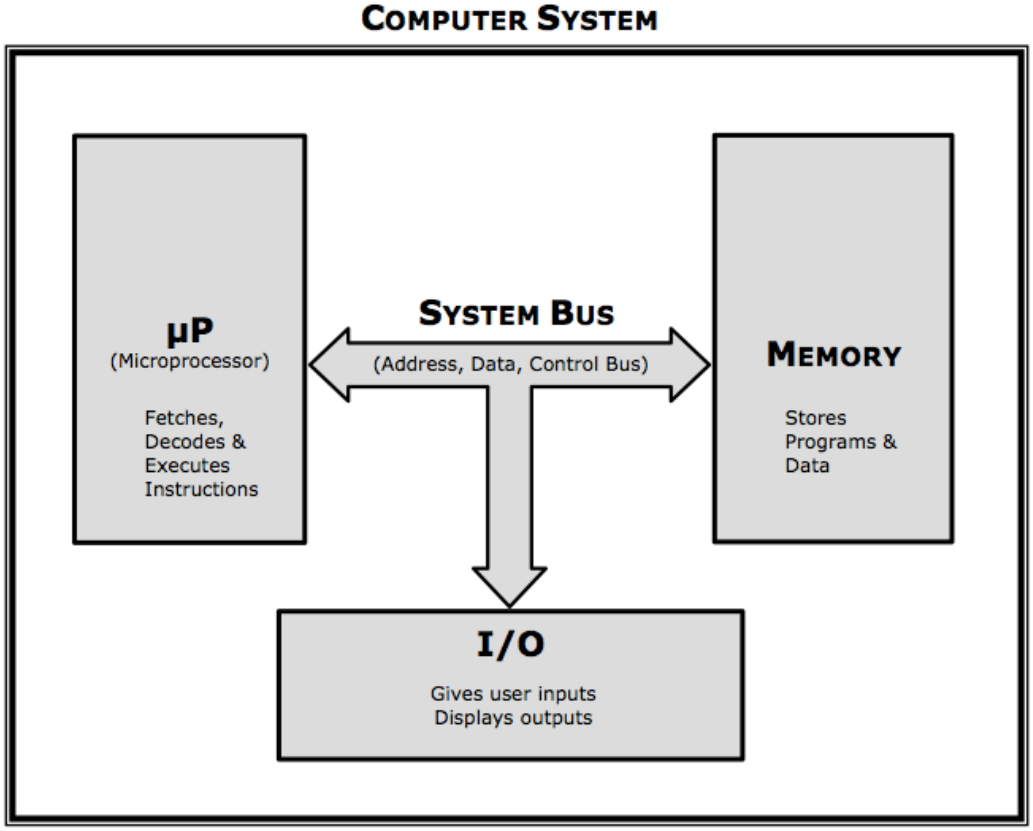
INTRODUCTION TO MICROPROCESSORS

WWW.BHARATACHARYAEDUCATION.COM

INTRODUCTION | BASIC ORGANIZATION OF A COMPUTER

A computer system, as we know it, consists of various components.
They can be broadly classified into three sections:
The Processor, Memory and I/O.



THE PROCESSOR – "µP"

The heart of the computer is its µP (Microprocessor).
Current generation computers use processors like Intel Core i3, i5 or i7 and so on. They have come a long way from the initial processors that you are about to learn E.g.: 8085, 8086 etc.

Back in the day (1940s), when micro-electronics was not invented, processors looked very different and were certainly not "micro" in appearance. They were created using huge arrays of physical switches which were operated manually and often occupied large rooms.

In the following decades, with the invention of micro-electronics, scientists managed to embed thousands of microscopic switches (transistors) inside a small chip, and called it a "**Micro-processor**".

Over the years, microprocessors grew in strength.
From housing a few thousand transistors (8085) to containing more than a billion transistors (Core i7), the computational power has been increasing exponentially. Having said that, some of the basics still remain the same.

To put it simply, **the main function of a µP is to Fetch, Decode and Execute instructions.**

Instructions are a part of programs. Programs are stored in the memory.
Firstly, µP fetches an instruction from the memory.
It then decodes the instruction. This means, it "understands" the binary pattern of the instruction, also called its opcode. Every instruction when stored in the memory is in its unique binary form, which indicates the operation to be performed. This is called its opcode.
Upon decoding the opcode, µP understands the operation to be performed and hence "executes" the instruction. This entire process is called an **"Instruction cycle"**.
Now the process is repeated for the next instruction.
Like this, one by one, all instructions of a program are executed.
Of course by advanced concepts like **pipelining**, **multitasking**, **multiprocessing** etc., this procedure has become very advanced and efficient today. You will get to learn all of them, in the due course of this ever intriguing subject.

We begin learning with basic processors like **8085** or **8086**, but make no mistake, non of this is "outdated". Yes, your mobile phone or your computer today uses the most advanced cutting edge processors (**A11 Bionic** et.al), but to run a **traffic light** or **TV remote** control you don't need a core i7 now, do you? And these are used by the millions across the world. They simply use processors of the same grade as an 8085 or an 8086, with different product numbers as they are made by various manufacturers.

MEMORY

Memory is used to store information.
It stores two kinds of information… programs and data.

For example:
MS Word is a program, and the word documents are its data.
Video player is a program, and the videos are its data.
WhatsApp is a program, and the messages are its data, and so on.
All programs and data are stored in the memory, in digitized form, where every information is represented in 1s and 0s called binary digits or simply bits.

There are various forms of memory devices.
The main memory also called primary memory consists of Ram and ROM.
Other memory devices like Hard disk, Floppy, CD/ DVD etc. are secondary storage devices.
Additionally there is also a high speed memory called Cache composed of SRAM.
For the majority portion of this book, you are dealing with the initial processors like 8086.
It will be in your best interest to think of Primary Memory only, whenever we speak of memory. That is because, secondary memory and high speed memories were implemented much later in the evolution of processors as the demand for mass storage and high speed performance started increasing. So, from now on in this book, unless specified otherwise, **the word memory refers to primary memory that is RAM and ROM.**

The memory is a series of locations.
Each location is identified by its own unique address.
Every location contains 1 Byte (8 bits) of data. There is a very good reason for this, and you will learn it when we discuss the topic of memory banking in 8086.

I/O DEVICES

I/O devices are used to enter programs and data as inputs and display or print the results as outputs. We are all familiar with devices such as the keyboard, mouse, printer, monitor etc. Every form of computer system has a set of I/O devices for human interaction. A device like a touch screen performs dual functions of both input and output.

The µP, Memory and I/O are all connected to each other using the System Bus.

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.

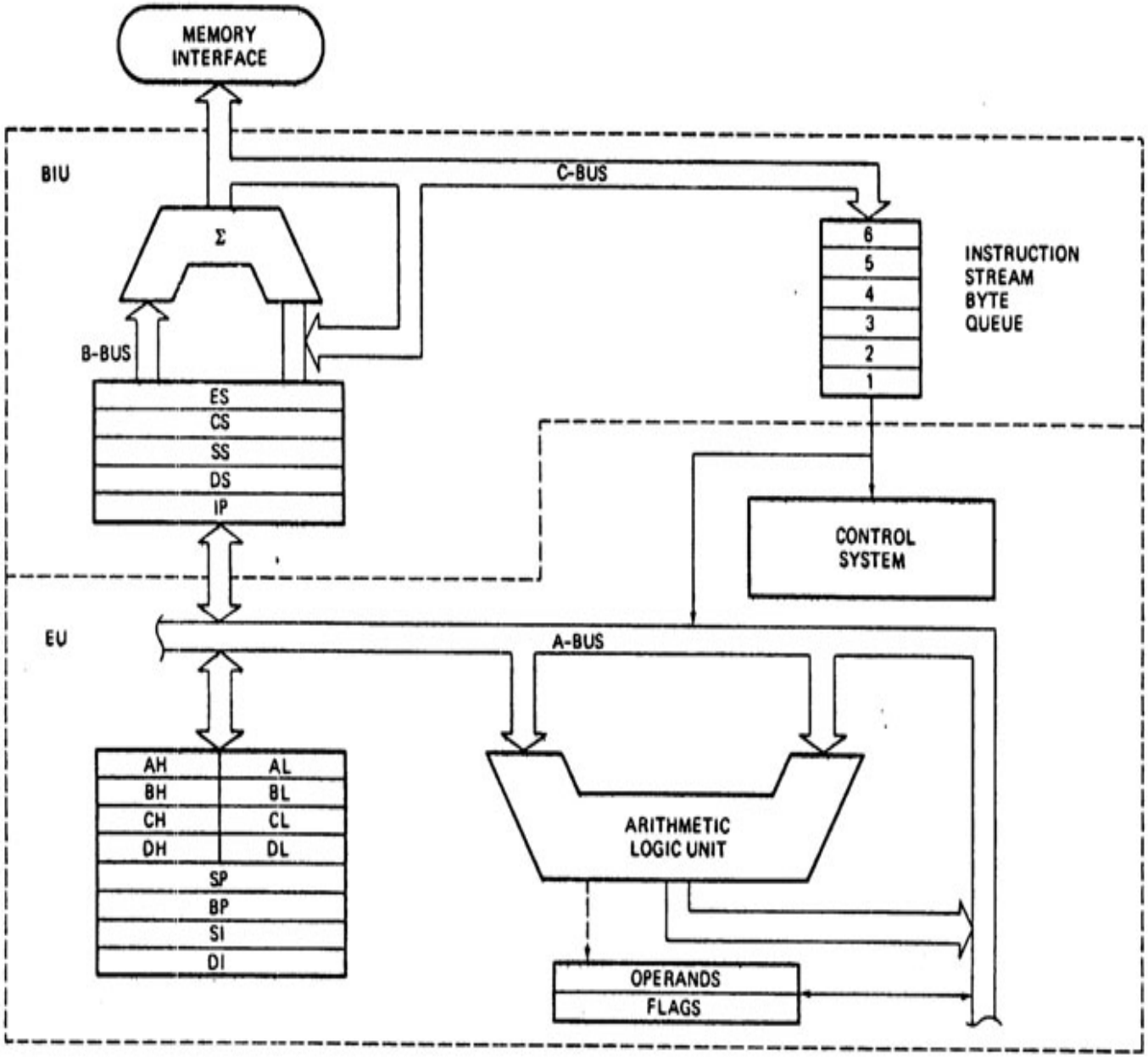
4) 8086 provides **Memory Banks**.
The entire memory of 1 MB is **divided into 2 banks of 512KB each**, in order to transfer 16-bits in 1 cycle. The banks are called **Lower Bank** (even) **and Higher Bank** (odd).

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).

ARCHITECTURE OF 8086



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**.

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:

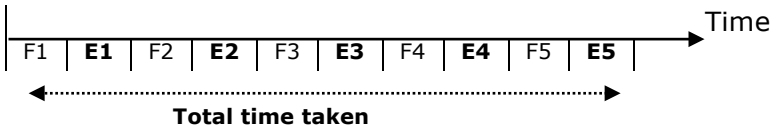
Physical address = Segment Address x 10h + Offset Address

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 0ffset 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.

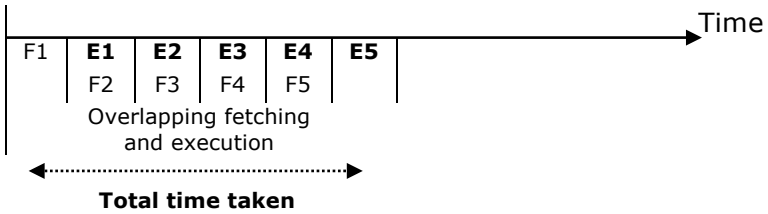
d) 6-Byte Pre-Fetch Queue {Pipelining – 4m}

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



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.**

*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.**

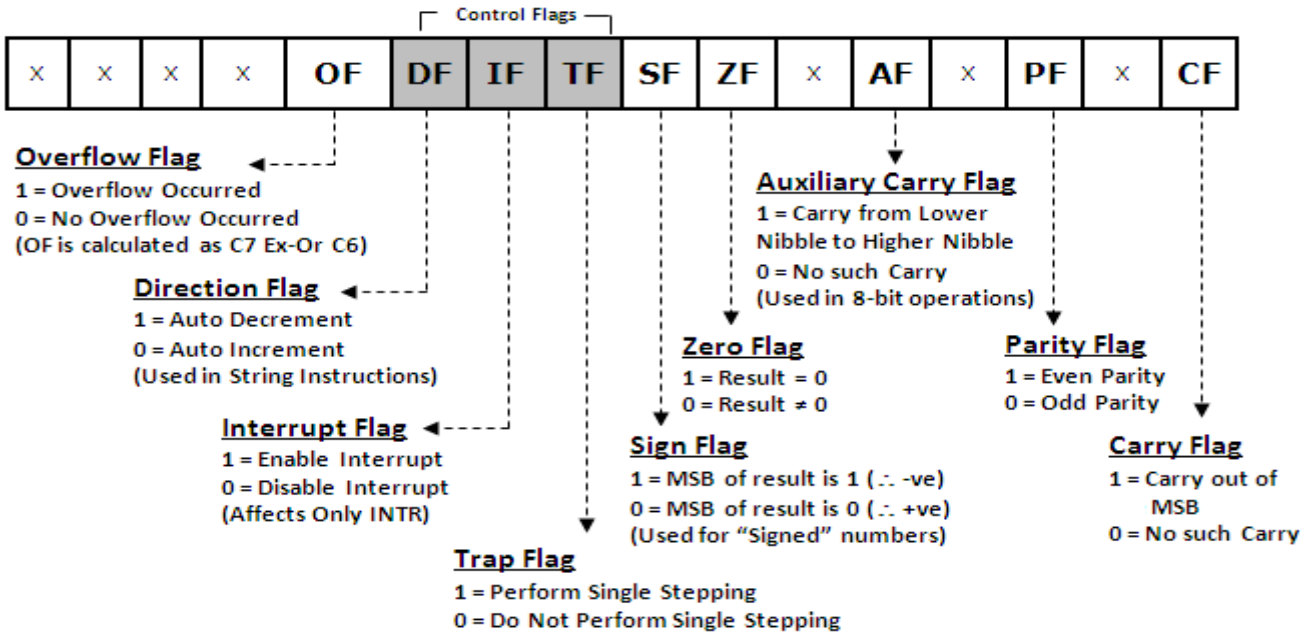
c) ALU (16-Bits)
It has a **16-bit ALU**. It performs 8 and 16-bit arithmetic and logic operations.

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.



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**.

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**.

1 MB

512 KB	**512 KB**

Odd Bank

- Also called as "Higher bank"
- Address range:

00001H
00003H
00005H
.
.
.
FFFFFH

- Selected when $\overline{\textbf{BHE}}$ = 0

Even Bank

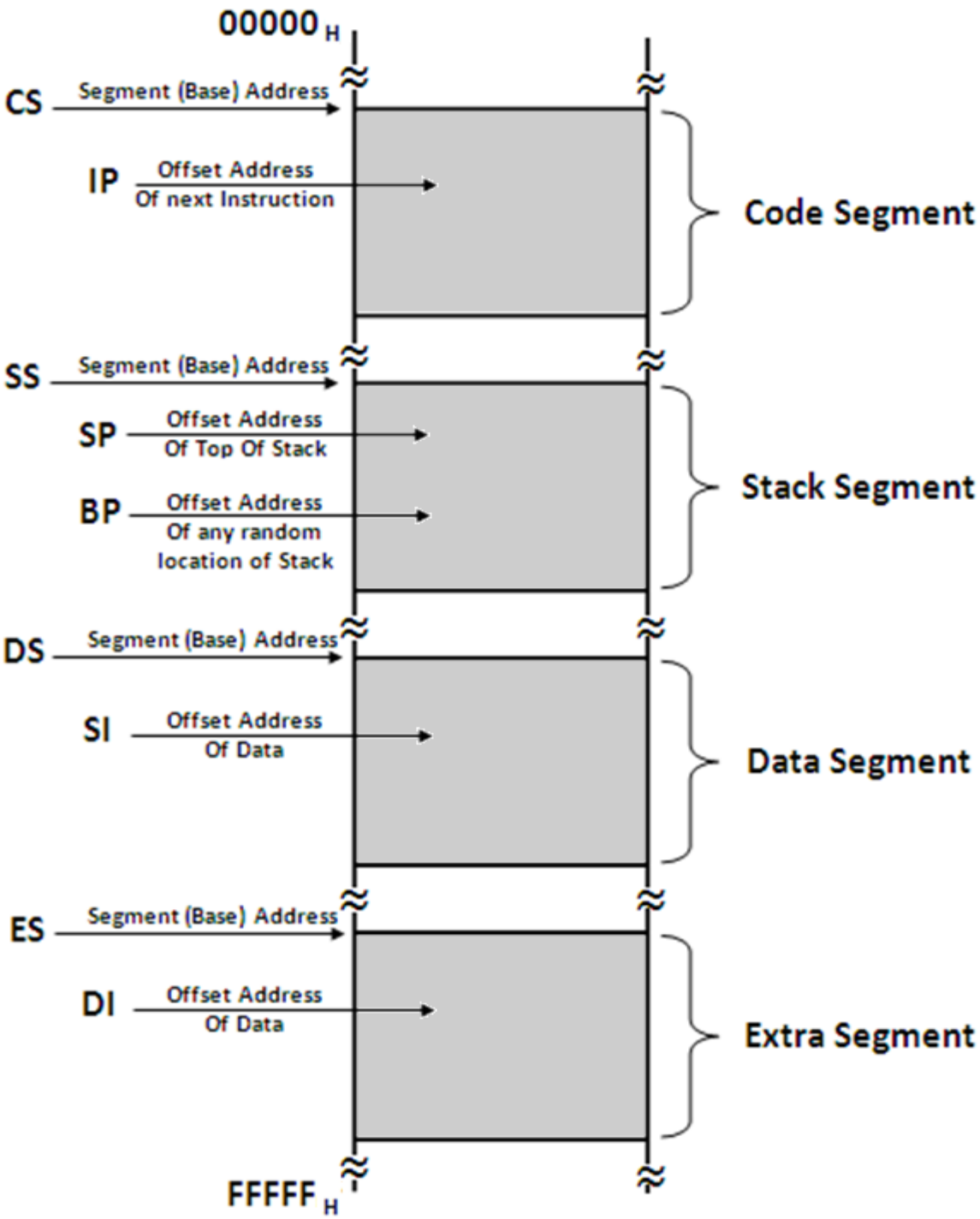
- Also called as "Lower bank"
- Address range:

00000H
00002H
00004H
.
.
.
FFFFEH

- Selected when A_0 = 0

$\overline{\textbf{BHE}}$	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).

MEMORY SEGMENTATION IN 8086



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).

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**.

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.

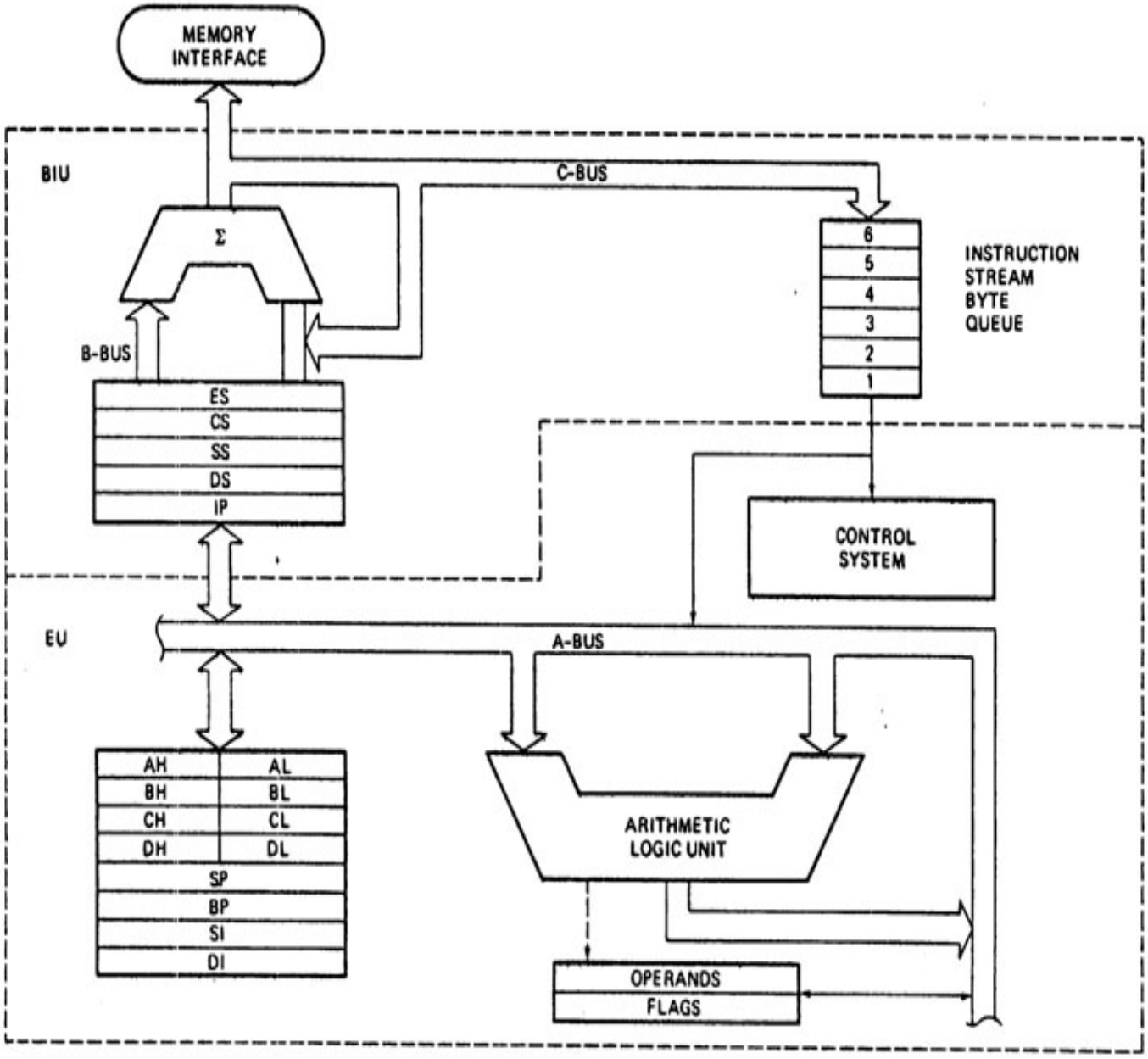
4) 8086 provides **Memory Banks**.
The entire memory of 1 MB is **divided into 2 banks of 512KB each**, in order to transfer 16-bits in 1 cycle. The banks are called **Lower Bank** (even) **and Higher Bank** (odd).

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).

ARCHITECTURE OF 8086



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**.

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:

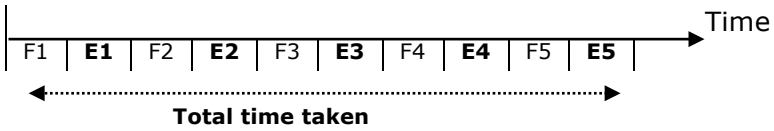
Physical address = Segment Address x 10h + Offset Address

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 0ffset 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.

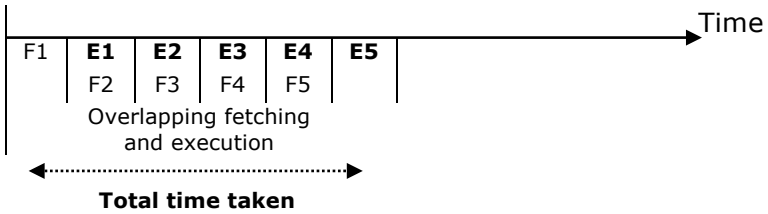
d) 6-Byte Pre-Fetch Queue {Pipelining – 4m}

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



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.**

*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.**

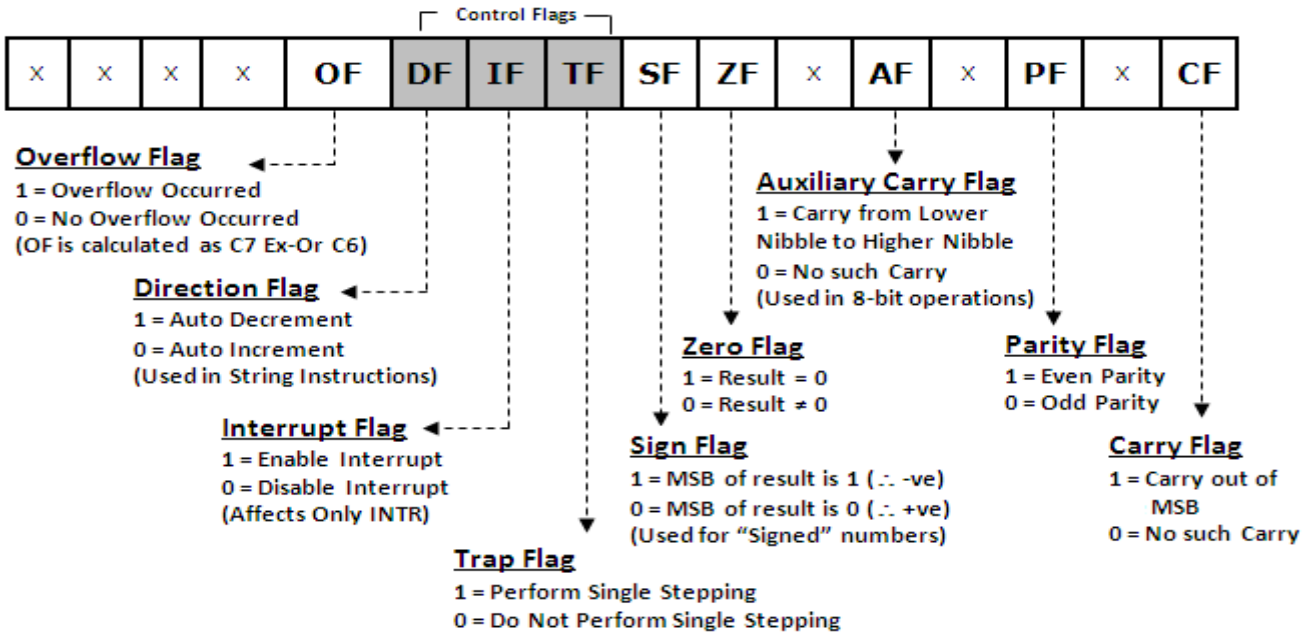
c) ALU (16-Bits)
It has a **16-bit ALU**. It performs 8 and 16-bit arithmetic and logic operations.

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.



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**.

*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.**

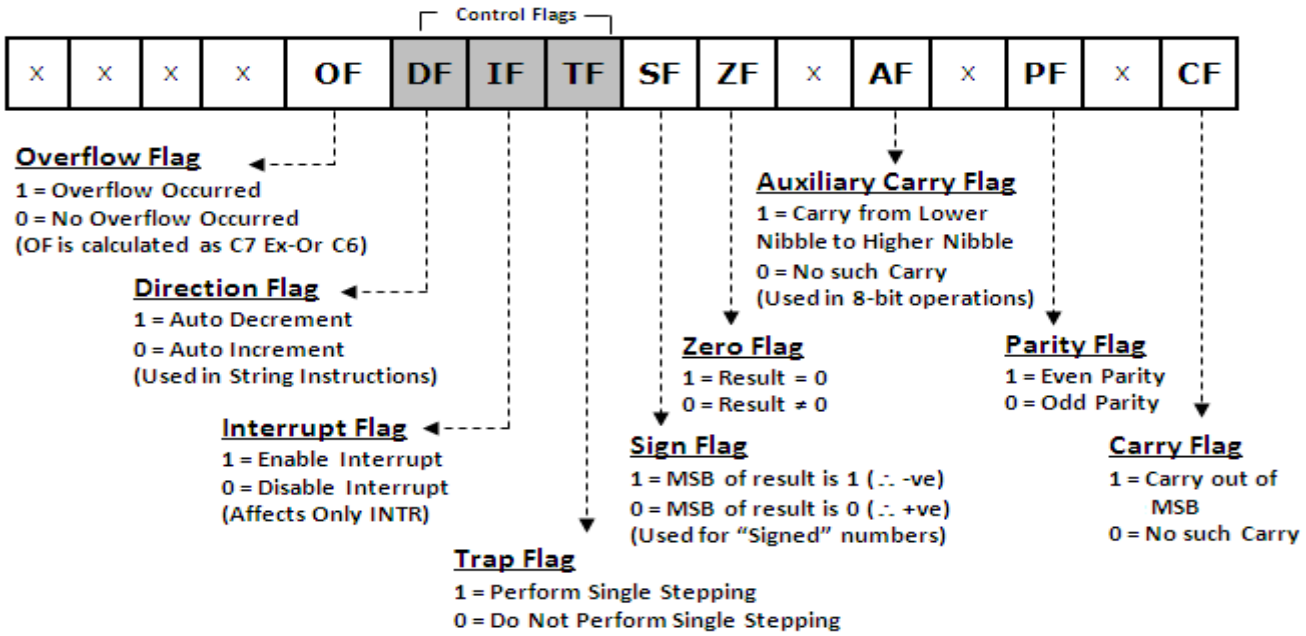
c) ALU (16-Bits)
It has a **16-bit ALU**. It performs 8 and 16-bit arithmetic and logic operations.

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.



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**.

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
; **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.

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 * 10H + BX

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

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.*

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] ← $Flag_H$, SS:[SP-2] ← $Flag_L$, SP ← SP – 2*

5) POPF

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

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]}
```

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*

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] ← $Flag_H$, SS:[SP-2] ← $Flag_L$, SP ← SP – 2*

5) POPF

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

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]}
```

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*

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] ← $Flag_H$, SS:[SP-2] ← $Flag_L$, SP ← SP – 2*

5) POPF

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

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]}
```

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*

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**.

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.

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

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

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**.

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.

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

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

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**!*

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.

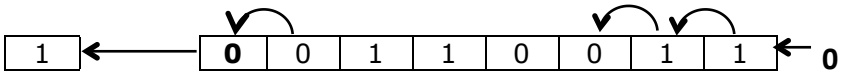
Destination: Register, Memory Location. #Please refer Bharat Sir's Lecture Notes for this ...

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

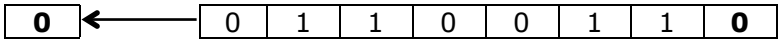
Assume:

Before Operation: BL = 0011 0011 and CF = 1

Carry **Destination**



After Operation: BL = 0110 0110 and CF = 0



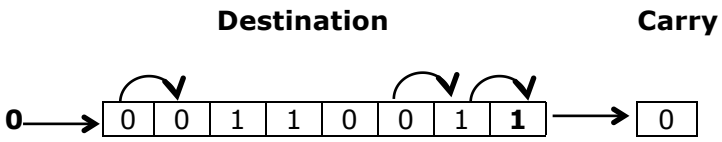
More examples:
MOV CL, 05H *; Load number of shifts in CL register.*
SAL BL, CL *; Left-Shift BL bits CL (5) number of times.*

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

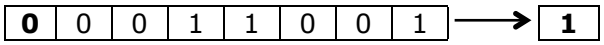
Bits are shifted 'count' number of times.
If count is 1, it is directly specified in the instruction.
If count > 1, it has to be given using CL register.
Eg: **SHR BL, 1** *; Right-Shift BL bits, once.*

Assume:

Before Operation: BL = 0011 0011 and CF = 0



After Operation: BL = 00011 1001 and CF = 1

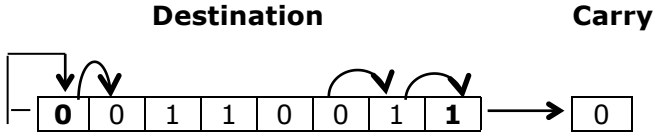


3) SAR destination, count
RIGHT-Shifts the bits of destination.
MSB placed **in MSB itself** (∴ Sign is preserved).
LSB shifted **into** the **CARRY**.

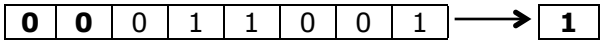
Bits are shifted 'count' number of times.
If count is 1, it is directly specified in the instruction.
If count > 1 it has to be given using CL register. ☺ For doubts contact Bharat Sir on 98204 08217
Destination: Register, Memory Location

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

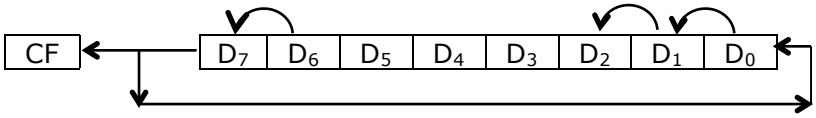


ROTATE INSTRUCTIONS

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.*

Carry **Destination**



More examples:

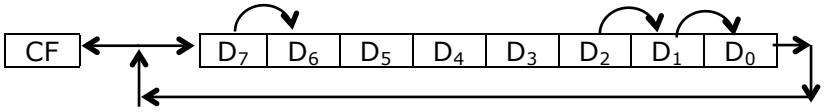
```
MOV CL, 05H      ; Load number of shifts in CL register.
ROL BL, CL       ; Left-Shift BL bits CL (5) number of times.
```

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.
```

Carry **Destination**

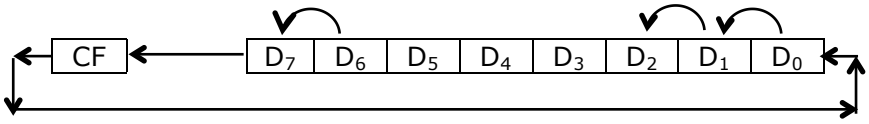


3) RCL destination, count

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.*

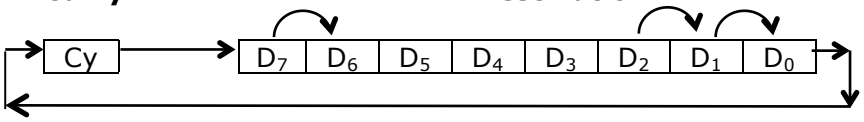
Carry **Destination**



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.*

Carry **Destination**



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

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]}
```

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:

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

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 ...

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**

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.
```

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*

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*

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.

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.

8086 PROGRAMMING

Q 1) **WAP to ADD two 16 bit numbers.**
Operands and the result should be in the data segment.

```
Data  SEGMENT                  // Starts a segment by the name Data

      A      DW    1234H       // Declares A  as 16-bits with value 1234H
      B      DW    5140H       // Declares B as 16-bits with value 5140H
      Sum    DW    ?           // Declares Result as a 16-bit word
      Carry  DB    00H         // Declare carry as an 8-bit variable with a value 0

Data  ENDS

Code  SEGMENT

      ASSUME CS: Code, DS: Data  // Informs the assembler about the correct segments

      MOV   AX, Data           // Puts segment address of Data into AX
      MOV   DS, AX             // Transfers segment address of Data from AX to DS

      MOV   AX, A              // Gets the value of A  into AX
      ADD   AX, B              // Adds the value of B into AX
      JNC   Skip               // If no carry then directly store the result
      MOV   Carry, 01H         // If carry produced then make variable "Carry=1"
Skip: MOV   Sum, AX            // Store the sum in the variable "Sum"

      INT3                     // Optional Breakpoint

Code ENDS

      END
```

Q 2) **WAP to ADD two 16 bit BCD numbers.**
Operands and the result should be in the data segment.

```
Data SEGMENT

      A      DW    1234H
      B      DW    5140H
      Sum     DW   ?
      Carry  DB    00H

Data  ENDS

Code  SEGMENT

      ASSUME CS: Code, DS: Data

      MOV   AX, Data
      MOV   DS, AX

      MOV   AX, A
      MOV   BX, B
      ADD   AL, BL
      DAA
      MOV   AL, AH
      ADC   AL, BH
      DAA
      JNC   Skip
      MOV   Carry, 01H

Skip: MOV   Sum, AX

      INT3

Code ENDS

      END
```

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

Q 3) **WAP to add a series of 10 bytes stored in the memory from locations 20,000H to 20,009H. Store the result immediately after the series.**

```
Code  SEGMENT

      ASSUME CS: Code

      MOV   AX, 2000H
      MOV   DS, AX

      MOV   SI, 0000H
      MOV   CX, OOOAH
      MOV   AX, 0000H
Back: ADD   AL, [SI]
      JNC   Skip
      INC   AH
Skip: INC   SI
      LOOP  Back
      MOV   [SI], AX

      INT3

Code ENDS

      END
```

Q 4) **WAP to transfer a block of 10 bytes from location 20,000H to 30,000H.**

```
Code  SEGMENT
      ASSUME CS: Code
      MOV   AX, 2000H
      MOV   DS, AX
      MOV   AX, 3000H
      MOV   ES, AX
      MOV   SI, 0000H
      MOV   DI, 0000H
      MOV   CX, 000AH
      CLD
REP   MOVSB
      INT3
Code ENDS
      END
```

Q 8) **WAP to multiply two 16-bit numbers. Operands and result in Data Segment.**

```
Data  SEGMENT
      A  DW 1234H
      B DW 1845H
      Result DD ?
Data  ENDS

Code  SEGMENT
      ASSUME CS: Code, DS: Data
      MOV   AX, Data
      MOV   DS, AX

      MOV   AX, A
      MUL   B
      LEA   BX, Result
      MOV   [BX], AX
      MOV   [BX+2], DX

      INT3
Code ENDS
      END
```

Q 9) **WAP to find "highest" in a given series of 10 numbers beginning from location 20,000H. Store the result immediately after the series.**

```
Code  SEGMENT
      ASSUME CS: Code
      MOV   AX, 2000H
      MOV   DS, AX

      MOV   SI, 0000H
      MOV   CX, 000AH
      MOV   AL, 00H
Back: CMP   AL, [SI]
      JNC   Skip
      MOV   AL, [SI]
Skip: INC   SI
      LOOP  Back
      MOV   [SI], AL

      INT3
Code ENDS
      END
```

Dear Students,
You have solved many more programs in the classroom.
Please refer to your lecture note book as well.
For doubts Call #BharatSir @9820408217.

Q 10) **WAP to find the number of –ve numbers in a series of 10 numbers from 20,000H. Store the result immediately after the series.**

```
Code  SEGMENT
      ASSUME CS: Code
      MOV   AX, 2000H
      MOV   DS, AX
      MOV   SI, 0000H
      MOV   CX, 000AH
      MOV   AH, 00H
Back: MOV   AL, [SI]
      RCL   AL, 01H
      JNC   Skip
      INC   AH
Skip: INC   SI
      LOOP  Back
      MOV   [SI], AH
      INT3
Code ENDS
      END
```

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

Q 11) **WAP to SORT a series of 10 numbers from 20,000H in ascending order.**

```
Code  SEGMENT
      ASSUME CS: Code
      MOV   AX, 2000H
      MOV   DS, AX
      MOV   CH, 09H
Bck2: MOV   SI, 0000H
      MOV   CL, 09H
Bck1: MOV   AX, [SI]
      CMP   AH, AL
      JNC   Skip
      XCHG  AL, AH
      MOV   [SI], AX
Skip: INC   SI
      DEC   CL
      JNZ   Bck1
      DEC   CH
      JNZ   Bck2
      INT3
Code ENDS
      END
```

Q 7) **Verify if "Block1" is a Palindrome.**
If yes, make "Pal = 1" in Data Seg.

```
Data  SEGMENT

      Block1 DB 'ABCDEEDCBA'
      Pal DB 00H

Data  ENDS
Extra SEGMENT

       Block2 DB 10 Dup(?)

Extra ENDS
Code  SEGMENT

      ASSUME CS: Code, DS: Data, ES: Extra
      MOV   AX, Data
      MOV   DS, AX
      MOV   AX, Extra
      MOV   ES, AX

      LEA   SI, Block1
      LEA   DI, Block2 + 9

      MOV   CX, 000AH
Back: CLD
      LODSB
      STD
      STOSB
      LOOP Back

      LEA   SI, Block1
      LEA   DI, Block2
      MOV   CX, 000AH
      CLD
REPZ  CMPSB
      JNZ   Skip
      MOV   Pal, 01H

Skip:INT3

Code ENDS

      END
```

Q 10) **WAP to find the number of –ve numbers in a series of 10 numbers from 20,000H. Store the result immediately after the series.**

```
Code  SEGMENT
      ASSUME CS: Code
      MOV   AX, 2000H
      MOV   DS, AX
      MOV   SI, 0000H
      MOV   CX, 000AH
      MOV   AH, 00H
Back: MOV   AL, [SI]
      RCL   AL, 01H
      JNC   Skip
      INC   AH
Skip: INC   SI
      LOOP  Back
      MOV   [SI], AH
      INT3
Code ENDS
      END
```

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

Q 11) **WAP to SORT a series of 10 numbers from 20,000H in ascending order.**

```
Code  SEGMENT
      ASSUME CS: Code
      MOV   AX, 2000H
      MOV   DS, AX
      MOV   CH, 09H
Bck2: MOV   SI, 0000H
      MOV   CL, 09H
Bck1: MOV   AX, [SI]
      CMP   AH, AL
      JNC   Skip
      XCHG  AL, AH
      MOV   [SI], AX
Skip: INC   SI
      DEC   CL
      JNZ   Bck1
      DEC   CH
      JNZ   Bck2
      INT3
Code ENDS
      END
```

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):

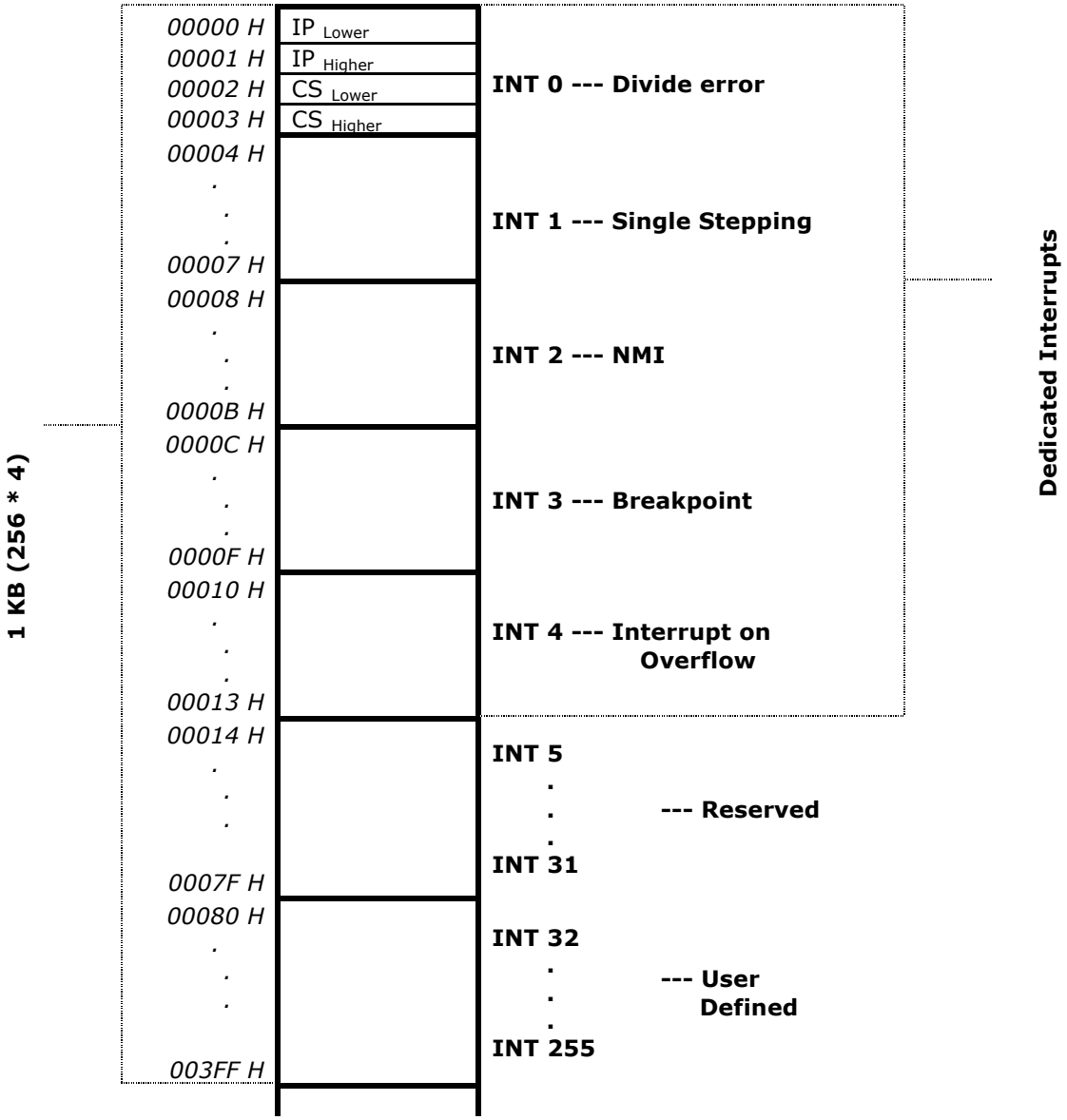
These interrupts are caused by writing the software interrupt instruction INTn where "n" can be any value from 0 to 255 (00H to FFH).
Hence all 256 interrupts can be invoked by software.

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**.


1 KB (256 * 4)
00000 H
00001 H
00002 H
00003 H
IP Lower
IP Higher
CS Lower
CS Higher
INT 0 --- Divide error
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
Dedicated Interrupts
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)

1) INT O (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.

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.

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.

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

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.

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.

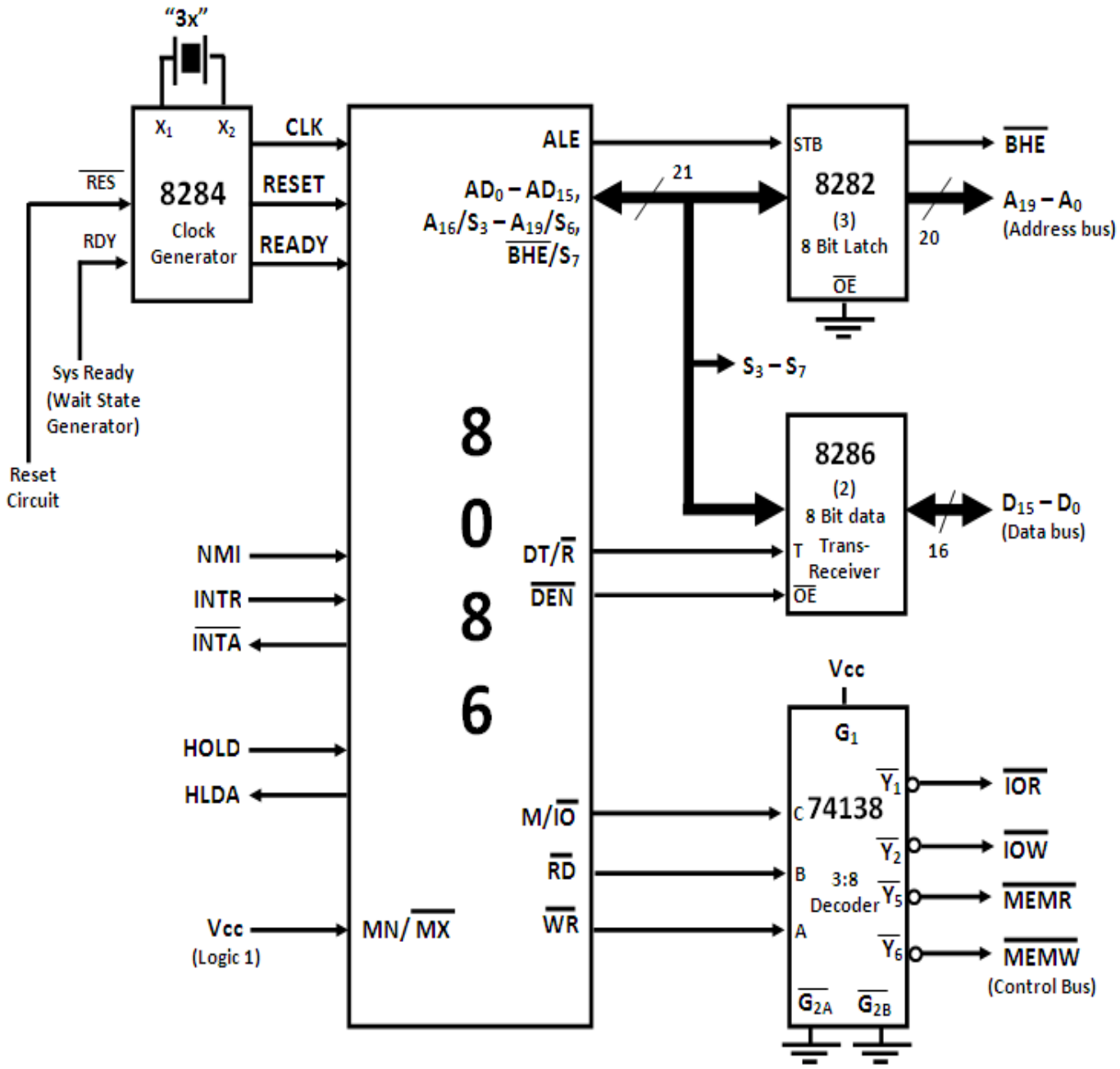
3) While calling the INT21H Dos interrupt, we must **first assign a correct value in AH** register.

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.

8086 MINIMUM MODE CONFIGURATION



1) **8086 works in Minimum Mode, when MN/ $\overline{MX}$ = 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 **$\overline{DEN}$** signal, while the direction of data is controlled by the **DT/ $\overline{R}$** signal. **$\overline{DEN}$** is connected to **$\overline{OE}$** and **DT/ $\overline{R}$** is connected to T. **Both $\overline{DEN}$ and DT/ $\overline{R}$ are given by 8086 itself.**

$\overline{DEN}$	DT/ $\overline{R}$	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/ $\overline{IO}$, $\overline{RD}$ and $\overline{WR}$ signals.**

For doubts contact Bharat Sir on 98204 08217

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

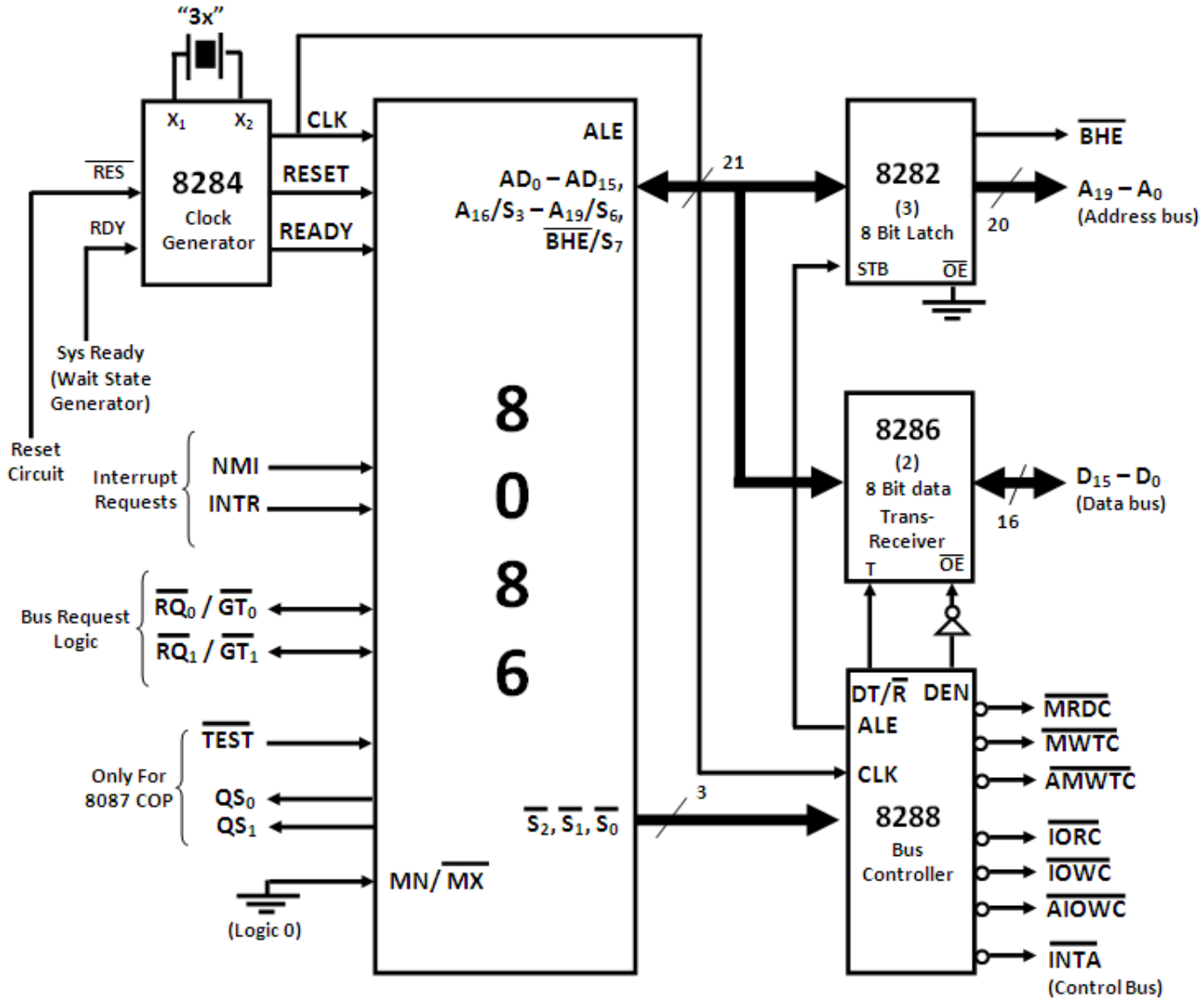
7) **M/ $\overline{IO}$, $\overline{RD}$, $\overline{WR}$** are **decoded by a 3:8 decoder like IC 74138.**

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

9) **$\overline{INTA}$ 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.**

8086 MAXIMUM MODE CONFIGURATION



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	**$\overline{INTA}$**
0	0	1	Read I/O Port	**$\overline{IORC}$**
0	1	0	Write I/O Port	**$\overline{IOWC}$** and **$\overline{AIOWC}$**
0	1	1	Halt	None
1	0	0	Instruction Fetch	**$\overline{MRDC}$**
1	0	1	Memory Read	**$\overline{MRDC}$**
1	1	0	Memory Write	**$\overline{MWTC}$** and **$\overline{AMWTC}$**
1	1	1	Inactive	None

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".

DATA FORMATS OF 8087

The data types supported by 8087 are as follows:

Integers	Decimal	Floating Point Numbers
1. Word Integer	1. Packed BCD	1. Short Real
2. Short Integer		2. Long Real
3. Long Integer		3. Temporary Real

Integers

- Here **MSB** represents the **sign** of the number.
- The **remaining** bits are used to represent the **magnitude** of the number.
- For **-ve numbers** the magnitude is stored in **2's complement form**.
- There are **three** data **types Word Int**., **Short Int**. and **Long Int**. of sizes 2 Bytes, 4 Bytes and 8 bytes respectively.

Packed BCD

- **Total size** of the number is **10 Bytes** (80 bits).
- Here a **number** is represented as a **series of 18 Packed BCD digits i.e. 9 bytes**.
- Hence each byte has two BCD digits.
- The **MSB of** the **10th byte** carries the **sign bit**, the **remaining 7 bits** are "**don't care**".
- Here a **–ve no** is **NOT stored** in its **2's complement form**.

Floating Point Numbers

- In some numbers, which have a fractional part, the position of the decimal point is not fixed as the number of bits before (or after) the decimal point may vary.
- **Eg: 0010.01001, 0.0001101, -1001001.01** etc.
- As shown above, the position of the decimal point is not fixed, instead it **"floats"** in the number. **For doubts contact Bharat Sir on 98204 08217**
- Such numbers are called Floating Point Numbers.
- Floating Point Numbers are stored in a "Normalized" form.

Normalization of a Floating Point Number

- Normalization is the process of shifting the point, left or right, so that there is only one non-zero digit to the left of the point. #Please refer Bharat Sir's Lecture Notes for this ...

Eg:

Floating Point Number		Normalized Number
0010.01001	⋯⋯➤	$1.001001*2^{1}$
0.0001101	⋯⋯➤	$1.101*2^{-4}$
-1001001.01	⋯⋯➤	$-1.00100101*2^{-6}$

- As seen above a Normalized number is represented as:

$(-1^{S}) \times (1.\mathbf{M}) \times (2^{E})$ Where: S = Sign, M = Mantissa and E = Exponent.

- As Normalized numbers are of the 1.M format, the "**1**" is not actually stored, it is instead **assumed**. This saves the storage space by 1 bit for each number.
- Also the Exponent is stored in the biased form by adding an appropriate bias value to it so that -ve exponents can be easily represented.

Advantages of Normalization.

1. Storing all numbers in a standard for makes **calculations easier** and **faster**.
2. By **not storing** the **1** (of 1.M format) for a number, considerable **storage space** is **saved**.
3. The **exponent** is **biased** so there is **no need** for **storing** its **sign bit** (as the biased exponent cannot be -ve).

There are three data types for Floating Point Numbers supported by 8087:

1. Short Real Format

- **32 bits** are used to store the **number**.
- **23 bits** are used for the **Mantissa**.
- **8 bits** are used for the Biased **Exponent**.
- **1 bit** used for the **Sign** of the number.
- The **Bias** value is $(\mathbf{127})_{10.}$
- The range is $\pm\mathbf{1*10^{-38}}$ **to** $\pm\mathbf{3*10^{38}}$ approximately.
- It is called as the **Single Precision Format** for Floating-Point Numbers.

2. Long Real Format

- **64 bits** are used to store the **number**.
- **52 bits** are used for the **Mantissa**.
- **11 bits** are used for the Biased **Exponent**.
- **1 bit** used for the **Sign** of the number.
- The **Bias** value is $(\mathbf{1023})_{10.}$
- The range is $\pm\mathbf{10^{-308}}$ **to** $\pm\mathbf{10^{308}}$ approximately.
- It is called as the **Double Precision Format** for Floating-Point Numbers.

3. Temporary Real Format

- **80 bits** are used to store the **number**.
- **64 bits** are used for the **Mantissa**.
- **15 bits** are used for the Biased **Exponent**.
- **1 bit** used for the **Sign** of the number.
- The **Bias** value is $(\mathbf{16383})_{10.}$
- The range is $\pm\mathbf{10^{-4932}}$ **to** $\pm\mathbf{10^{4932}}$ approximately.
- **1** (of 1.M format) is **present at 63rd bit**, hence the decimal point is assumed between the 62nd and the 63rd bit. **For doubts contact Bharat Sir on 98204 08217**
- **8087 stores** numbers **internally** in **this format** as it has the **biggest range**.
- It is also called as **Extended Precision Format** or the **internal format** of 8087.

Short Real – 32 bit format – Bias value 127

S	E	M
Sign (1)	Biased Exponent (8)	Mantissa (23)

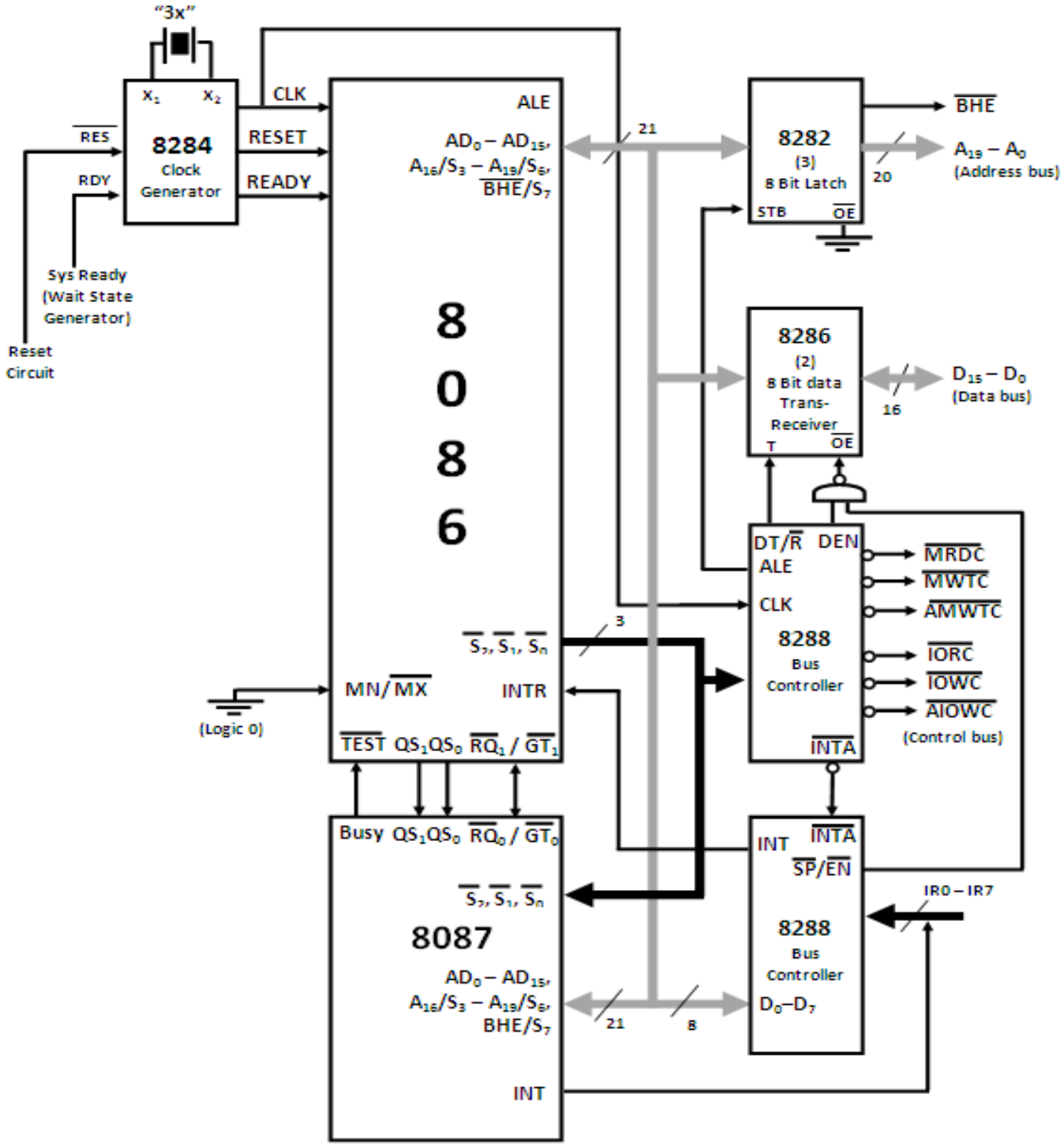
Long Real – 64 bit format – Bias value 1023

S	E	M
Sign (1)	Biased Exponent (11)	Mantissa (52)

Temp Real – 80 bit format – Bias value 16383

S	E	M
Sign (1)	Biased Exponent (15)	Mantissa (64)

CO-PROCESSOR CONFIGURATION --- 8086 WITH 8087



1) As a co-processor (8087) is connected to 8086, 8086 operates in **Maximum Mode**.
   Hence **MN/ $\overline{\text{MX}}$ is grounded**.

2) **8284** provides the common **CLK, RESET, READY** signals,
   8282 are used to **latch** the **address**, **8286** are used as **data transreceivers**,
   8288 generates **control signals** using S_2, S_1, S_0 as input from the currently active processor,
   8259 PIC is used to accept the **interrupt from 8087** and send it **to** the **μP**.

3) This interface is also called a **Closely Coupled Co-Processor configuration**.
   Here **8086** is called as the **Host** and 8087 as **Co-Processor**, as it cannot operate all by itself.

4) We write a **homogeneous program** which contains both 8086 as well as 8087 instructions.

5) **Only 8086 can fetch instructions**, but these **instructions also enter 8087**.
   8087 treats 8086 instructions as NOP.

6) **ESC** is used as a **prefix for 8087 instructions**.
   When an instruction with ESC prefix (5 MSB bits as **11011**) is encountered, 8087 is activated.

7) The **ESC instruction** is **decoded by both** 8086 and 8087.

8) If **ESC is present 8087 executes** the instruction and 8086 discards it.
   If **ESC is not preset then 8086 executes the instruction** and 8087 discards it.

9) Once 8087 **begins execution** it makes the **BUSY o/p high** which is connected to the **$\overline{\text{TEST}}$** of μP.
   Now **8087** is **executing its instruction** and **8086** moves ahead with **its next instruction**.
   Hence ***MULTIPROCESSING*** *takes place.* **For doubts contact #BharatSir @9820408217**

10) **During execution**, **if 8087 needs** to read/write more data (operands) from **the memory**, then it does so by **stealing bus cycles** from the μP in the following manner:
    The **$\overline{\text{RQ}}$ / $\overline{\text{GT}}$** of 8087 is connected to **$\overline{\text{RQ}}$ / $\overline{\text{GT}}$** of the μP.
    8087 gives an active low **Request pulse**.
    8086 completes the current bus cycle and **gives** the **grant pulse** and **enters** the **Hold state**.
    8087 uses the **shared system bus** to **perform** the **data transfer with** the **memory**.
    8087 gives the **release pulse** and **returns** the **system bus** back **to** the **μP**.

11) If **8086** requires the result of the 8087 operation, it first **executes** the **WAIT** instruction.
    WAIT makes the **μP check** the **$\overline{\text{TEST}}$** pin.
    If the **TEST** pin is **high** (**8087** is **BUSY**), then the **μP enters WAIT state**
    It **comes out** of it only **when** **$\overline{\text{TEST}}$** is low (**8087** has **finished its execution**).
    Thus 8086 gets the correct result of an 8087 operation.

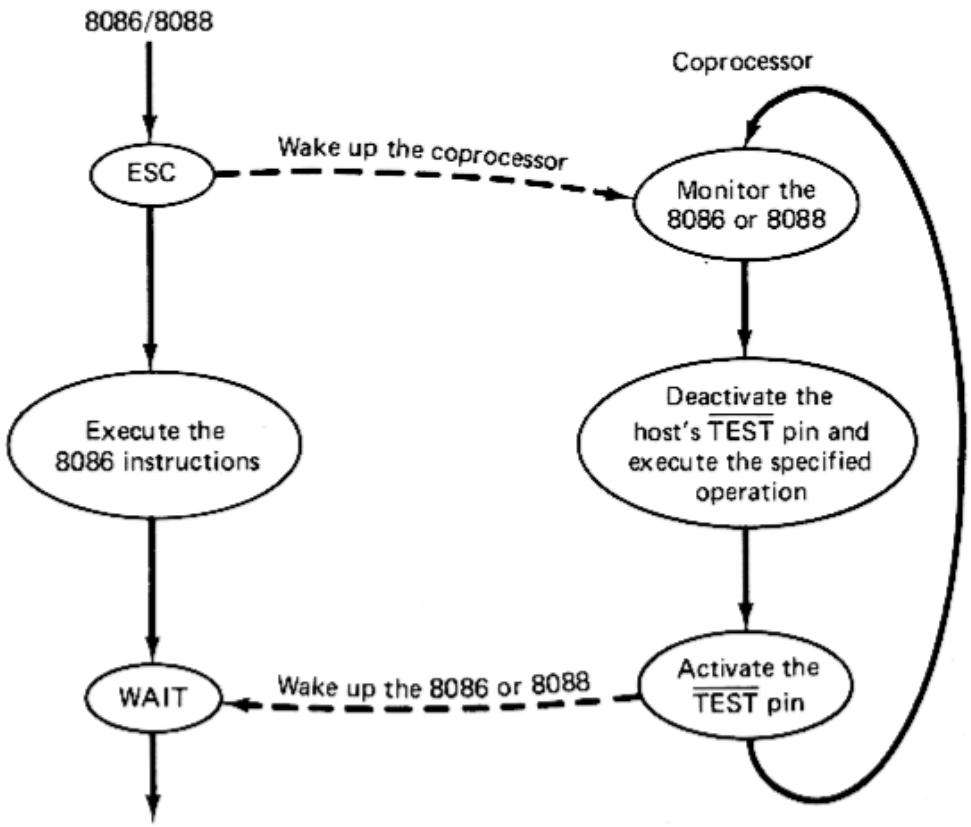
12) During the execution **if** an **exception occurs**, which is **unmasked**, **8087 interrupts μP** using the **INT** o/p **pin through** the **PIC 8259**.

13) The **QS_0** and **QS_1** lines are used by 8087 to monitor the queue of 8086.
8087 needs to know when 8086 will decode the ESC instruction so it synchronizes its queue with 8086 using **QS_0** and **QS_1** as follows: #Please refer Bharat Sir's Lecture Notes for this ...

QS_1	QS_0	8087 Operation
0	0	NOP
0	1	8087 removes Opcode from Queue and compares 5 MSB bits with 11011.
1	0	8087 clears its queue.
1	1	8087 removes operand if earlier comparison with Opcode is successful.

This is the **complete inter-processor communication** between 8086 and 8087 to form a Homogeneous System.

FLOWCHART (Optional – only for understanding)



Programmable Interrupt Controller

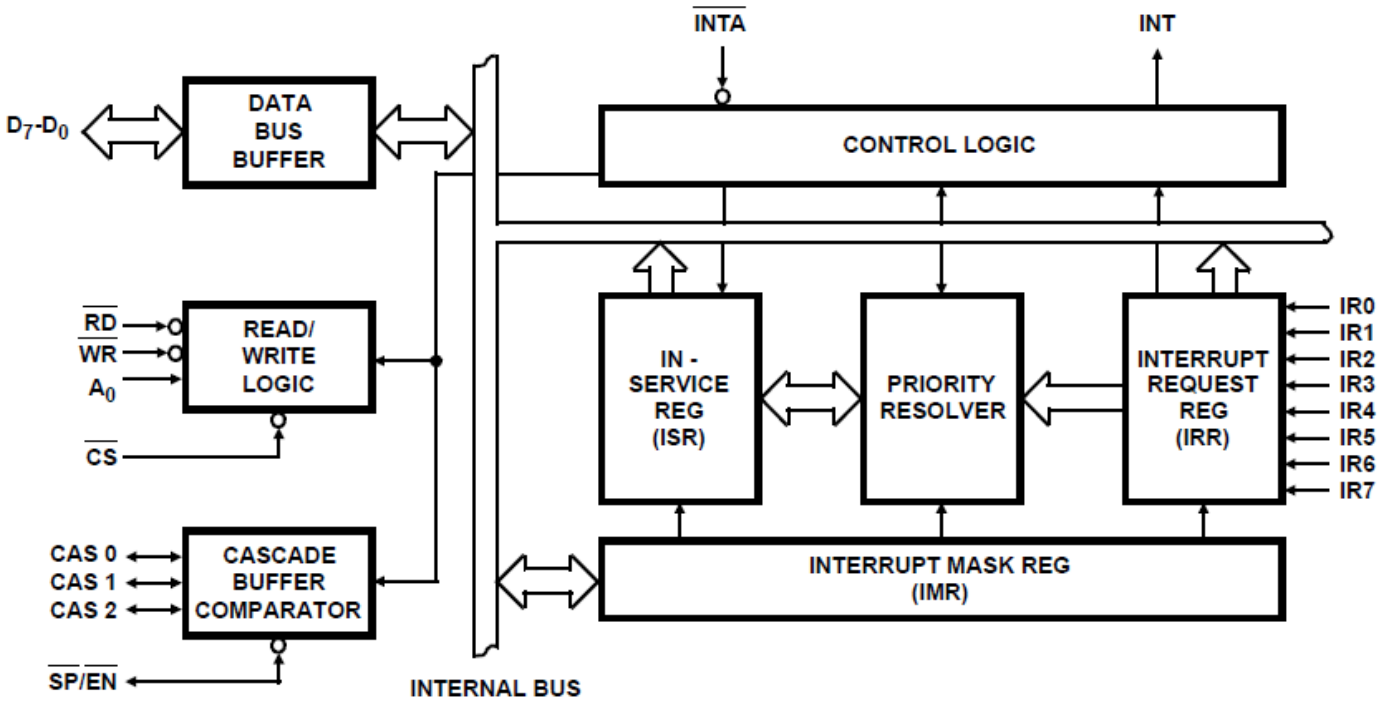
8259 | PIC

WWW.BHARATACHARYAEDUCATION.COM

Salient Features

1) PIC 8259 is a Programmable Interrupt Controller that can work with 8085, 8086 etc.
2) **It is used to increase the number of interrupts.**
3) A single 8259 provides 8 interrupts while a cascaded configuration of 1 master 8259 and 8 slave 8259s can provide up to 64 interrupts.
4) 8259 can handle **edge as well as level triggered** interrupts.
5) 8259 has a **flexible priority** structure. ☺ For doubts contact Bharat Sir on 98204 08217
6) In 8259 interrupts can be **masked** individually.
7) The **Vector address** of the interrupts is **programmable**.
8) 8259 has to be **compulsorily initialized** by giving commands, to decide several properties such as Vector Numbers, Priority, Masking, Triggering etc.
9) In a **cascaded configuration**, **each 8259 has to be individually initialized**, master as well as each slave.

ARCHITECTURE OF 8259



1) Interrupt Request Register (IRR)

8259 has **8 interrupt** input lines **IR_7 … IR_0**.
The IRR is an **8-bit register** having **one bit** for **each** of the **interrupt** lines.
When an **interrupt request** occurs on any of these lines, the **corresponding bit** is **set** in the Interrupt Request Register (**IRR**).

2) In-Service Register (InSR)

It is an **8-bit** register, which **stores** the **level** of the Interrupt Request, which is **currently** being **serviced**.

3) Interrupt Mask Register (IMR)

It is an **8-bit** register, which stores the **masking pattern** for the interrupts of 8259. It stores **one bit per interrupt level.**

4) Priority Resolver

It **examines** the **IRR**, **InSR**, and **IMR** and determines which interrupt is of **highest priority** and should be sent to the µP.

5) Control Logic

It has **INT output connected to** the **INTR** of the µP, to **send** the **Interrupt** to µP.

It also has the **$\overline{\text{INTA}}$ input** signal **connected to** the **$\overline{\text{INTA}}$** of the µP, to **receive** the interrupt **acknowledge**. It is also used to control the remaining blocks.

6) Data Bus Buffer

It is a bi-directional buffer used to **interface** the internal **data bus** of 8259 with the external (system) data bus.

7) Read/Write Logic

It is used to accept the **$\overline{\text{RD}}$** , **$\overline{\text{WR}}$** , A_0 and CS signal.

It also holds the Initialization Command Words (ICW's) and the Operational Command Words (OCW's).

8) Cascade Buffer / Comparator

It is used in **cascaded mode** of operation.
It has two components:

i. CAS_2, CAS_1, CAS_0 lines:

These lines are **output for the master**, **input for the slave**.
The **Master sends** the **address of** the **slave** on these lines (hence output).
The **Salves read** the **address** on these lines (hence input).
As there are 8 interrupt levels for the Master, there are **3 CAS lines (2^3 = 8).**

ii. $\overline{\text{SP}}$ / $\overline{\text{EN}}$ (Slave Program/Master Enable): **For doubts contact Bharat Sir on 98204 08217**

In **Buffered Mode**, it **functions** as the **EN line** and is used to **enable** the **buffer**.
In **Non buffered mode**, it **functions** as the **SP output line**.
For Master 8259 **SP** should be **high**, and **for** the **Slave SP** should be **low**.

INTERFACING AND WORKING OF A "SINGLE" 8259

A single 8259 can accept 8 interrupts.
Whenever a device interrupts 8259, 8259 will interrupt the **µP on INTR pin.**
Hence, first the **INTR** signal of the µP **should be enabled** using the **STI** instruction**.**
8259 is **initialized** by giving **ICW1** and **ICW2** (compulsory) and **ICW4** (optional).
Note that **ICW3** is **not given** as Single 8259 is used. OCWs are given if required.

Once 8259 is **initialized**, the **following sequence** of events takes place when one or more **interrupts occur** on the IR lines of the 8259.

1) The **corresponding bit** for an interrupt is **set in IRR**.

2) The **Priority Resolver checks** the 3 registers:
   IRR (for highest interrupt request)
   IMR (for the masking Status)
   InSR (for the current level serviced)
   and **determines** the **highest priority** interrupt.
   It **sends** the **INT** signal **to** the **µP**.

3) The **µP finishes** the **current instruction** and **acknowledges** the interrupt by **sending** the **first** $\overline{\textbf{INTA}}$ **pulse**.

4) On receiving the first $\overline{\textbf{INTA}}$ signal, the **corresponding bit** in the **InSR** is **set** (indicating that now this interrupt is in service) and the **bit** in the **IRR** is **reset** (to indicate that the request is accepted).
   For doubts contact Bharat Sir on 98204 08217
   8259 now **prepares** to send the Vector number **N** to the µP on the data bus.

5) The **µP sends the second** $\overline{\textbf{INTA}}$ **pulse** to 8259.

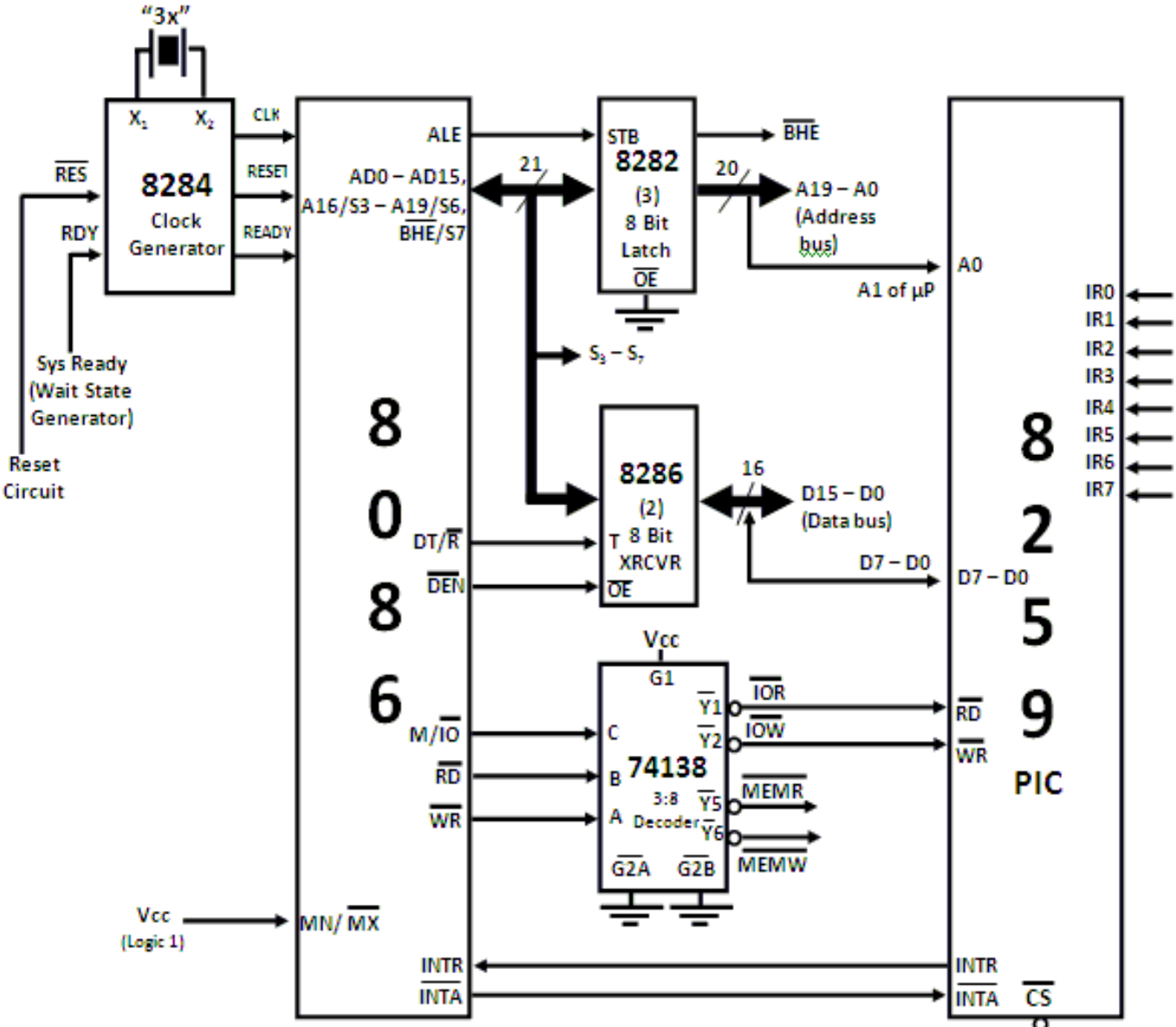
6) In response to the 2nd $\overline{\textbf{INTA}}$ pulse, **8259 sends the one byte Vector Number N to µP**.

7) Now the **µP multiplies N x 4**, to get the values of CS and IP **from the IVT**.

8) **In** the **AEOI Mode** the **InSR bit is reset** at this point, **otherwise** it **remains set until** an **appropriate EOI** command is given at the End of the ISR.

9) The **µP pushes** the contents of **Flag** Register, **CS**, **IP**, into the **Stack, Clears IF and TF** and **transfers program to** the **address** of the **ISR**.. *#Please refer Bharat Sir's Lecture Notes for this ...*

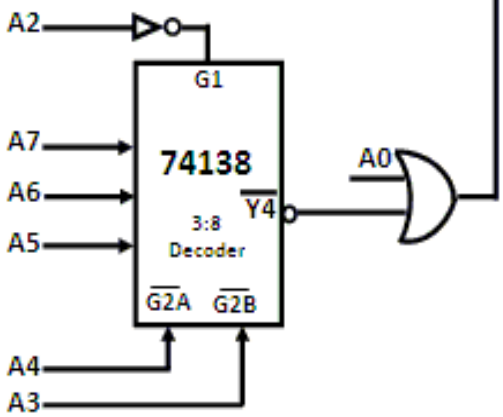
10) **The ISR thus begins**.



I/O Map of 8259 at 80H

	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
ICW1	1	0	0	0	0	0	0	0	80H
ICW2	1	0	0	0	0	0	1	0	82H

Chip Selection
Internal Selection
Bank Selection



Interfacing and Working of "CASCADED" 8259

When **more than one 8259s** are connected to the µP, it is called as a **Cascaded configuration**.
A Cascaded configuration **increases** the **number of interrupts** handled by the system.
As the **maximum** number of **8259s** interfaced can be **9** (1 Master and 8 Slaves) the **Maximum** number of **interrupts** handled can be **64**.

The **master 8259** has $\overline{\textbf{SP}}$ **/** $\overline{\textbf{EN}}$ **= +5V** and the **slave** has $\overline{\textbf{SP}}$ **/** $\overline{\textbf{EN}}$ **= 0V**.

Each slave's INT output is **connected** to the **IR input** of the **Master**.
The **INT** output **of** the **Master** is **connected** to the **INTR** input of the **µP**.
The **master addresses** the individual **slaves through** the **CAS_2, CAS_1, CAS_0 lines** connected from the master to each of the slaves.
First the **INTR** signal of the µP **should be enabled** using the **STI** instruction**.**
Each 8259 (Master or Slave) **has** its **own address** and **has to be initialized separately** by giving ICWs as per requirement.

When an **interrupt request** occurs **on** a **SLAVE**, the events are performed:

1) The **slave 8259 resolves** the **priority** of the interrupt and **sends** the **interrupt to** the **master** 8259.

2) The **master resolves** the **priority** among its slaves and **sends** the **interrupt to** the **µP**.

3) The **µP finishes the current instruction** and **responds** to the interrupt **by sending 2** $\overline{\textbf{INTA}}$ **pulses**.

4) **In response to the first** $\overline{\textbf{INTA}}$ **pulse** the **following events occur**:

   i. The master **sends** the **3-bit salve identification number** on the **CAS lines**.

   ii. The **Master sets** the **corresponding bit** in **its InSR**.

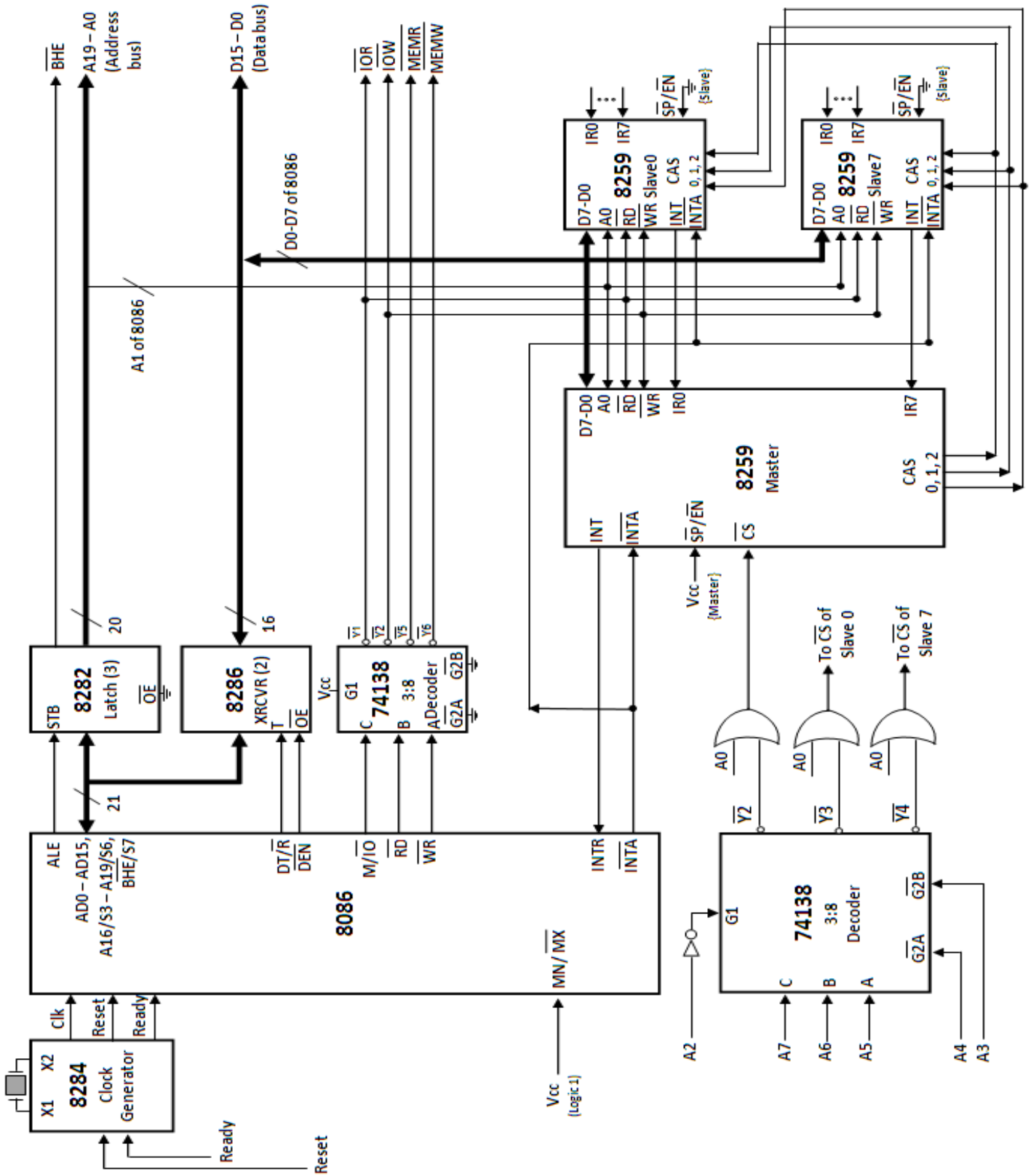
   iii. The **Slave identifies** its number on the **CAS lines** and **sets** the **corresponding bit in its InSR**.

5) In **response to** the **second** $\overline{\textbf{INTA}}$ pulse the **slave** places **Vector Number N** on the data bus.

6) **During** the **2nd** $\overline{\textbf{INTA}}$ pulse the **InSR bit** of the **slave** is **cleared** in **AEOI mode**, **otherwise** it is **cleared by** the **EOI command** at the end of the ISR.

7) The **µP pushes** the contents of **Flag** Register, **CS**, **IP**, into the **Stack, Clears IF and TF** and **transfers program to** the **address** of the **ISR**.. *#Please refer Bharat Sir's Lecture Notes for this ...*

8) **The ISR thus begins**.


BHE
A19 – A0 (Address bus)
D15 – D0 (Data bus)
IOR
IOW
MEMR
MEMW
D0-D7 of 8086
A1 of 8086
20
16
21
8282 Latch (3)
8286 XRCVR (2)
74138 3:8 Decoder
8086
8284 Clock Generator
8259 Slave0
8259 Slave7
8259 Master
ALE
AD0 – AD15, A16/S3 – A19/S6, BHE/S7
DT/R
DEN
M/IO
RD
WR
INTR
INTA
MN/MX
Clk
Reset
Ready
X1
X2
SP/EN
{Slave}
{Master}
Vcc
(Logic 1)
To CS of Slave 0
To CS of Slave 7
A0
A2
A3
A4
A5
A6
A7

I/O map		A7	A6	A5	A4	A3	A2	A1	A0	I/O
8259	ICW1	**0**	**1**	**0**	0	0	0	**0**	0	**40**
Master	ICW2	**0**	**1**	**0**	0	0	0	**1**	0	**42**
8259	ICW1	**0**	**1**	**1**	0	0	0	**0**	0	**60**
Slave 0	ICW2	**0**	**1**	**1**	0	0	0	**1**	0	**62**
8259	ICW1	**1**	**0**	**0**	0	0	0	**0**	0	**80**
Slave 7	ICW2	**1**	**0**	**0**	0	0	0	**1**	0	**82**

PRIORITY MODES OF 8259

Fully Nested Mode (FNM)

It is the **default mode** of 8259.
It is a **fixed priority** mode.
IR_0 has the **highest** priority and **IR_7** has the **lowest** priority.
It is preferred for **"Single"** 8259.

Special Fully Nested Mode (SFNM)

This mode can be **used for** the **Master 8259 in** a **cascaded** configuration.
Its **priority structure** is fixed and is the **same as FNM** (IR_0 highest and IR_7 lowest).
Additionally, in SFNM, the **Master would recognize a higher priority interrupt from a slave,** *whose another interrupt is currently being serviced*. This is **possible only in SFNM**.

Rotating Priority Modes

There are **two** rotating priority modes:
Automatic Rotation and Specific Rotation

Automatic Rotation Mode

This is a rotating priority mode.
It is **preferred** when **several interrupt** sources are of **equal priority**.
In this mode, **after** a device receives **service**, it **gets** the **lowest priority**.
All other priorities rotate subsequently. **For doubts contact Bharat Sir on 98204 08217**
Eg: If IR_2 is has just been serviced, it will get the lowest priority.

Specific Rotation Mode

It is **also** a **rotating** priority mode, **but here** the **user can select** any **IR level for lowest priority**, and thus fix all other priorities.

Special Mask Mode (SMM)

Usually 8259 **prevents interrupt requests lower or equal** to the interrupt, which is **currently** in service.
In SMM 8259 **permits interrupts** of **all levels** (lower or higher) **except** the one **currently** in service.
As we are specially masking the current interrupt, it is called Special Mask Mode.
This mode is preferred when we don't want priority

Poll Mode

Here the **INT line** of 8259 is **not used** hence 8259 cannot interrupt the **µp**.
Instead, the **µP will give Poll command** to 8259 using OCW3.
In **return**, **8259 provides** a **Poll Word** to the µP.
The Poll Word **indicates** the **highest priority interrupt**, which requires service.

Poll Word

I	x	x	x	x	W_2	W_1	W_0

1= Valid Interrupt
0 = No valid interrupt

Level No of the highest priority interrupt to be serviced

W_2	W_1	W_0
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Thereafter the **µP services the interrupt**. **For doubts contact Bharat Sir on 98204 08217**
Advantage: The µP's program is not disturbed. It can be used when the ISR is common for several Interrupts. It can be used to increase the total number of interrupts beyond 64.
Drawback: If the polling interval is too large, the interrupts will be serviced after long intervals. If the polling interval is small, lot of time may be wasted in unnecessary polls.

Buffered Mode

In this mode $\overline{\textbf{SP}}$ / $\overline{\textbf{EN}}$ becomes **low** during $\overline{\textbf{INTA}}$ cycle.
This signal is used to enable the buffer.

EOI – (End Of Interrupt)

When the **µP responds** to an interrupt request by **sending the first** $\overline{\textbf{INTA}}$ signal, the **8259 sets** the **corresponding bit** in the In Service Register (**InSR**).
This **begins** the **service** of the interrupt.
When this bit in the In Service Register is **cleared**, **it is called** as **End of Interrupt (EOI)**.

EOI Modes:

1) Normal EOI Mode:

Here an EOI Command is necessary. The EOI Command is given by the programmer at the end of the ISR. It causes 8259 to clear the bit from In Service Register. There are two types of EOI Commands:

Non Specific EOI Command:

Here the programmer doesn't specify the Bit number to be cleared. 8259 automatically clears the highest priority bit from In Service Register.

Specific EOI Command:

Here the programmer specifies the Bit number to be cleared from In Service Register.

2) Auto EOI Mode (AEOI):

In AEOI mode the EI command is not needed. Instead, 8259 will itself clear the corresponding bit from In Service Register at the end of the 2nd $\overline{\textbf{INTA}}$ pulse.

PRIORITY MODES OF 8259

Fully Nested Mode (FNM)

It is the **default mode** of 8259.
It is a **fixed priority** mode.
IR_0 has the **highest** priority and **IR_7** has the **lowest** priority.
It is preferred for **"Single"** 8259.

Special Fully Nested Mode (SFNM)

This mode can be **used for** the **Master 8259 in** a **cascaded** configuration.
Its **priority structure** is fixed and is the **same as FNM** (IR_0 highest and IR_7 lowest).
Additionally, in SFNM, the **Master would recognize a higher priority interrupt from a slave,** *whose another interrupt is currently being serviced*. This is **possible only in SFNM**.

Rotating Priority Modes

There are **two** rotating priority modes:
Automatic Rotation and Specific Rotation

Automatic Rotation Mode

This is a rotating priority mode.
It is **preferred** when **several interrupt** sources are of **equal priority**.
In this mode, **after** a device receives **service**, it **gets** the **lowest priority**.
All other priorities rotate subsequently. **For doubts contact Bharat Sir on 98204 08217**
Eg: If IR_2 is has just been serviced, it will get the lowest priority.

Specific Rotation Mode

It is **also** a **rotating** priority mode, **but here** the **user can select** any **IR level for lowest priority**, and thus fix all other priorities.

Special Mask Mode (SMM)

Usually 8259 **prevents interrupt requests lower or equal** to the interrupt, which is **currently** in service.
In SMM 8259 **permits interrupts** of **all levels** (lower or higher) **except** the one **currently** in service.
As we are specially masking the current interrupt, it is called Special Mask Mode.
This mode is preferred when we don't want priority

Poll Mode

Here the **INT line** of 8259 is **not used** hence 8259 cannot interrupt the **µp**.
Instead, the **µP will give Poll command** to 8259 using OCW3.
In **return**, **8259 provides** a **Poll Word** to the µP.
The Poll Word **indicates** the **highest priority interrupt**, which requires service.

Poll Word

I	x	x	x	x	W_2	W_1	W_0

1= Valid Interrupt
0 = No valid interrupt

Level No of the highest priority interrupt to be serviced

W_2	W_1	W_0
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Thereafter the **µP services the interrupt**. **For doubts contact Bharat Sir on 98204 08217**
Advantage: The µP's program is not disturbed. It can be used when the ISR is common for several Interrupts. It can be used to increase the total number of interrupts beyond 64.
Drawback: If the polling interval is too large, the interrupts will be serviced after long intervals. If the polling interval is small, lot of time may be wasted in unnecessary polls.

Buffered Mode

In this mode $\overline{\textbf{SP}}$ / $\overline{\textbf{EN}}$ becomes **low** during $\overline{\textbf{INTA}}$ cycle.
This signal is used to enable the buffer.

EOI – (End Of Interrupt)

When the **µP responds** to an interrupt request by **sending the first** $\overline{\textbf{INTA}}$ signal, the **8259 sets** the **corresponding bit** in the In Service Register (**InSR**).
This **begins** the **service** of the interrupt.
When this bit in the In Service Register is **cleared**, **it is called** as **End of Interrupt (EOI)**.

EOI Modes:

1) Normal EOI Mode:

Here an EOI Command is necessary. The EOI Command is given by the programmer at the end of the ISR. It causes 8259 to clear the bit from In Service Register. There are two types of EOI Commands:

Non Specific EOI Command:

Here the programmer doesn't specify the Bit number to be cleared. 8259 automatically clears the highest priority bit from In Service Register.

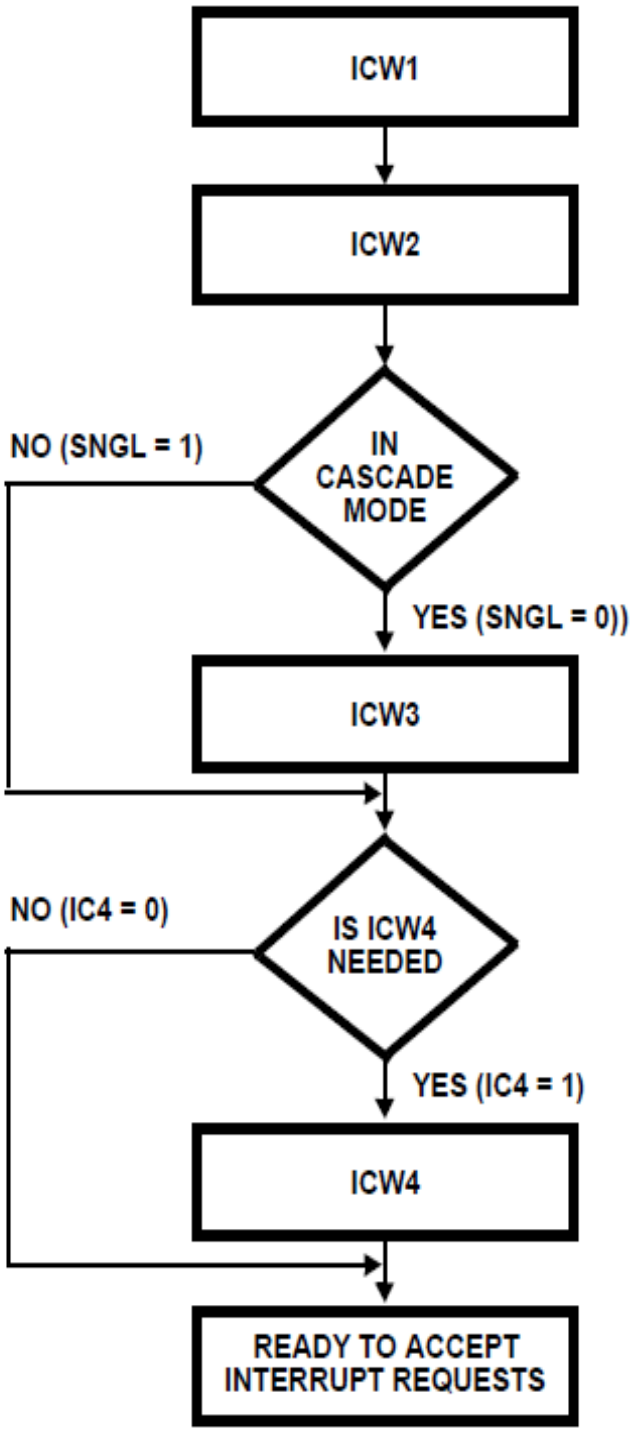
Specific EOI Command:

Here the programmer specifies the Bit number to be cleared from In Service Register.

2) Auto EOI Mode (AEOI):

In AEOI mode the EI command is not needed. Instead, 8259 will itself clear the corresponding bit from In Service Register at the end of the 2nd $\overline{\textbf{INTA}}$ pulse.

Initialization of 8259



As seen above there are **two types** of **commands**, **Initialization Command Words (ICWs)** and **Operational Command Words (OCWs).**

ICWs

ICWs have to be **given during** the **initialization** of 8259 (i.e. **before** the µP can start **using 8259**).
For doubts contact Bharat Sir on 98204 08217
ICW1 and **ICW2** are **compulsory**.
If Cascaded, **ICW3** has to be given.
Whether **ICW4** is **required** or not, is **specified in the ICW1**.
If ICW4 is **required**, it has to be **written**.
It is **important** that the ICWs are **written in** the **above sequence only**.
None of the ICWs can be individually repeated, but the entire initialization can be repeated if required.

OCWs

OCWs are **given during** the **operation** of 8259 (i.e. **after** the µP has **started using** 8259).
OCWs are **not compulsory**.
OCWs **do not** have to be given in a specific order.
OCWs can be individually repeated.

They are mainly used to alter the **masking** status and the **operation modes** of 8259.

ICW1

A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	A_7	A_6	A_5	1	LTIM	ADI	SNGL	IC4

1 = ICW4 needed
0 = No ICW4 needed

1 = Single
0 = Cascade Mode

CALL address interval
1 = Interval of 4
0 = Interval of 8

1 = Level triggered mode
0 = Edge triggered mode

A_7 - A_5 of Interrupt vector address
(MCS-80/85 mode only)

ICW2

A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	A_{15} / T_7	A_{14} / T_6	A_{13} / T_5	A_{12} / T_4	A_{11} / T_3	A_{10}	A_9	A_8

A_{15} - A_8 of interrupt vector address
(MCS80/85 mode)

T_7 - T_3 of interrupt vector address
(8086/8088 mode)

ICW3 (MASTER DEVICE)

A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	S_7	S_6	S_5	S_4	S_3	S_2	S_1	S_0

1 = IR input has a slave
0 = IR input does not have a slave

ICW3 (SLAVE DEVICE)

A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	0	0	0	0	0	ID_2	ID_1	ID_0

SLAVE ID (NOTE)

0	1	2	3	4	5	6	7
0	1	0	1	0	1	0	1
0	0	1	1	0	0	1	1
0	0	0	0	1	1	1	1

ICW4

A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	0	0	0	SFNM	BUF	M/S	AEOI	µPM

1 = 8086/8088 mode
0 = MCS-80/85 mode

1 = Auto EOI
0 = Normal EOI

0	X	- Non buffered mode
1	0	- Buffered mode slave
1	1	- Buffered mode master

1 = Special fully nested moded
0 = Not special fully nested mode

OCW1

A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	M_7	M_6	M_5	M_4	M_3	M_2	M_1	M_0

Interrupt Mask
1 = Mask set
0 = Mask reset

OCW2

A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	R	SL	EOI	0	0	L_2	L_1	L_0

IR LEVEL TO BE ACTED UPON

0	1	2	3	4	5	6	7
0	1	0	1	0	1	0	1
0	0	1	1	0	0	1	1
0	0	0	0	1	1	1	1

0	0	1	Non-specific EOI command	End of interrupt
0	1	1	† Specific EOI command	
1	0	1	Rotate on non-specific EOI command	Automatic rotation
1	0	0	Rotate in automatic EOI mode (set)	
0	0	0	Rotate in automatic EOI mode (clear)	
1	1	1	† Rotate on specific EOI command	
1	1	0	† Set priority command	Specific rotation
0	1	0	No operation	

† L_0 - L_2 are used

OCW3

A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	ESMM	SMM	0	1	P	RR	RIS

READ REGISTER COMMAND

0	1	0	1
0	0	1	1
No Action		Read IR reg on next RD pulse	Read IS reg on next RD pulse

1 = Poll command
0 = No poll command

SPECIAL MASK MODE

0	1	0	1
0	0	1	1
No Action		Reset special mask	Set special mask

8259 PROGRAMMING

Normally in the exam you are asked to initialize a **single 8259** for some given specifications. **For doubts contact Bharat Sir on 98204 08217**
Remember that **in any case ICW1, ICW2 and ICW4 will be required.**
If the system is **cascaded** then **ICW3** is required.
For a **cascaded system** remember that **every 8259 has to be initialized,** i.e. the entire procedure has to be **repeated for each 8259.**
If **masking** is asked then **OCW1** is required.
If **rotating priority** is asked then **OCW2** is required.
Finally, if **SMM or Polling** is asked then **OCW3** is required.

Q 1) WAP to initialize Single 8259 as follows
Edge triggered,
Single,
Auto EOI Mode,
Buffered Mode,
Mask IR3, IR4, IR5, IR6,
Vector number of IR0 is 40H.
Assume 8259 is at Port Address 80H.

Soln:

```
     Code SEGMENT
          ASSUME CS: Code

   Start: MOV AL, 13H
          OUT  80H, AL        // ICW1 = 0001 0011 = 13H

          MOV  AL, 40H
          OUT  82H, AL        // ICW2 = 0100 0000 = 40H

          MOV  AL, 0BH
          OUT  82H, AL        // ICW4 = 0000 1011 = 0BH

          MOV  AL, 78H
          OUT  82H, AL        // OCW1 = 0111 1000 = 78H

          INT  03H

     Code ENDS

          END Start
```

Q 2) **WAP to initialize Cascaded 8259.**
One Master, two slaves connected on IR2 and IR3 of master.
Master: Port address 80H. Vector Number of IR6 is 46H. Edge triggered. AEOI Mode. SFNM. Keyboard Interrupt connected on IR4.
Slave2: Port address 84H. Vector Number of IR0 is 50H. Level triggered. Normal EOI Mode. Printer Interrupt on IR0. Card Reader Interrupt on IR1.
Slave3: Port address 90H. Vector Number of IR6 is 76H. Edge triggered. AEOI Mode. External Interrupts connected on IR0, IR1, IR2 and IR7.
For all the above 8259's, mask the unwanted interrupts.
Also show the decoding for the above circuit.

Soln: **For doubts contact Bharat Sir on 98204 08217**

```
Code   SEGMENT
       ASSUME CS: Code
Start: MOV  AL, 11H     // MASTER 8259
       OUT  80H, AL          // ICW1 = 0001 0001 = 11H
       MOV  AL, 40H
       OUT  82H, AL          // ICW2 = 0100 0000 = 40H
       MOV  AL, 0CH
       OUT  82H, AL          // ICW3 = 0000 1100 = 0CH
       MOV  AL, 1FH
       OUT  82H, AL          // ICW4 = 0001 1111 = 1FH
       MOV  AL, E3H
       OUT  82H, AL          // OCW1 = 1110 0011 = E3H

       MOV  AL, 19H     // SLAVE at IR2
       OUT  84H, AL          // ICW1 = 0001 1001 = 19H
       MOV  AL, 50H
       OUT  86H, AL          // ICW2 = 0101 0000 = 50H
       MOV  AL, 02H
       OUT  86H, AL          // ICW3 = 0000 0010 = 02H
       MOV  AL, 09H
       OUT  86H, AL          // ICW4 = 0000 1001 = 09H
       MOV  AL, FCH
       OUT  86H, AL          // OCW1 = 1111 1100 = FCH

       MOV  AL, 11H     // SLAVE at IR3
       OUT  90H, AL          // ICW1 = 0001 0001 = 11H
       MOV  AL, 70H
       OUT  92H, AL          // ICW2 = 0111 0000 = 70H
       MOV  AL, 03H
       OUT  92H, AL          // ICW3 = 0000 0011 = 03H
       MOV  AL, 0BH
       OUT  92H, AL          // ICW4 = 0000 1011 = 0BH
       MOV  AL, 78H
       OUT  92H, AL          // OCW1 = 0111 1000 = 78H

       INT 03H
Code   ENDS
END    Start
```

INTERFACING AND WORKING OF A "SINGLE" 8259

A single 8259 can accept 8 interrupts.
Whenever a device interrupts 8259, 8259 will interrupt the **µP on INTR pin.**
Hence, first the **INTR** signal of the µP **should be enabled** using the **STI** instruction**.**
8259 is **initialized** by giving **ICW1** and **ICW2** (compulsory) and **ICW4** (optional).
Note that **ICW3** is **not given** as Single 8259 is used. OCWs are given if required.

Once 8259 is **initialized**, the **following sequence** of events takes place when one or more **interrupts occur** on the IR lines of the 8259.

1) The **corresponding bit** for an interrupt is **set in IRR**.

2) The **Priority Resolver checks** the 3 registers:
   IRR (for highest interrupt request)
   IMR (for the masking Status)
   InSR (for the current level serviced)
   and **determines** the **highest priority** interrupt.
   It **sends** the **INT** signal **to** the **µP**.

3) The **µP finishes** the **current instruction** and **acknowledges** the interrupt by **sending** the **first** $\overline{\textbf{INTA}}$ **pulse**.

4) On receiving the first $\overline{\textbf{INTA}}$ signal, the **corresponding bit** in the **InSR** is **set** (indicating that now this interrupt is in service) and the **bit** in the **IRR** is **reset** (to indicate that the request is accepted).
   For doubts contact Bharat Sir on 98204 08217
   8259 now **prepares** to send the Vector number **N** to the µP on the data bus.

5) The **µP sends the second** $\overline{\textbf{INTA}}$ **pulse** to 8259.

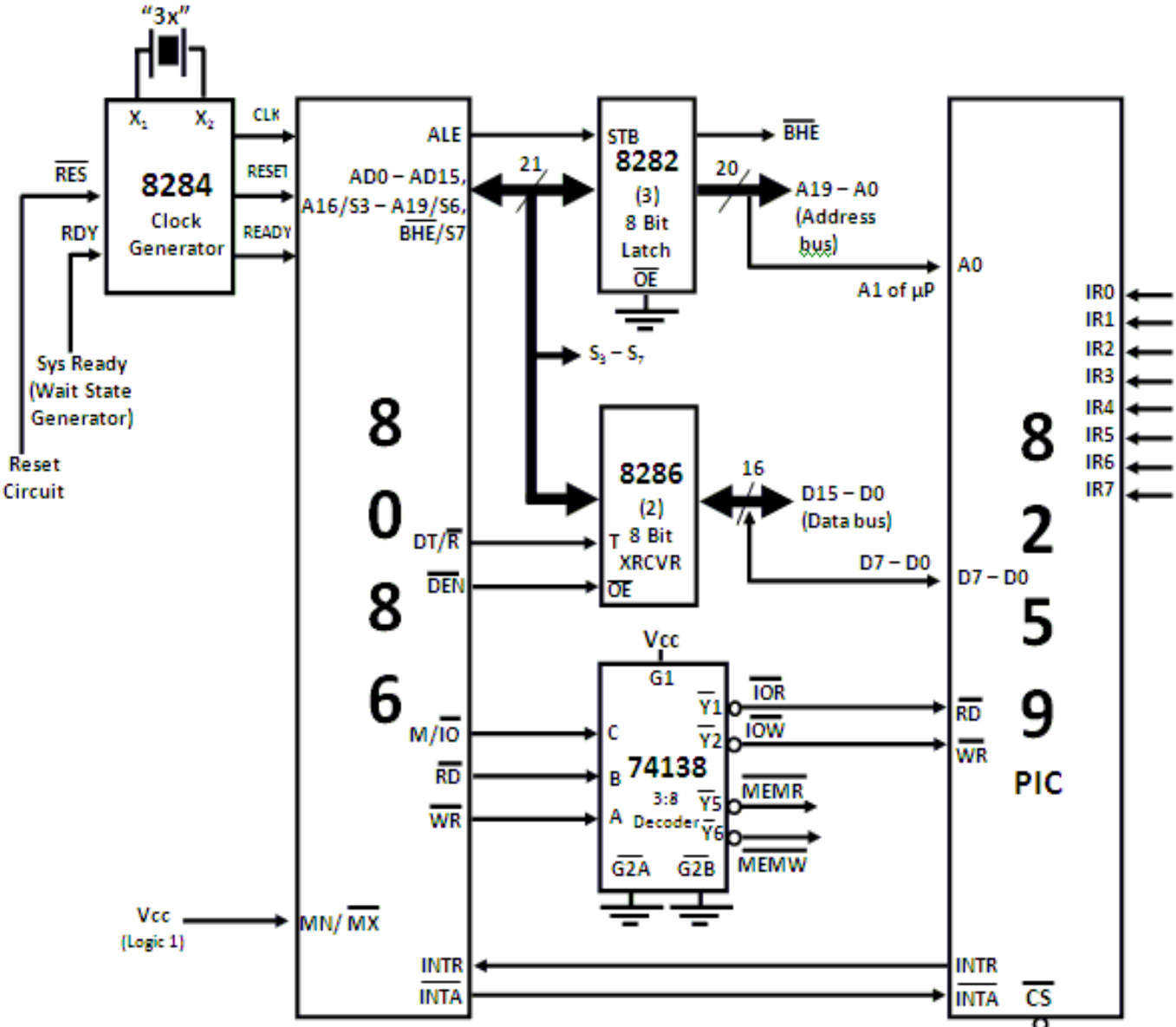
6) In response to the 2nd $\overline{\textbf{INTA}}$ pulse, **8259 sends the one byte Vector Number N to µP**.

7) Now the **µP multiplies N x 4**, to get the values of CS and IP **from the IVT**.

8) **In** the **AEOI Mode** the **InSR bit is reset** at this point, **otherwise** it **remains set until** an **appropriate EOI** command is given at the End of the ISR.

9) The **µP pushes** the contents of **Flag** Register, **CS**, **IP**, into the **Stack, Clears IF and TF** and **transfers program to** the **address** of the **ISR**.. *#Please refer Bharat Sir's Lecture Notes for this ...*

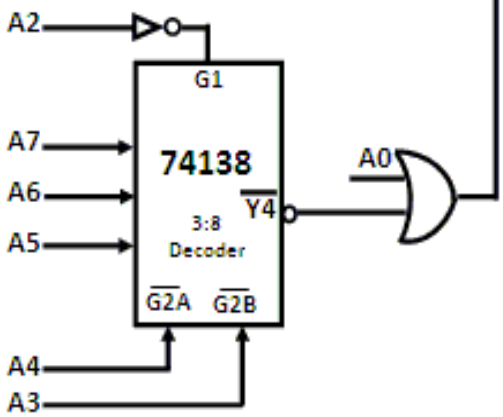
10) **The ISR thus begins**.



I/O Map of 8259 at 80H

	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
ICW1	1	0	0	0	0	0	0	0	80H
ICW2	1	0	0	0	0	0	1	0	82H

Chip Selection
Internal Selection
Bank Selection



Interfacing and Working of "CASCADED" 8259

When **more than one 8259s** are connected to the µP, it is called as a **Cascaded configuration**.
A Cascaded configuration **increases** the **number of interrupts** handled by the system.
As the **maximum** number of **8259s** interfaced can be **9** (1 Master and 8 Slaves) the **Maximum** number of **interrupts** handled can be **64**.

The **master 8259** has $\overline{\textbf{SP}}$ **/** $\overline{\textbf{EN}}$ **= +5V** and the **slave** has $\overline{\textbf{SP}}$ **/** $\overline{\textbf{EN}}$ **= 0V**.

Each slave's INT output is **connected** to the **IR input** of the **Master**.
The **INT** output **of** the **Master** is **connected** to the **INTR** input of the **µP**.
The **master addresses** the individual **slaves through** the **CAS_2, CAS_1, CAS_0 lines** connected from the master to each of the slaves.
First the **INTR** signal of the µP **should be enabled** using the **STI** instruction**.**
Each 8259 (Master or Slave) **has** its **own address** and **has to be initialized separately** by giving ICWs as per requirement.

When an **interrupt request** occurs **on** a **SLAVE**, the events are performed:

1) The **slave 8259 resolves** the **priority** of the interrupt and **sends** the **interrupt to** the **master** 8259.

2) The **master resolves** the **priority** among its slaves and **sends** the **interrupt to** the **µP**.

3) The **µP finishes the current instruction** and **responds** to the interrupt **by sending 2** $\overline{\textbf{INTA}}$ **pulses**.

4) **In response to the first** $\overline{\textbf{INTA}}$ **pulse** the **following events occur**:

   i. The master **sends** the **3-bit salve identification number** on the **CAS lines**.

   ii. The **Master sets** the **corresponding bit** in **its InSR**.

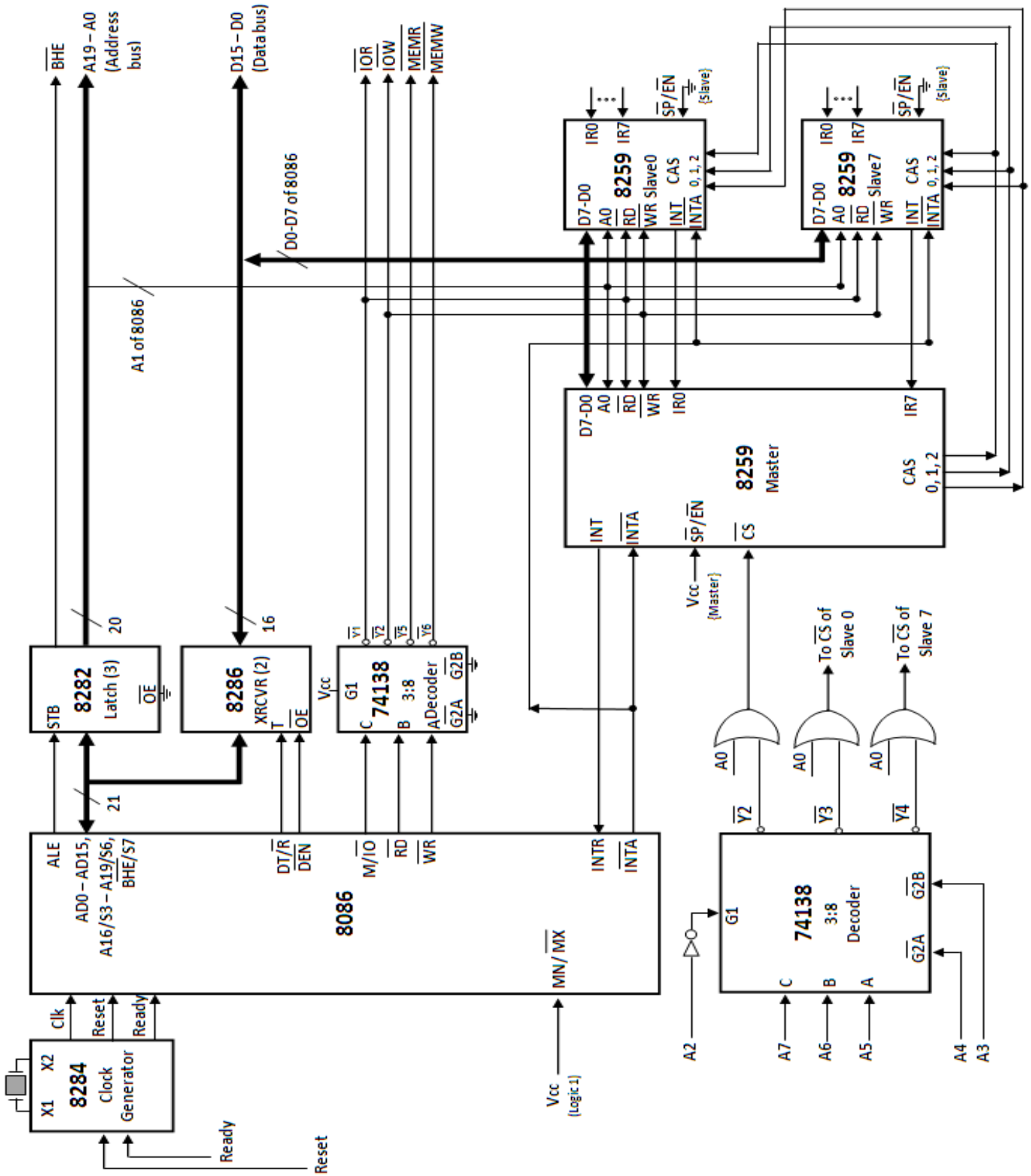
   iii. The **Slave identifies** its number on the **CAS lines** and **sets** the **corresponding bit in its InSR**.

5) In **response to** the **second** $\overline{\textbf{INTA}}$ pulse the **slave** places **Vector Number N** on the data bus.

6) **During** the **2nd** $\overline{\textbf{INTA}}$ pulse the **InSR bit** of the **slave** is **cleared** in **AEOI mode**, **otherwise** it is **cleared by** the **EOI command** at the end of the ISR.

7) The **µP pushes** the contents of **Flag** Register, **CS**, **IP**, into the **Stack, Clears IF and TF** and **transfers program to** the **address** of the **ISR**.. *#Please refer Bharat Sir's Lecture Notes for this ...*

8) **The ISR thus begins**.


BHE
A19 – A0 (Address bus)
D15 – D0 (Data bus)
IOR
IOW
MEMR
MEMW
D0-D7 of 8086
A1 of 8086
20
16
21
8282 Latch (3)
8286 XRCVR (2)
74138 3:8 Decoder
8086
8284 Clock Generator
8259 Slave0
8259 Slave7
8259 Master
ALE
AD0 – AD15, A16/S3 – A19/S6, BHE/S7
DT/R
DEN
M/IO
RD
WR
INTR
INTA
MN/MX
Clk
Reset
Ready
X1
X2
SP/EN
{Slave}
{Master}
Vcc
(Logic 1)
To CS of Slave 0
To CS of Slave 7
A0
A2
A3
A4
A5
A6
A7

I/O map		A7	A6	A5	A4	A3	A2	A1	A0	I/O
8259	ICW1	**0**	**1**	**0**	0	0	0	**0**	0	**40**
Master	ICW2	**0**	**1**	**0**	0	0	0	**1**	0	**42**
8259	ICW1	**0**	**1**	**1**	0	0	0	**0**	0	**60**
Slave 0	ICW2	**0**	**1**	**1**	0	0	0	**1**	0	**62**
8259	ICW1	**1**	**0**	**0**	0	0	0	**0**	0	**80**
Slave 7	ICW2	**1**	**0**	**0**	0	0	0	**1**	0	**82**

Programmable Peripheral Interface

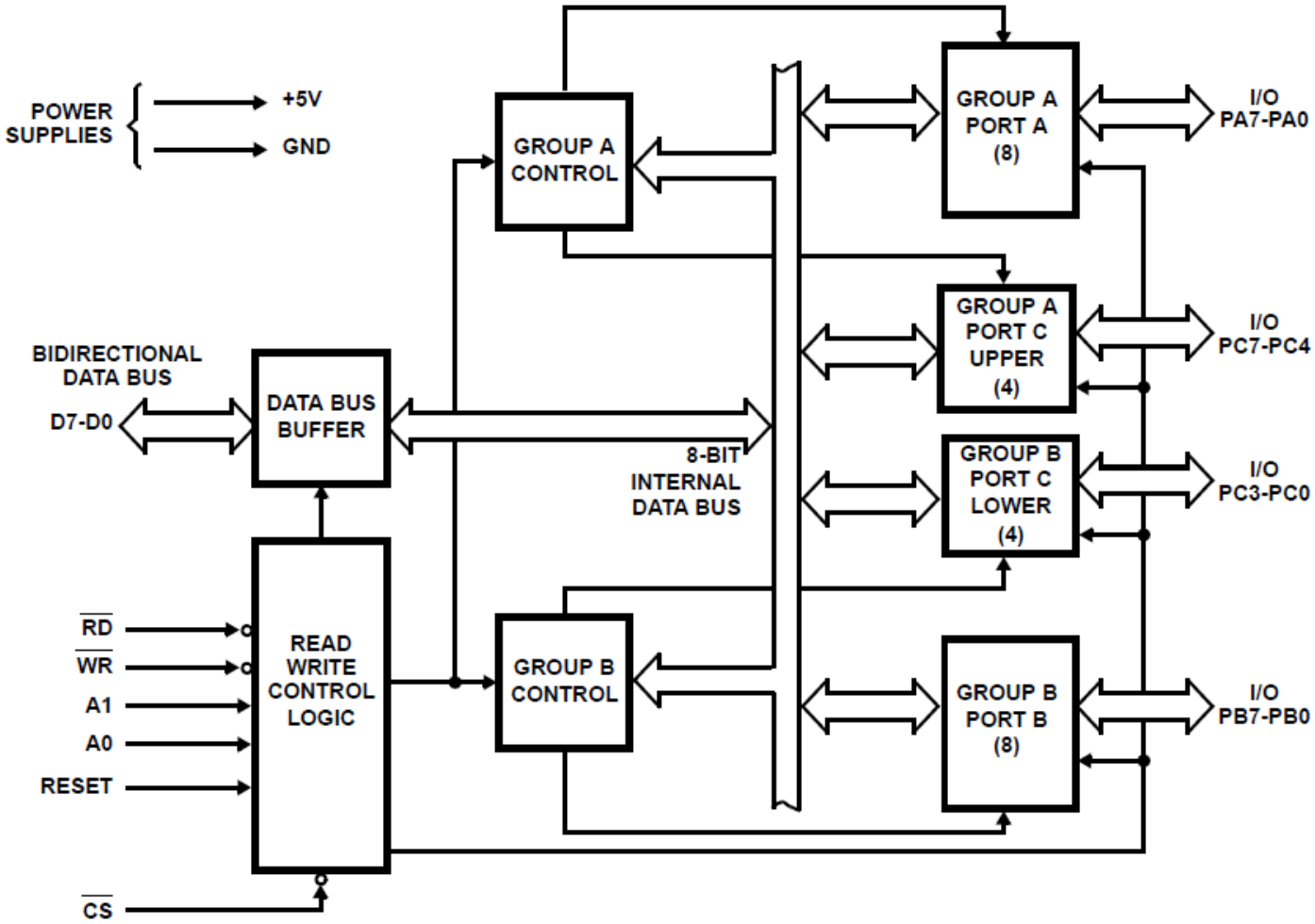
8255 | PPI

WWW.BHARATACHARYAEDUCATION.COM

Salient Features

1) It is a **programmable** general-purpose **I/O** device.
2) It has 3 8-bit bi-directional I/O ports: Port A, Port B, and Port C.
3) It provides 3 modes of data transfer: Simple I/O, Handshake I/O and Bi-directional Handshake.
4) Additionally it also provides a Bit Set Reset Modes to alter individual bits of Port C.

ARCHITECTURE OF 8255



The architecture of 8255 can be divided into the following parts:

1) Data Bus Buffer

This is a 8-bit bi-directional buffer used to interface the internal data bus of 8255 with the external (system) data bus.
The CPU transfers data to and from the 8255 through this buffer.

2) Read/Write Control Logic

It accepts address and control signals from the µP.
The Control signals determine whether it is a read or a write operation and also select or reset the 8255 chip. **For doubts contact Bharat Sir on 98204 08217**
The Address bits (A_1, A_0) are used to select the Ports or the Control Word Register as shown:

For 8255 A_1 A_0	For 8086 A_2 A_1	Selection	Sample address
0 0	0 0	Port A	80 H (i.e. 1000 0**00**0)
0 1	0 1	Port B	82 H (i.e. 1000 0**01**0)
1 0	1 0	Port C	84 H (i.e. 1000 0**10**0)
1 1	1 1	Control Word	86 H (i.e. 1000 0**11**0)

The Ports are controlled by their respective Group Control Registers.

3) Group A Control

This Control block controls Port A and Port C_{Upper} i.e. PC_7-PC_4.
It accepts Control signals from the Control Word and forwards them to the respective Ports.

4) Group B Control

This Control block controls Port B and Port C_{Lower} i.e. PC_3-PC_0.
It accepts Control signals from the Control Word and forwards them to the respective Ports.

5) Port A, Port B, Port C

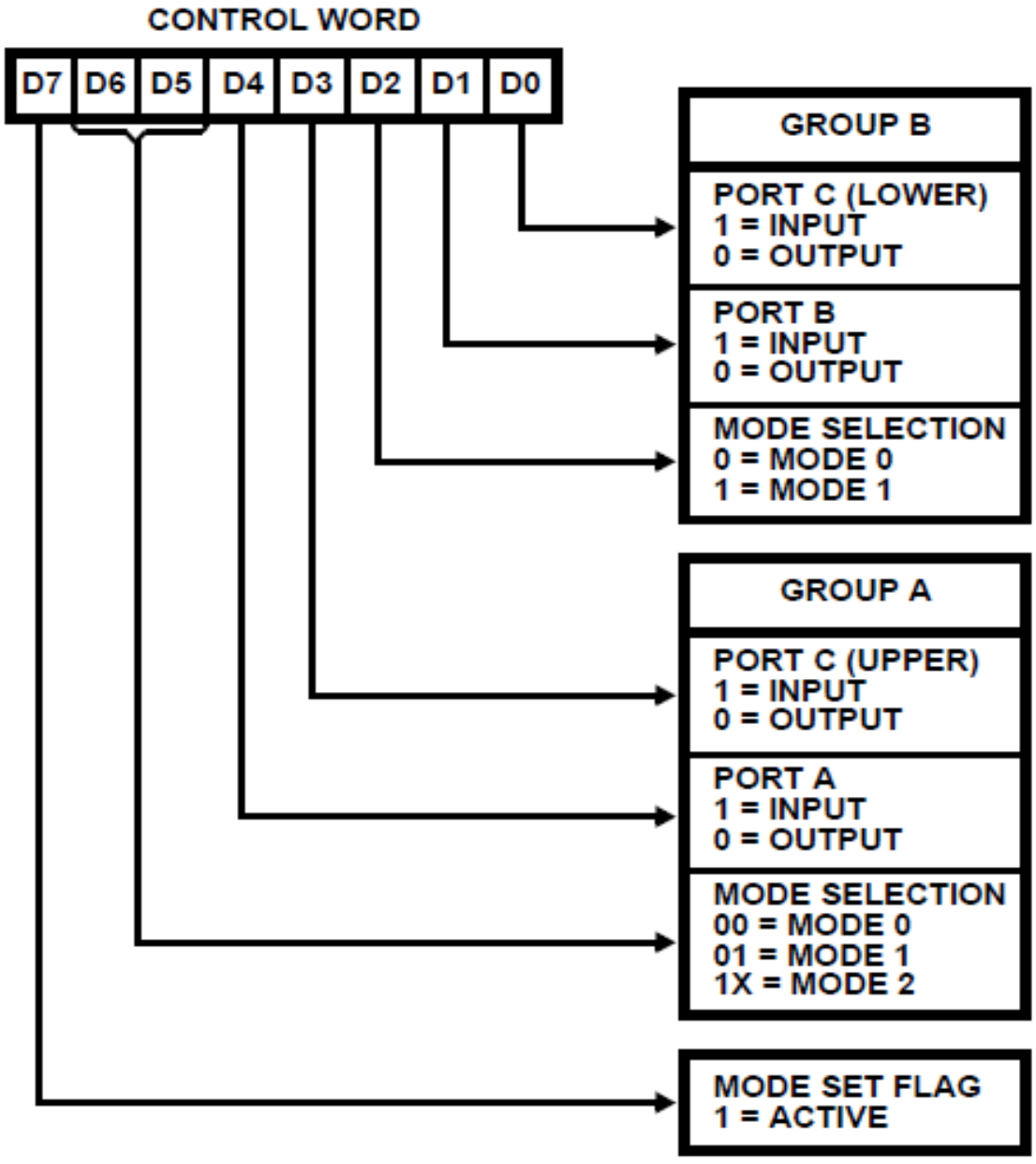
These are 8-bit Bi-directional Ports.
They can be programmed to wok in the various modes as follows:

Port	Mode 0	Mode 1	Mode 2
Port A	Yes	Yes	Yes
Port B	Yes	Yes	**No** *(Mode 0 or Mode 1)*
Port C	Yes	**No** *(Handshake signals)*	**No** *(Handshake signals)*

ONLY Port C can also be programmed to work in Bit Set reset Mode to manipulate its individual bits.

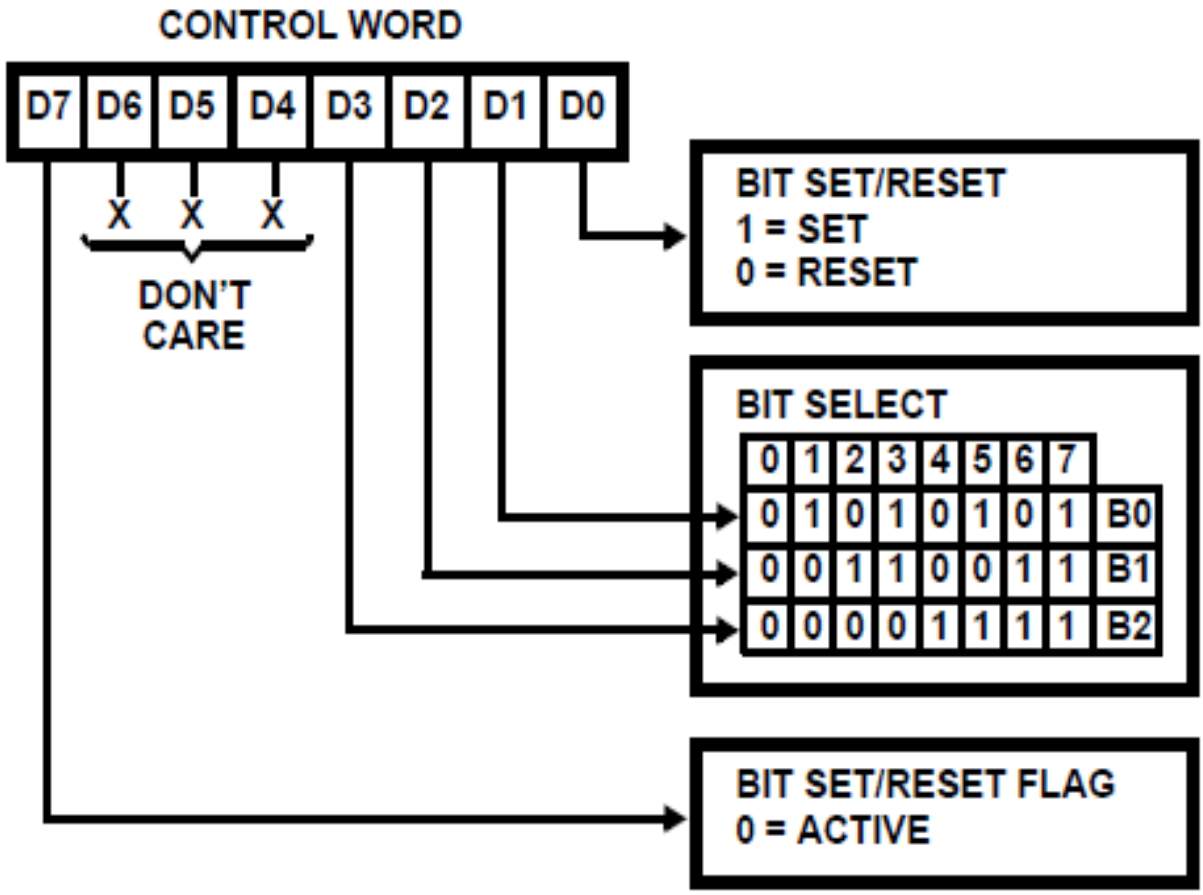
1)Control Word of 8255 - I/O Mode (I/O Command)

To do 8-bit data transfer using the Ports A, B or C, 8255 needs to be in the IO mode. The bit pattern for the control word in the IO mode is as follows:



2)Control Word of 8255 - BSR Mode (BSR Command) { ONLY for Port C}

- The BSR Mode is used **ONLY for Port C.**
- In this Mode the **individual bits** of Port C can be **set or reset**.
- This is very useful as it provides **8 individually controllable lines** which can be used while interfacing with devices like an **A to D Converter** or a 7-segment display etc.
- The individual bit is **selected** and Set/reset through the **control word**.
- Since the D7 bit of the Control Word is 0, the BSR operation **will not affect the I/O operations** of 8255. **For doubts contact Bharat Sir on 98204 08217**



DATA TRANSFER MODES OF 8255

❖ Mode 0 (Simple Bi-directional I/O)

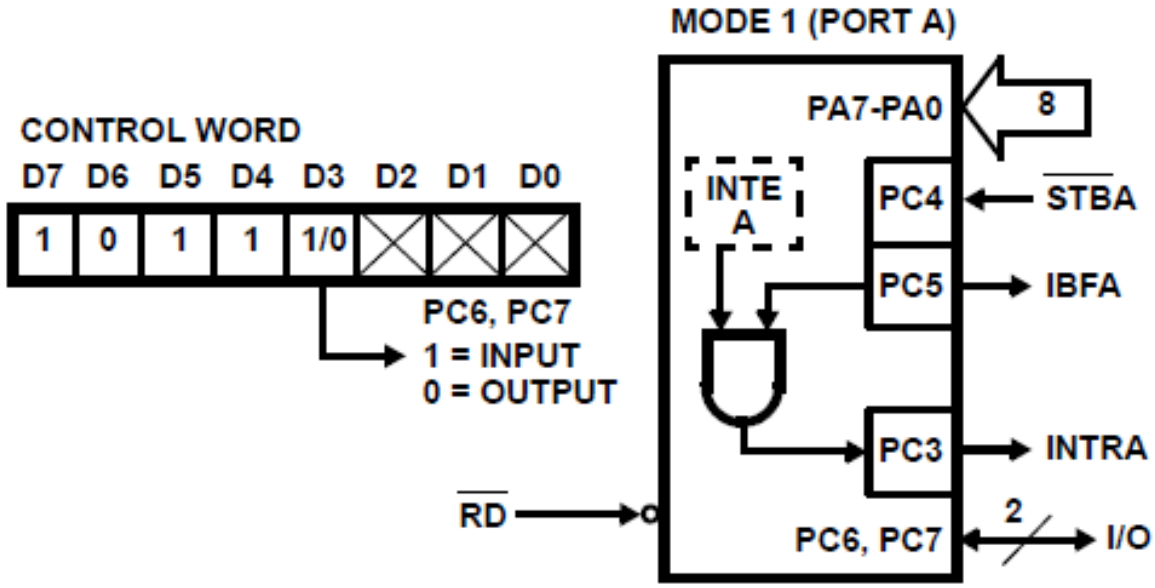
Port A and Port B used as 2 Simple 8-bit I/O Ports.
Port C is used as 2 simple 4-bit I/O Ports.
Each port can be programmed as input or output individually.
Ports do not have handshake or interrupting capability.
Hence, **slower** devices cannot be interfaced.

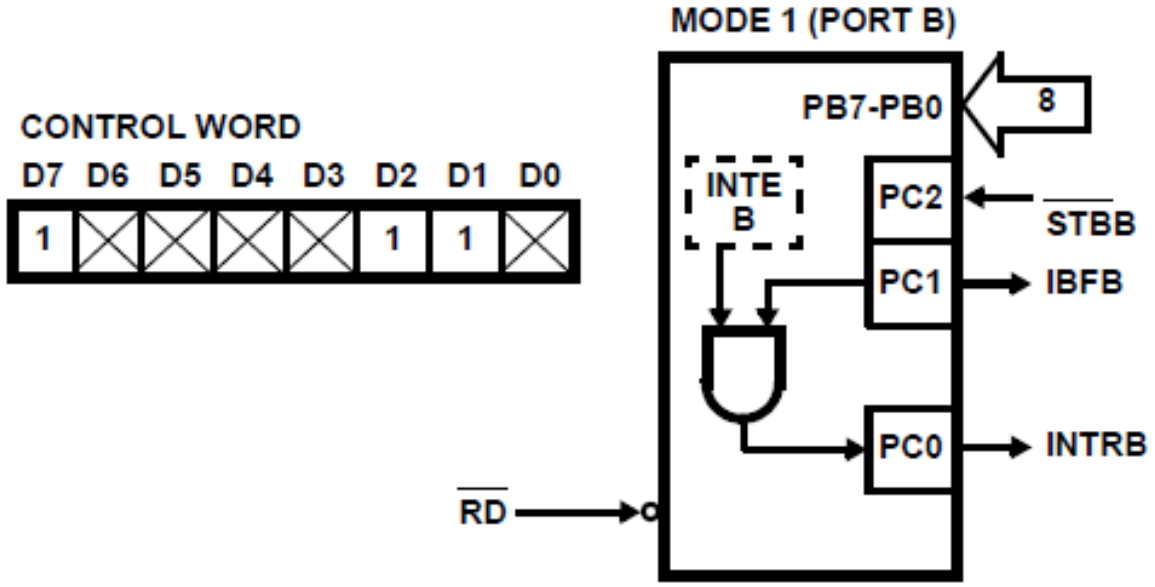
❖ Mode 1 (Handshake I/O)

In Mode 1, handshake signals are exchanged between the devices before the data transfer takes place.
Port A and Port B used as 2 8-bit I/O Ports that can programmed in Input OR in output mode.
Each Port uses 3 lines from Port C for handshake. The remaining lines of Port C can be used for simple IO.
Interrupt driven data transfer and **Status driven** data transfer possible.
Hence, **slower** devices can be interfaced.

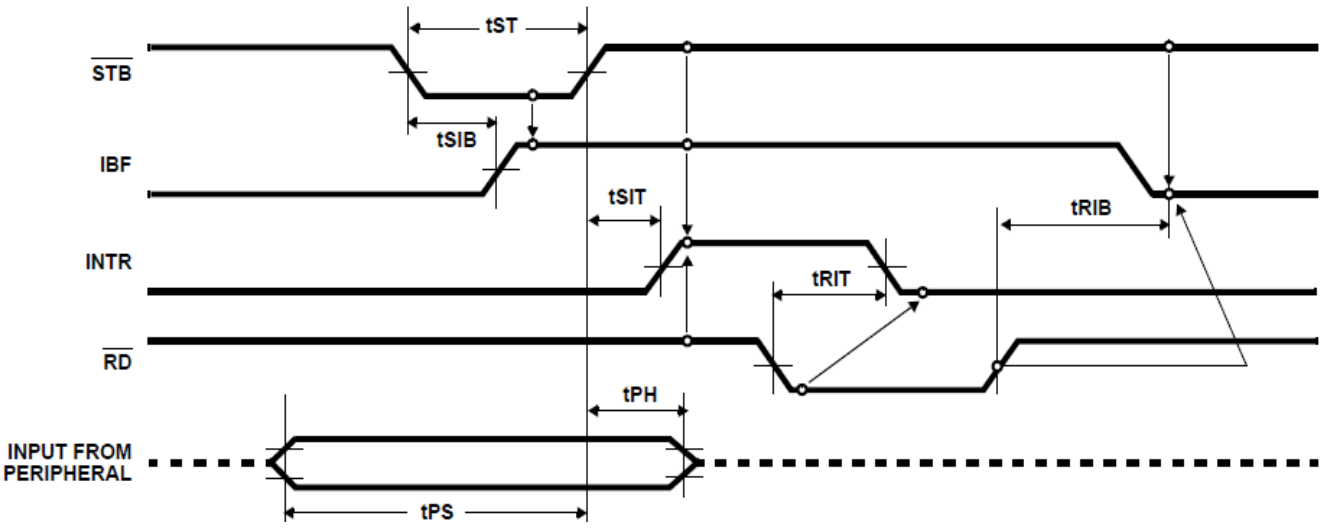
The handshake signals are different for input and output modes.
#Please refer Bharat Sir's Lecture Notes for this ...

♦ Mode 1 (Input Handshaking)





Timing Diagram for Mode 1 Input Transfer



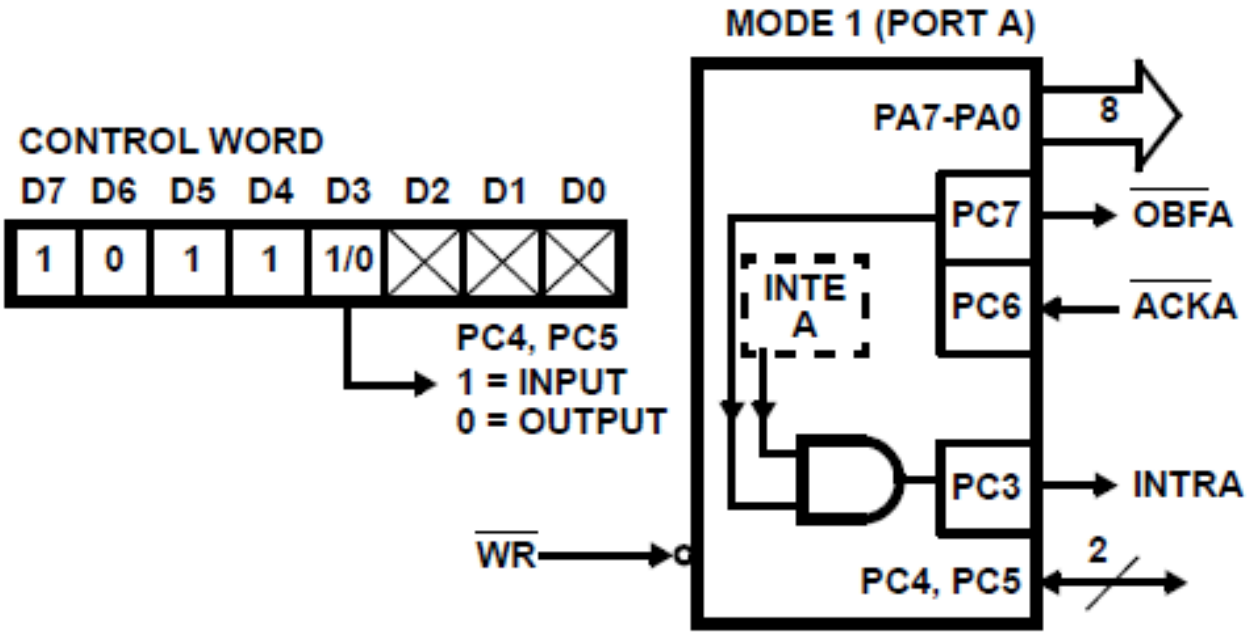
Working:

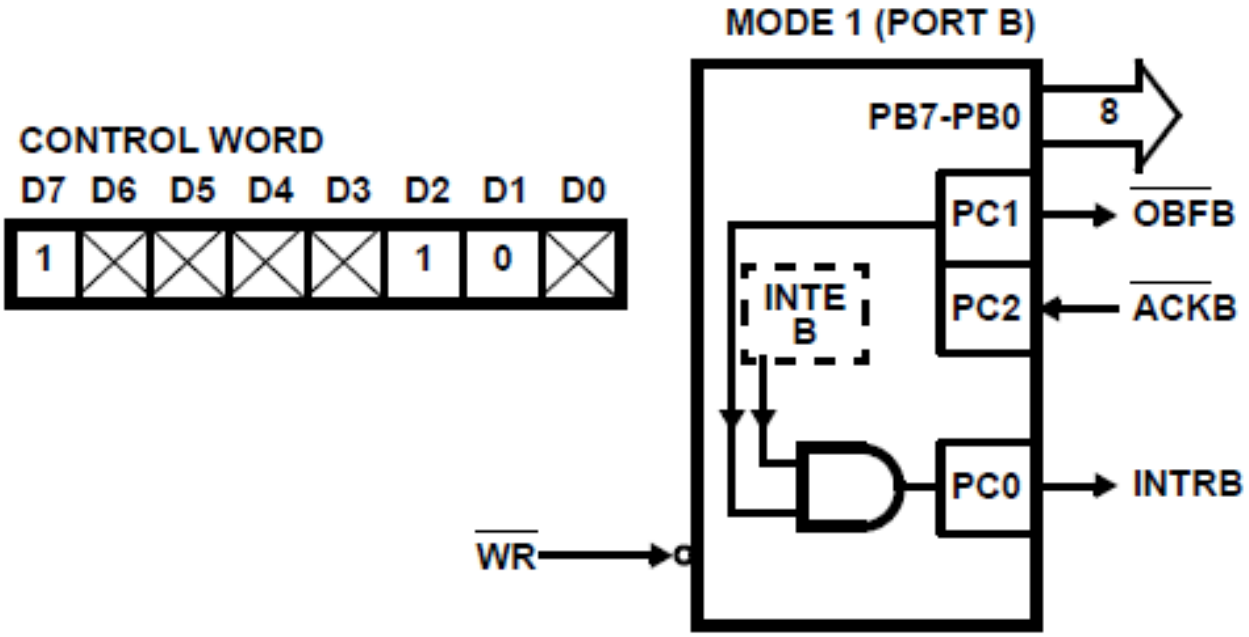
Each port uses **3 lines** of **Port C** for the following signals:
STB (Strobe), **IBF** (Input Buffer Full) ➔ Handhsake signals
INTR (interrupt) ➔ Interrupt signal
Additionally the $\overline{RD}$ signal of 8255 is also used.

Handshaking takes place in the following manner:

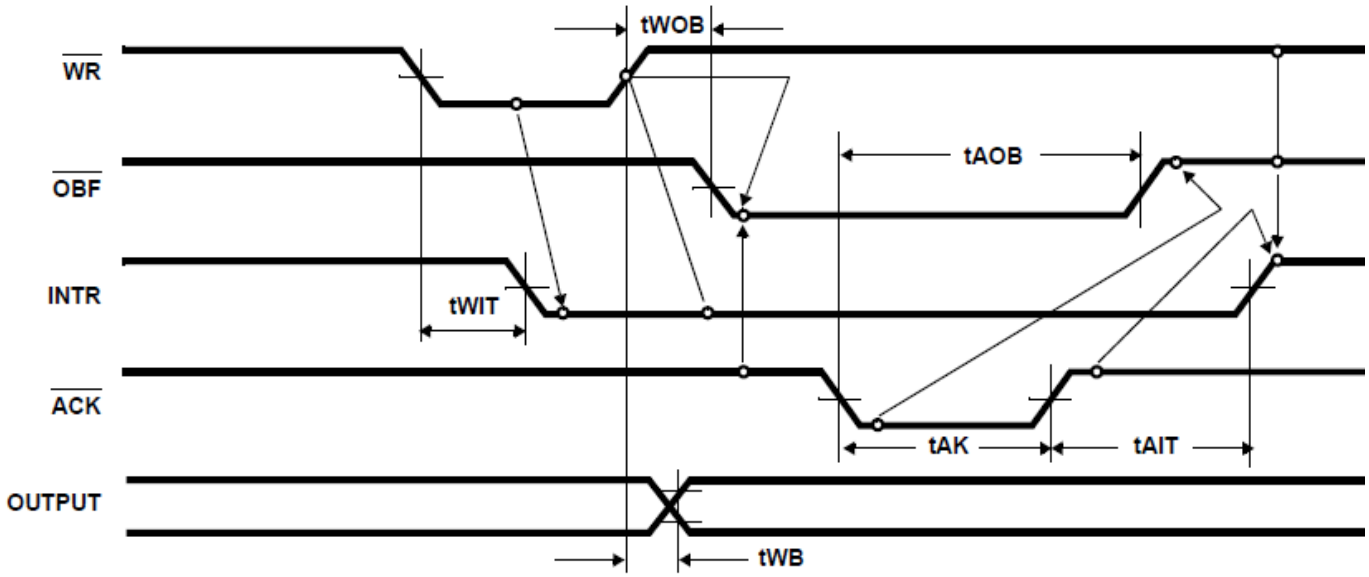
1) The **peripheral** device **places data** on the Port **bus** and informs the Port by **making STB low**.
2) The **input Port accepts** the **data** and informs the peripheral to wait by making **IBF high**. This **prevents** the peripheral from **sending more data** to the 8255 and **hence data loss** is prevented. ☺ In case of doubts, contact Bharat Sir: - 98204 08217.
3) **8255 interrupts** the **µP** through the **INTR** line provided the INTE flip-flop is set.
4) **In response** to the Interrupt, the **µP issues** the $\overline{RD}$ signal and **reads** the **data**. The **data byte** is **thus transferred** to the **µP**.
5) Now, the **IBF** signal **goes low** and the peripheral can **send more data** in the above sequence.

♦ Mode 1 (Output Handshaking)





Timing Diagram for Mode 1 Output Transfer



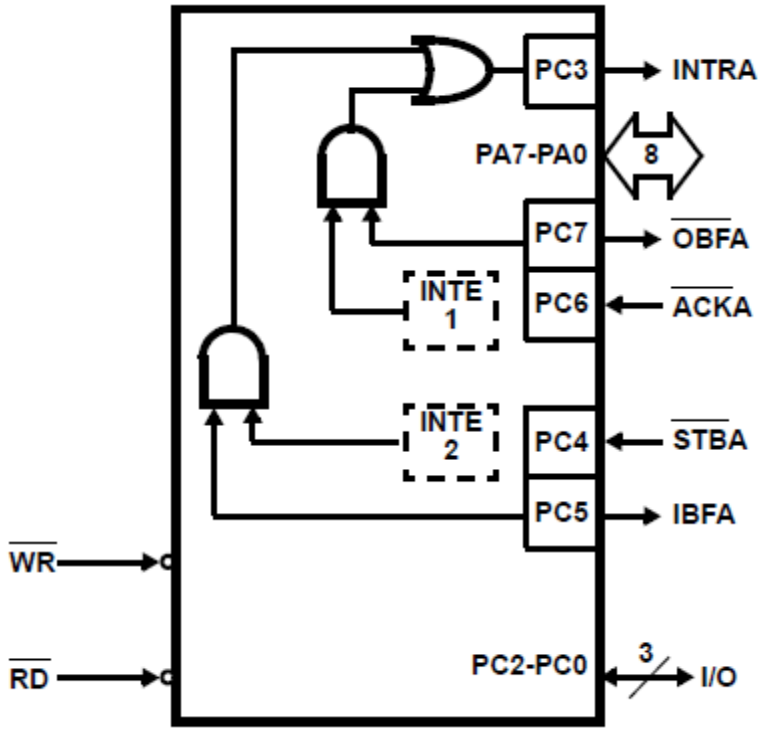
Working

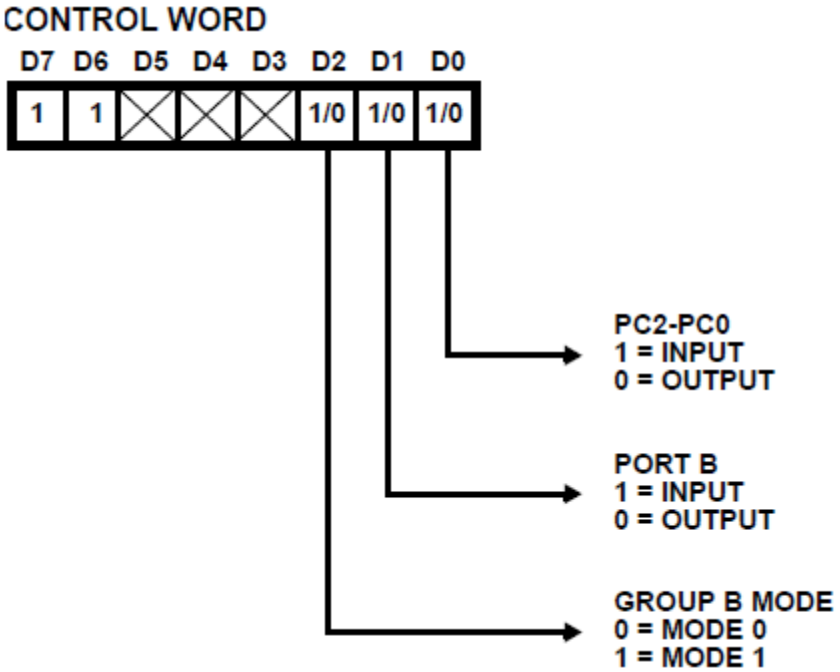
Each port uses **3 lines** of **Port C** for the following signals:
$\overline{\textbf{OBF}}$ (Output Buffer Full), $\overline{\textbf{ACK}}$ (Acknowledgement) ➔ Handhsake signals

INTR (interrupt) ➔ Interrupt signal. Additionally the $\overline{\textbf{WR}}$ signal of 8255 is also used. **Handshaking** takes place in the following manner:

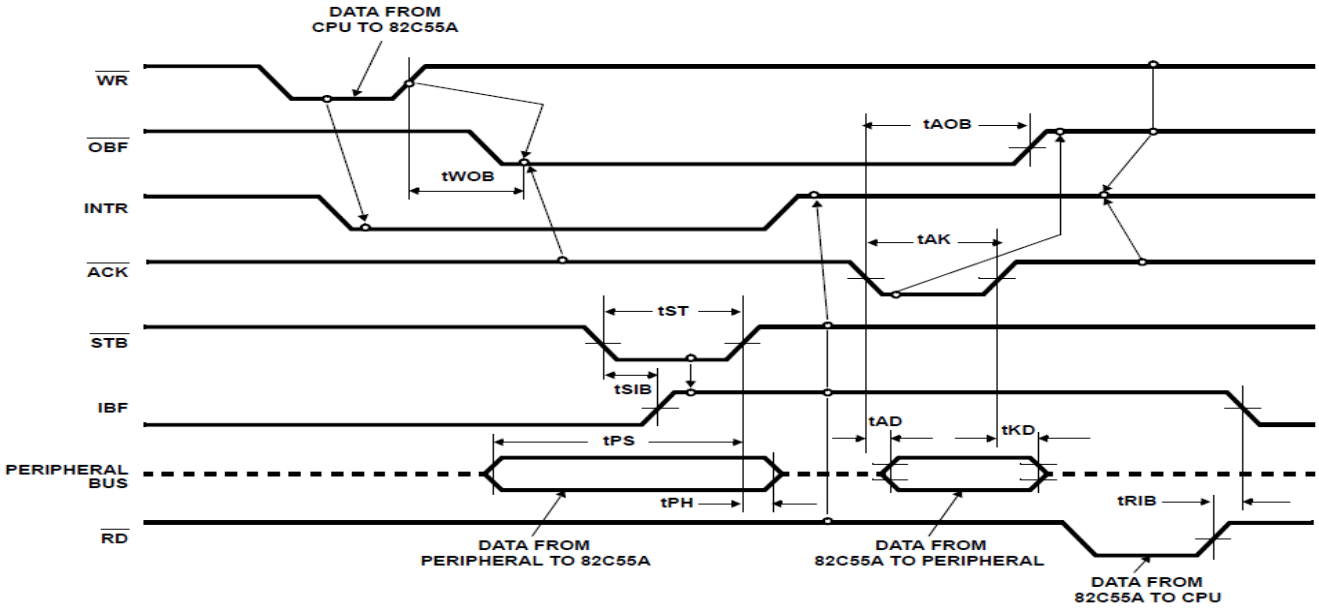
1) When the output port is empty (indicated by a high on the INTR line), the **µP writes data** on the output port by giving the $\overline{\textbf{WR}}$ signal.
2) As soon as the $\overline{\textbf{WR}}$ operation is complete, the **8255 makes the INTR low**, indicating that the µP should **wait**. This **prevents** the µP from **sending more data** to the 8255 and **hence data loss** is prevented.
3) **8255** also **makes** the $\overline{\textbf{OBF}}$ **low** to indicate to the output peripheral that **data** is **available** on the data bus.
4) The **peripheral accepts** the **data** and sends an acknowledgement by making the $\overline{\textbf{ACK}}$ **low**. The **data byte** is **thus transferred** to the peripheral.
5) Now, the $\overline{\textbf{OBF}}$ and $\overline{\textbf{ACK}}$ lines **go high**.
6) The **INTR** line **becomes high** to **inform** the **µP** that **another byte** can be **sent.** i.e. the output port is empty.
   This process is repeated for further bytes.

❖ Mode 2 (Bi-directional Handshake I/O)





Timing Diagram for Mode 2 Bi-Directional Transfer



Working:

In this mode, **Port A** is **used** as an **8-bit bi-directional Handshake I/O Port**.
Port A requires **5 signals from Port C** for doing Bi-directional handshake.
Port B has the following **options**:

1) **Use** the **remaining 3 lines** of **Port C** for handshaking so that **Port B is in Mode 1**. Here **Port C** lines will be **completely used for handshaking** (5 by Port A and 3 by Port B).
   OR
2) **Port B** works in **Mode 0** as simple I/O.
   In this case the **remaining 3 lines** of **Port C** can be used for **data transfer**.

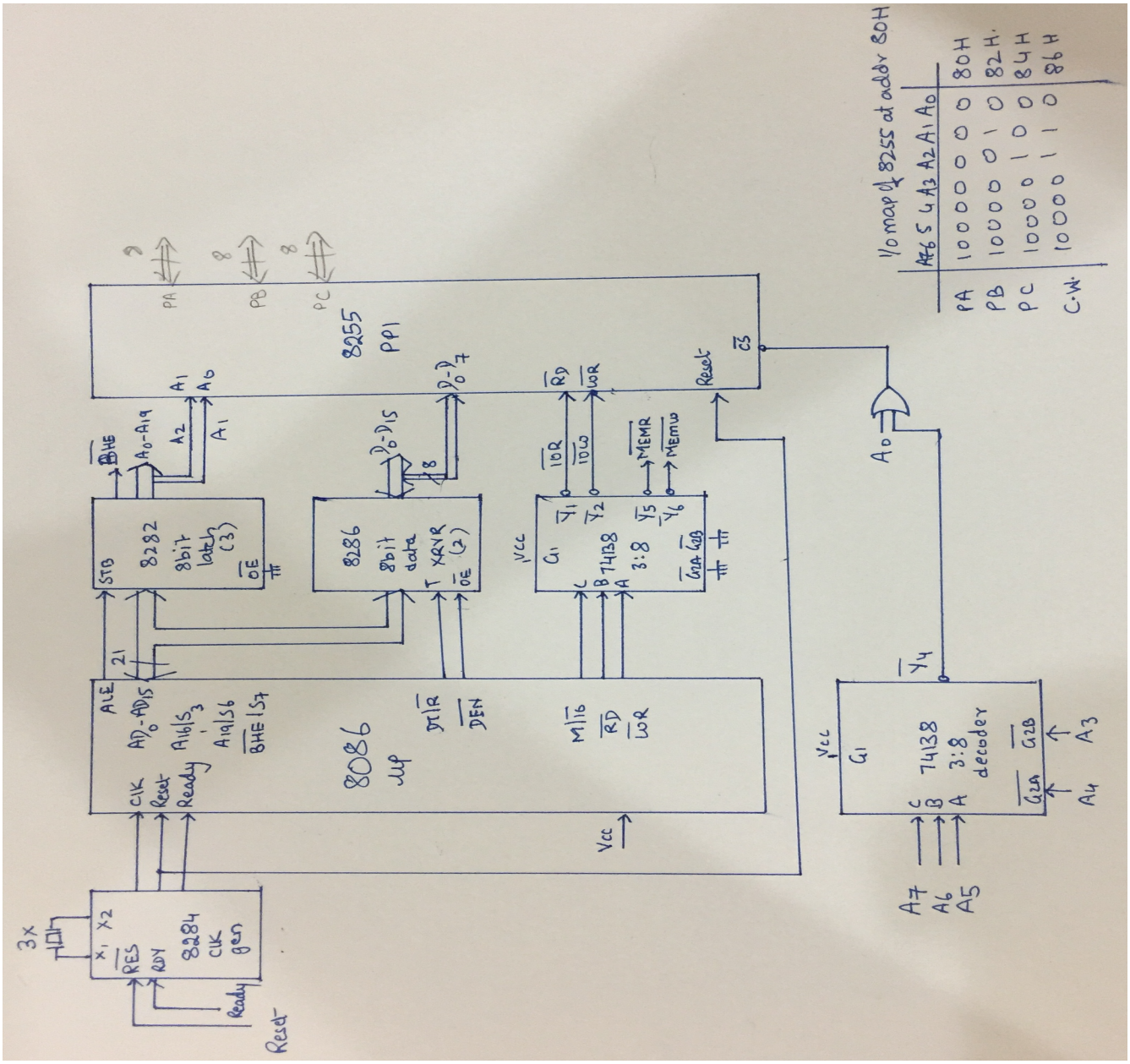
Port A can be used for data transfer between two computers as shown.
The high-speed computer is known as the master and the dedicated computer is known as the slave.
Handshaking process is similar to Mode 1.

For **Input**:
STB and **IBF** ➔ handshaking signals, **INTR** ➔ Interrupt signal.
For **Output**:
$\overline{\textbf{OBF}}$ and $\overline{\textbf{ACK}}$ ➔ handshaking signals, **INTR** ➔ Interrupt signal.

Thus the 5 signals used from Port C are:
STB, IBF, INTR, $\overline{\textbf{OBF}}$ and $\overline{\textbf{ACK}}$.

INTERFACING OF 8255 WITH 8086

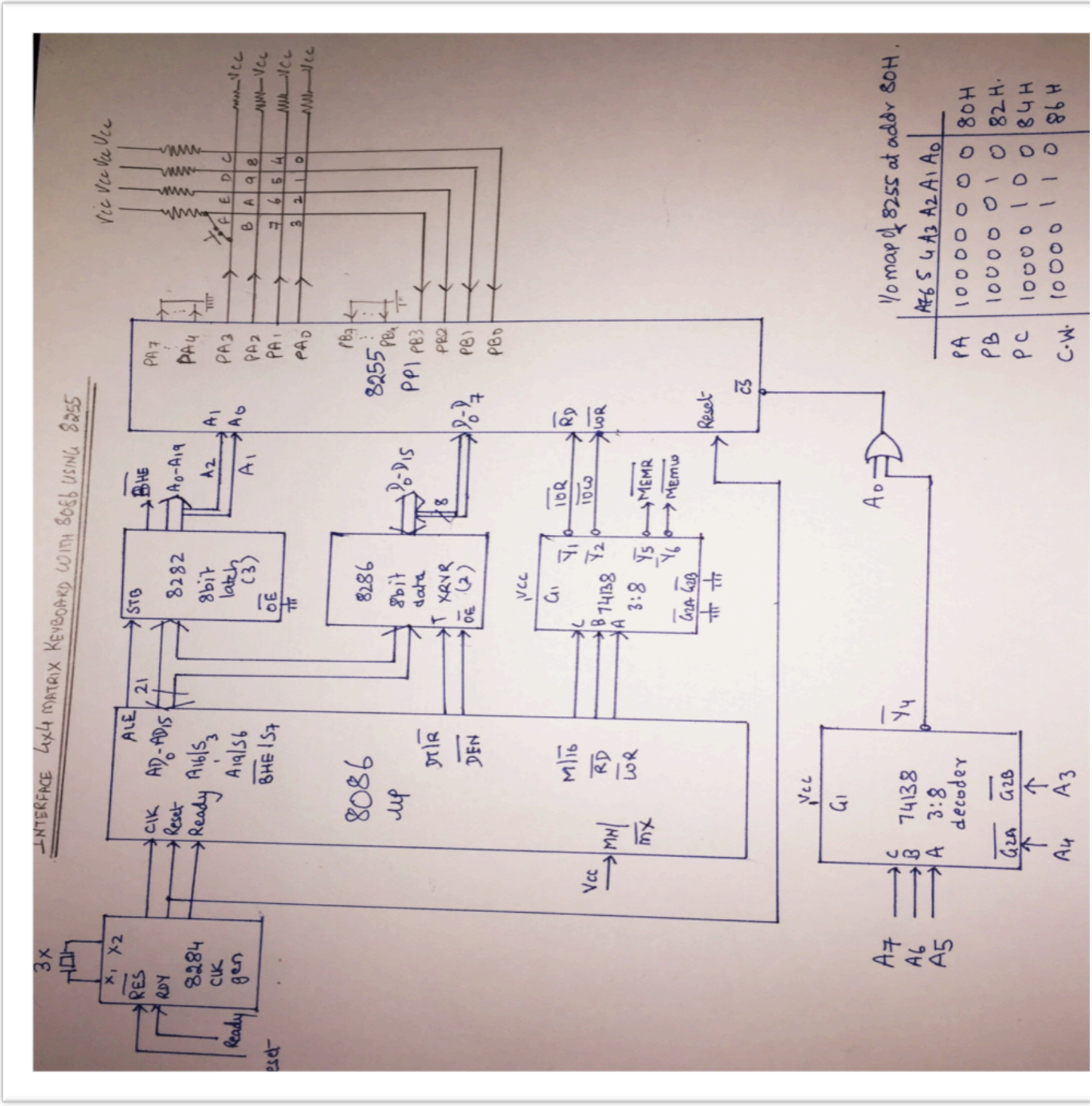
1) 8255 is a **programmable peripheral interface**.
   It is used to interface microprocessor with I/O devices via three ports: PA, PB, PC.
   All ports are 8-bit and bidirectional.
2) 8255 transfers data with the microprocessor through its **8-bit data bus**.
3) The **two address lines** A1 and A0 are used to make **internal selection** in 8255.
   They can have 4 options, selecting PA, PB, PC or the control word.
   The ports are selected to transfer data.
   The Control word is selected to send commands.
4) **Two commands** can be sent to 8255, called the I/O command and the BSR command.
   I/O command is used to initialize the **mode and direction** of the ports.
   BSR command is used to **set or reset a single line** of Port C.
5) 8255 has **three operational modes** of data transfer.
6) **Mode 0** is a **simple data transfer** mode.
   It does not perform handshaking but all three ports are available for data transfer.
7) **Mode 1** performs **unidirectional handshaking**.
   That makes transfers more reliable.
   Port C lines are used by Port A and Port B to perform Handshaking.
8) **Mode 2** performs **bidirectional handshaking**.
   Only Port A can operate in Mode 2.
   At that time Port B can operate in Mode 1 or Mode 0.
   Port C lines are again used up for performing Handshaking for Port A and Port B.


I/o map of 8255 at addr 80H
A7 A6 A5 A4 A3 A2 A1 A0
PA 1 0 0 0 0 0 0 0 80H
PB 1 0 0 0 0 0 1 0 82H
PC 1 0 0 0 0 1 0 0 84H
C.W. 1 0 0 0 0 1 1 0 86H
8255 PPI
PA
PB
PC
8086 up
8282 8bit latch (3)
8286 8bit data XCVR (2)
74138 3:8 decoder
8284 CLK gen
ALE
CLK
Reset
Ready
RES
RDY
3x
Vcc
A7
A6
A5
A4
A3

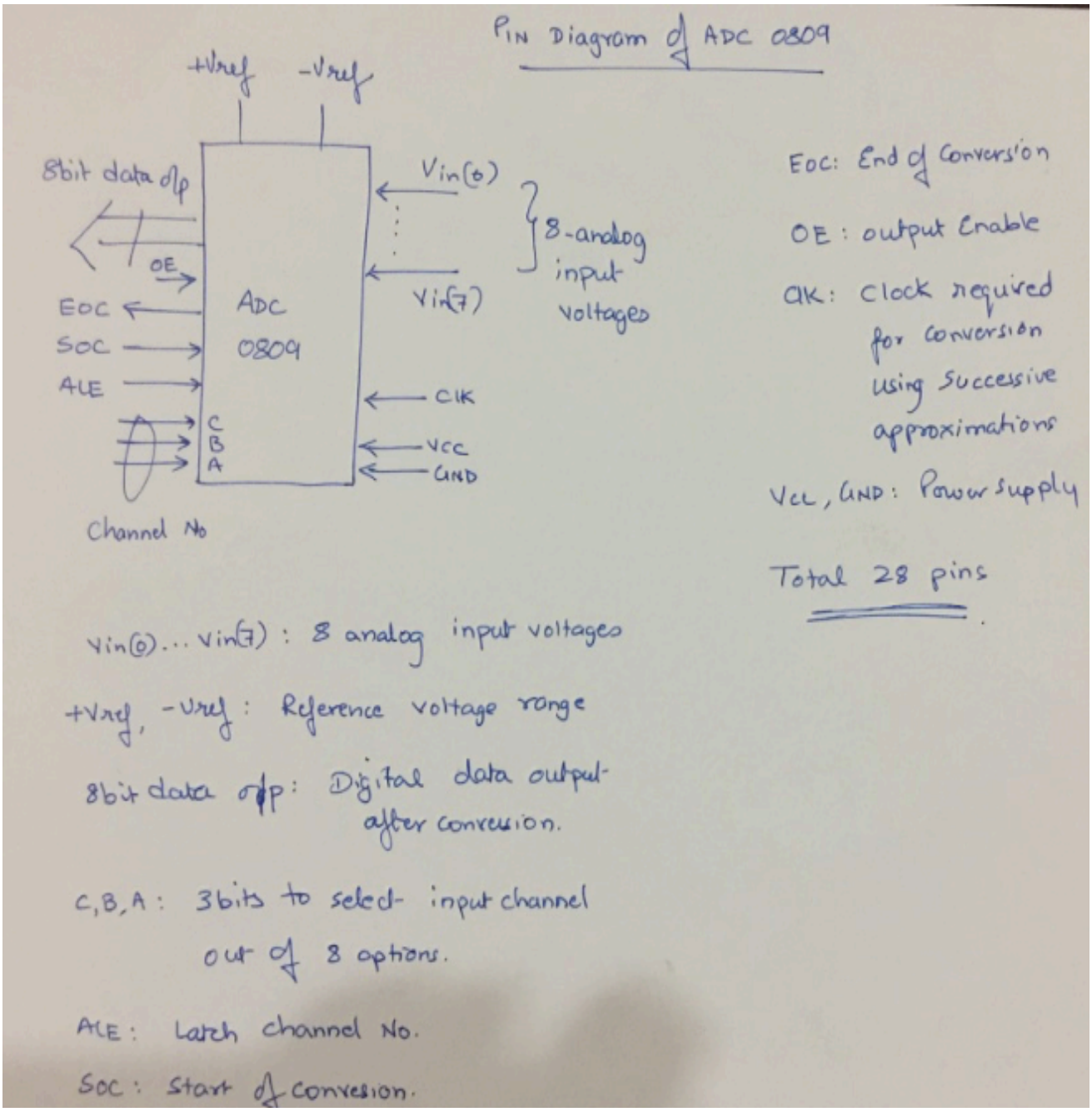
4 x 4 MATRIX KEYBOARD

Interface a 4 x 4 Matrix Keyboard to 8086 using 8255

1) A Matrix Keyboard is formed by a combination of **rows and columns**.
2) The advantage of a matrix keyboard is that we can interface **more keys** using **less lines**.
3) A 4x4 Matrix Keyboard uses **4 lines as rows and 4 as columns**.
4) This provides 16 intersecting points to connect **16 keys**.
5) The **rows are used as outputs** and **columns as inputs**.
6) 4 lines of **Port A** are used as **Rows** and 4 lines of **Port B** as **Columns**.
7) Keys are connected in such a way that **only when a key is pressed**, the corresponding row and column will **get connected**.
8) The **columns** are connected to **Vcc** via a pull up resister (~10k ohms).
9) The columns by default contain **logic "1"**.
10) Firstly, to know, **whether a key is pressed, we output "0" on all rows**.
11) If columns still contain all "1"s, then **no key is pressed**.
12) If any column contains a "0", then **some key has been pressed**.
13) Now we **singularly output a zero on each row** and read the columns again.
14) This is how we identify the row, the column and hence the key.

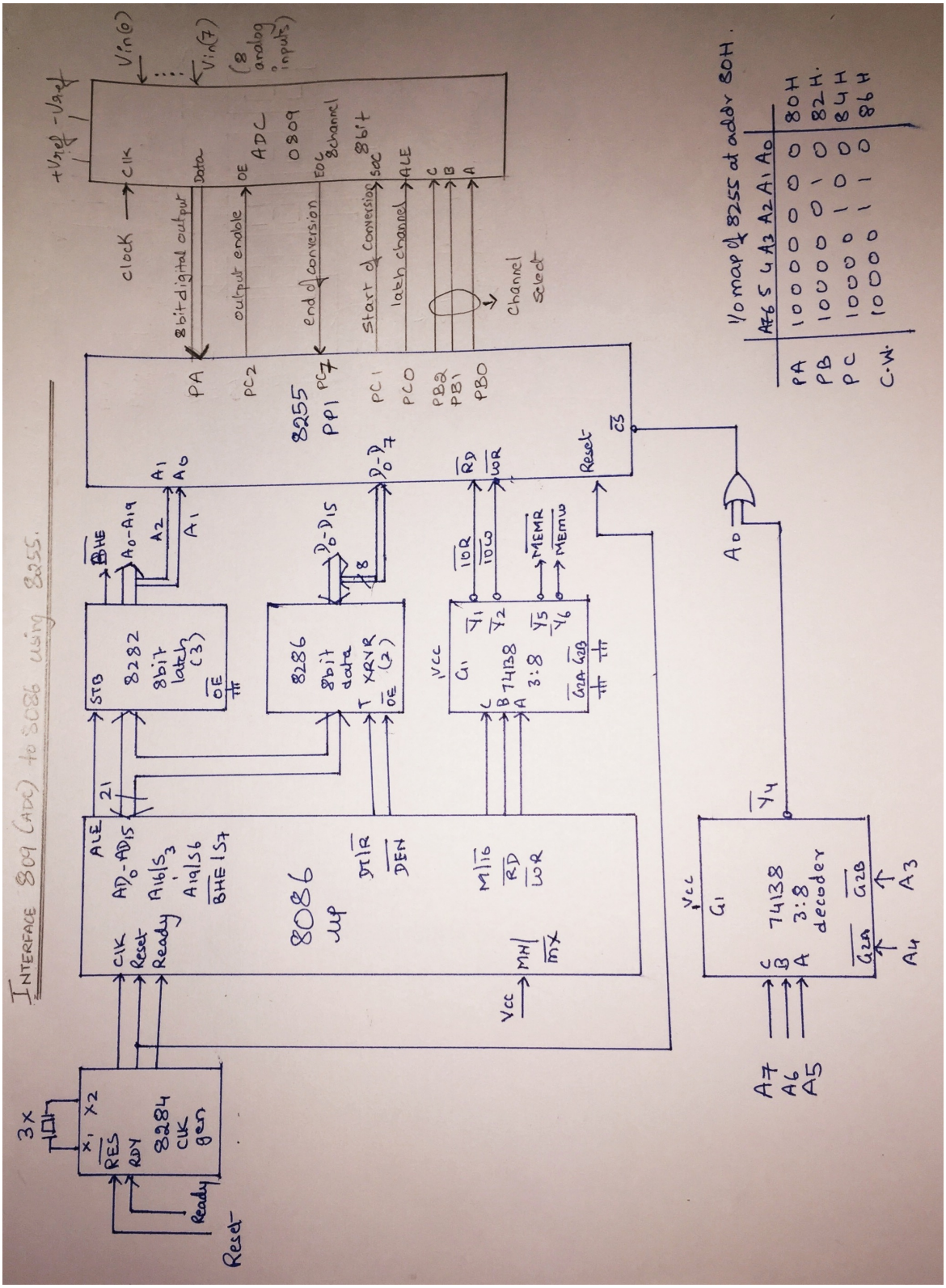

INTERFACE 4x4 MATRIX KEYBOARD WITH 8086 USING 8255
8255 PPI
8086 uP
8284 CLK gen
8282 8bit latch (3)
8286 8bit data XCVR (2)
74138 3:8 decoder
I/O map of 8255 at addr 80H.
A7 A6 A5 A4 A3 A2 A1 A0
PA 10000000 80H
PB 10000010 82H
PC 10000100 84H
C.W. 10000110 86H

ADC 0809


Pin Diagram of ADC 0809
+Vref
-Vref
8bit data o/p
OE
EOC
SOC
ALE
C
B
A
ADC
0809
Vin(0)
Vin(7)
8-analog input voltages
CLK
Vcc
GND
Channel No
EOC: End of Conversion
OE: output Enable
CLK: Clock required for conversion using successive approximations
Vcc, GND: Power supply
Total 28 pins
Vin(0)... Vin(7): 8 analog input voltages
+Vref, -Vref: Reference voltage range
8bit data o/p: Digital data output after conversion.
C,B,A: 3bits to select input channel out of 8 options.
ALE: Latch channel No.
SOC: Start of conversion.

Interface ADC 0809 to 8086 using 8255

1) ADC 0809 is an **8 channel, 8 bit ADC**.
2) It can **convert** an analog **voltage** input **into** an 8 bit digital **data** output.
3) To select an input out of 8 options, there are **three select lines** (C, B and A).
4) We **put a channel number** on these lines (0…7) and latch it using ALE.
5) Now we **give SOC** indicating start of conversion.
6) The channel voltage is internally **sampled** and held into a capacitor.
7) Conversion takes place internally using "**Successive Approximations** Algorithm".
8) **Reference voltage** for conversion is provided using **+Vref and –Vref**.
9) The **clock supply** needed for conversion is given through **CLK** (typically ~ 1MHz).
10) The **end of conversion** is indicated by the ADC using **EOC signal**.
11) Now we **give the OE** signal enabling 8-bit data output from the **ADC to 8255**.
12) This data from 8255 is now **transferred to the microprocessor**.
13) The process is repeated for **subsequent channels**, by changing the channel number.
14) The ADC could also be connected directly to 8086 but **using an 8255 just makes it easier** as the **port lines of 8255 can control various functions of the ADC.**
15) ADCs have a **vast use** in the modern electronic world for **Data Acquisition Systems**.
16) They can be used for **temperature sensing, voice recording, speed sensing** etc.

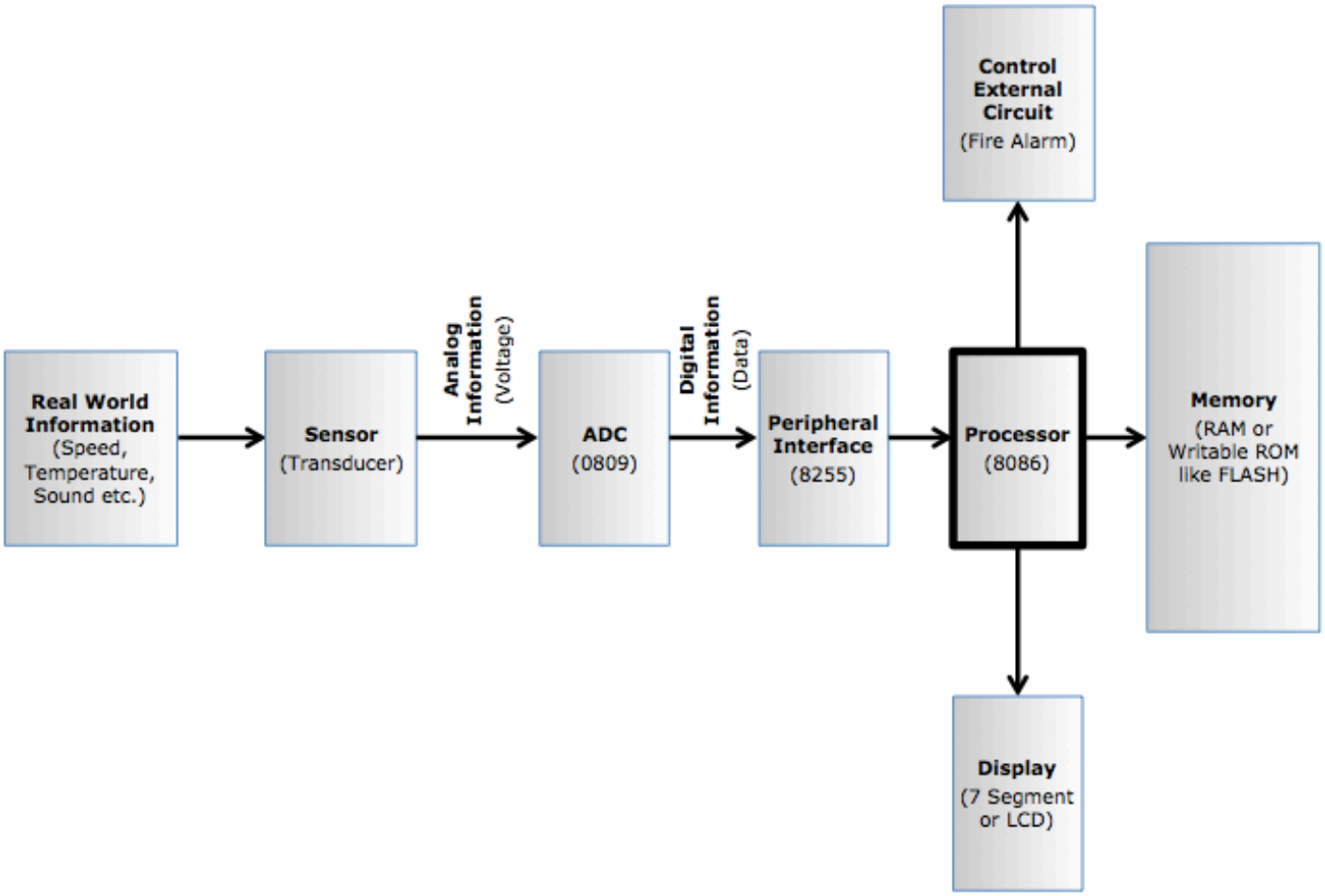

INTERFACE 809 (ADC) to 8086 using 8255.
+Vref -Vref
Vin(0)
Vin(7)
(8 analog inputs)
ADC 0809
clock
8 bit digital output
output enable
end of conversion
Start of Conversion
latch channel
Channel select
8255 PPI
PA
PB
PC
CW
I/O map of 8255 at addr 80H.
A7 A6 A5 A4 A3 A2 A1 A0
PA 10000000 80H
PB 10000010 82H
PC 10000100 84H
C.W. 10000110 86H
8086 up
8282 8bit latch (3)
8286 8bit data T XRVR (2)
74138 3:8 decoder
8284 CLK gen
3x
Reset
Ready

DATA ACQUISITION SYSTEM

Explain a Data Acquisition System using 8086

1) A data acquisition system is required whenever we need to **obtain real world data such as speed, temperature, sound** etc.
2) Real World data is first **converted into electrical voltage pulses** by a sensor like a transducer, a microphone etc.
3) This is now **Analog information**.
4) This is **fed into an ADC** to convert it into **Digital information**, that's data.

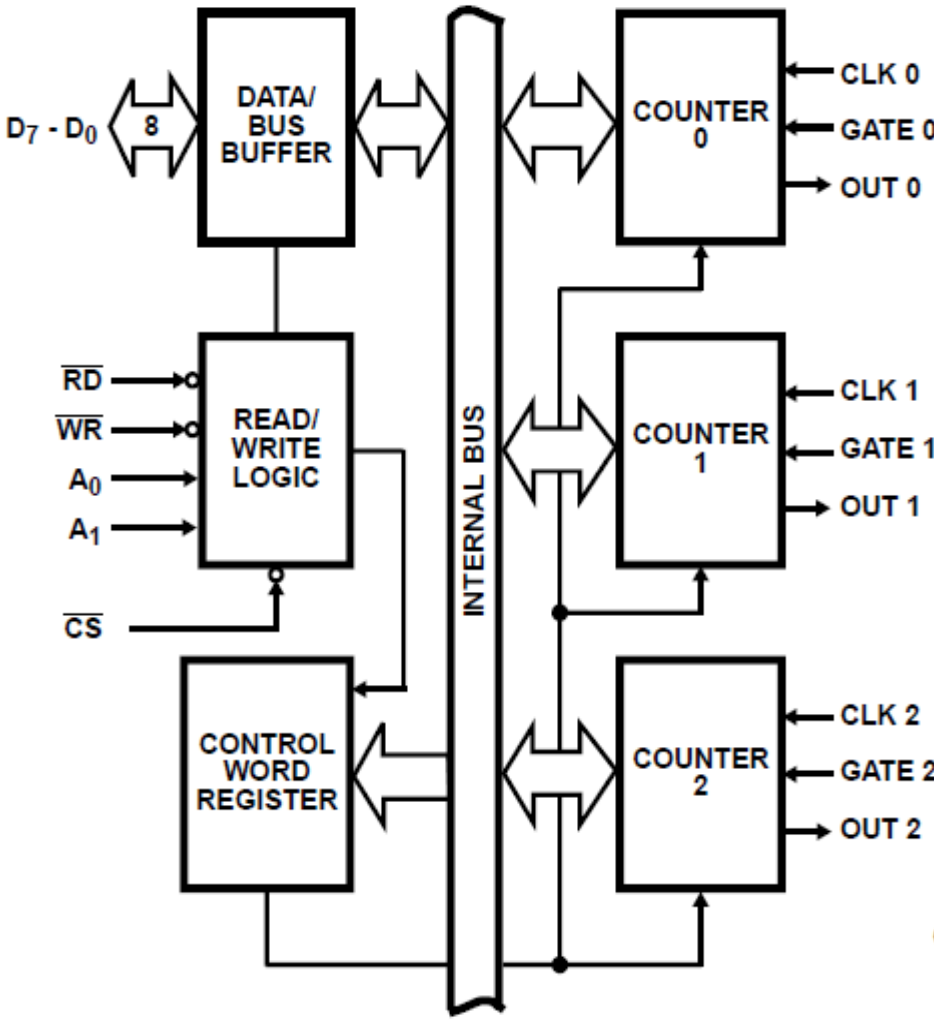
   E.g.:: An **ADC 0809** will convert every Analog sample into 8-bit data.
5) Such data is passed on to a **peripheral interface** device like **8255**.
6) From 8255, it is collected by the **microprocessor 8086**.
7) The ADC could also be connected directly to 8086 but using an 8255 **just makes it easier** as the **port lines of 8255 can control various functions of the ADC.**
8) This data is **stored by the microprocessor into the system memory**.
9) Further on, it **can be processed in various ways**.
10) If it is **Audio**, it can be **stored as an mp3 file**.
11) If it is **temperature or speed** it can be displayed on a **seven segment display**.
12) **Applications: Temperature sensing** in Fire Detection systems**, Speed sensing** in Speed Limiting systems**, Audio recording and playback** (Remember Talking Tomcat app ;-))


Real World Information (Speed, Temperature, Sound etc.)
Sensor (Transducer)
Analog Information (Voltage)
ADC (0809)
Digital Information (Data)
Peripheral Interface (8255)
Processor (8086)
Control External Circuit (Fire Alarm)
Memory (RAM or Writable ROM like FLASH)
Display (7 Segment or LCD)

8254
PROGRAMMABLE INTERVAL TIMER

IC **8254** is **used** as a timer device to produce **Hardware delays**.
It can **also** be **used** to generate a **real-time clock**, or as a **square wave** generator **etc**.
Hardware delays are **more useful** than software delays because the µP is not actively involved in producing the delay. Thus when the delay is being produced the **µP is free** to execute its own program.
The counting is done using **3 independent 16-bit down counters**.
These counters can take the count in **BCD or** in **Binary**.
Once the Counter finishes counting (reqd delay is produced), 8254 interrupts µP.

Architecture of 8254



The architecture of 8254 can be divided into the following parts:

1) Data Bus Buffer

It is used to **interface** the internal **data bus** with the external (system) data bus.
It is thus connected to $\mathbf{D_7 - D_0}$ form the µP.

2) Read Write Logic

It accepts the $\overline{\mathbf{RD}}$ & $\overline{\mathbf{WR}}$ signals, which are used to **control** the flow of **data** through data bus.

The $\overline{\mathbf{CS}}$ signal is used to **select** the **8254** chip.

It also accepts the $\mathbf{A_1 - A_0}$ address lines which are used to **select** one of the **Counters** or the **Control Word** as shown below:

For 8254 A_1 A_0	For 8086 A_2 A_1	Selection	Sample address
0 0	0 0	Counter 0	80 H (i.e. 1000 0**00**0)
0 1	0 1	Counter 1	82 H (i.e. 1000 0**01**0)
1 0	1 0	Counter 2	84 H (i.e. 1000 0**10**0)
1 1	1 1	Control Word	86 H (i.e. 1000 0**11**0)

3) Control Word Register

SC_1	SC_0	RW_1	RW_0	M_2	M_1	M_0	BCD

SC_1 SC_0	Selection
0 0	Select Counter 0
0 1	Select Counter 1
1 0	Select Counter 2
1 1	**READ BACK COMMAND** (Only for 8254; illegal for 8253)

RW_1 RW_0	Selection
0 0	**COUNTER LATCH COMMAND**
0 1	Read/Write LSB Only
1 0	Read/Write MSB Only
1 1	Read/Write LSB First and then MSB

M_2 M_1 M_0	Mode Selection
0 0 0	Mode 0 --- Interrupt On Terminal Count
0 0 1	Mode 1 --- Monostable Multivibrator
X 1 0	Mode 2 --- Rate Generator
X 1 1	Mode 3 --- Square Wave Generator
1 0 0	Mode 4 --- Software Triggered Strobe
1 0 1	Mode 5 --- Hardware Triggered Strobe

BCD	Type of Count
0	Binary Counter (1 digit ➔ 0H … FH) ∴ Hex Count
1	BCD Counter (1 digit ➔ 0 …9) ∴ Decimal Count

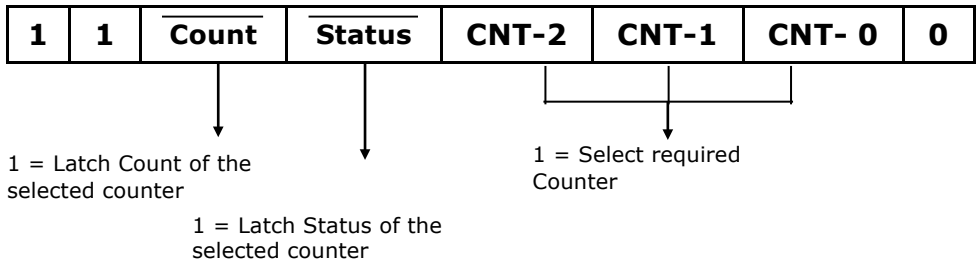
The Control Word Register is an **8-bit** register that holds the Control Word as shown above.
It is selected when $A_1 - A_0$ contain 11.
It has a different format when a Read Back command is given for 8254, as shown below

Read Operations
There are 3 ways in which the µP can read the current count:

A) **Ordinary Read**
In this method, the **counting** is **stopped** by controlling the gate input of a selected counter.
The Counter is then selected by $A_1 - A_0$ and IO Read operation is performed.
First **IO Read** will give the Lower byte of the Count value, and the second IO Read will give the higher byte. **For doubts contact Bharat Sir on 98204 08217**
The **disadvantage** here is that **counting** is **disturbed/stopped**.

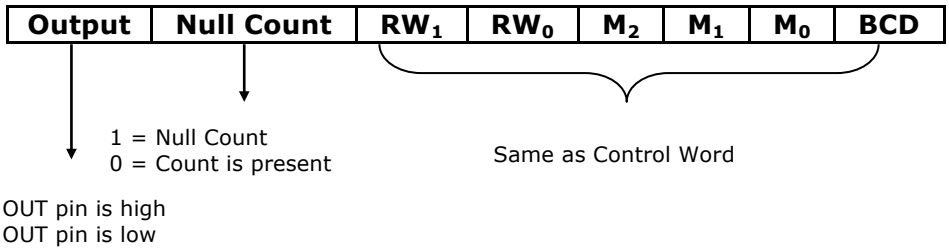
B) **Read on Fly**
In this method, the µP reads the **count** value **while** the **counting** is still in progress.
Thus, it is called as **Read On Fly**.
The appropriate value is written in the control word, and IO Read operation is performed.
The current value of the **Count** is **"latched" internally** and returned to the µP.
The **advantage** here is that counting is **not disturbed**.

C) **Read Back Command**



- The Read Back Command is available **only** for **8254** and not for 8253.
- The Read Back Command **reads** the **Count** value in the **same manner** as **Read On Fly**.
- In **addition** to the Count, the current **Status** can **also** be **latched** using the Read Back command.
- Thus, the appropriate value (for latching the Count and/or Status of the selected counter) is placed in the Control word as shown above.
- The **advantage** here is that the **Count** and the **Status both** can be read **without disturbing the counting**.

Status Word (Status returned after the Read Back Command)



4) 3 Independent Counters

- 8254 has **3 Independent**, **16-bit down counters**.
- Each counter can operate have a **Binary** or **BCD** count.
- Each counter can be in **one of** the **six** possible **modes**.
- Each counter can have a **max count** of **2^{16} = 65535 i.e. FFFFH**.
- Each counter has the following signals:

i. **Clk (Clock Input)**
ii. **Gate(Gate Input)**
iii. **Out (Clock Output)**

- The **input** clock signal is applied on the **CLK** line.
- The **counter decrements** the "**count value**" on **every pulse** of the **input clock at CLK**.
- **When** the **count** becomes **zero** (**Terminal Count i.e. TC**), the **status** of the **OUT** pin **changes**. **For doubts contact Bharat Sir on 98204 08217**
  This can be used to interrupt the µP.
- The **GATE** pin is used to **control** the **Counting**.
  In most modes, the **count** value gets **decremented only if** the **GATE** pin is **high**.

Cell: 98204 08217
BharatSir@hotmail.com

i. **Clk (Clock Input)**
ii. **Gate(Gate Input)**
iii. **Out (Clock Output)**

- The **input** clock signal is applied on the **CLK** line.
- The **counter decrements** the "**count value**" on **every pulse** of the **input clock at CLK**.
- **When** the **count** becomes **zero** (**Terminal Count i.e. TC**), the **status** of the **OUT** pin **changes**.
  This can be used to interrupt the µP.
- The **GATE** pin is used to **control** the **Counting**.
  In most modes, the **count** value gets **decremented only if** the **GATE** pin is **high**.

Timer Modes of 8254

♦ **Mode 0 --- Interrupt on Terminal Count**

1) When this mode is selected **OUT** pin is **initially low**.
2) The **count** value is **loaded**.
3) **GATE** pin is made **high**, so **counting** is **enabled**.
4) **During counting, OUT** pin remains **low**.
5) **On Terminal Count** (TC) the **OUT** pin goes **high**, and remains high.
6) **During counting** if **GATE** is made **low**, it **disables counting**.
   When **GATE** is made **high**, counting **Resumes**.
   Effect of Gate:
   Low ➔ Disables Counting
   High ➔ Enables (Resumes) Counting

♦ **Mode 1 --- Monostable Multivibrator**

1) When this mode is selected **OUT** pin is **initially high**.
2) The **count** value is **loaded**.
3) **Counting begins ONLY when** a **rising edge** is applied to the **GATE**.
4) **OUT** pin goes **low** and remains low **during counting**.
5) **On Terminal Count** (TC) the **OUT** pin goes **high**, and remains high.
6) **During counting** if **GATE** is made **low**, it **has no effect** on the Counting**.**
7) The **GATE pin** can be used as a **Trigger**. ☺ In case of doubts, contact Bharat Sir: - 98204 08217.
   The **Counter** can be **re-triggered** by applying a **rising edge** on the **GATE**.
   This would **Restart** the **counting**, and hence re-trigger it.
   Effect of Gate:
   Low ➔ No Effect
   High(Trigger) ➔ Starts Counting, can also re-rtigger it.

♦ **Mode 2 --- Rate Generator**

1) When this mode is selected **OUT** pin is **initially high**.
2) The **count** value is **loaded**.
3) **GATE** pin is made **high**, so **counting** is **enabled**.
4) **During counting, OUT** pin remains **high**.
5) The OUT pin goes low for one clock cycle just before the TC.
6) The initial count is reloaded and the above process repeats.
   Thus, this mode produces a Continuous Pulse.
7) **During counting** if **GATE** is made **low**, it **disables counting**.
   When **GATE** is made **high**, counting **Restarts**.
   Effect of Gate:
   Low ➔ Disables Counting
   High ➔ Enables (Restarts) Counting
8) It is also called a divide by n counter, as for a count n, the input frequency is divided by n to produce the output frequency.

BHARAT ACHARYA EDUCATION
Cell: 98204 08217
BharatSir@hotmail.com

♦ **Mode 3 --- Square Wave Generator**

1) When this mode is selected **OUT** pin is **initially high**.
2) The **count** value is **loaded**.
3) **GATE** pin is made **high**, so **counting** is **enabled**.
4) **OUT** pin remains **high** for **half of the count** (n/2) and remains **low** for the **remaining half**.
5) **On TC,** the **Count** is **reloaded** and the **process repeats** itself producing a **continuous square wave**.
6) **During counting** if **GATE** is made **low**, it **disables counting**.
   When **GATE** is made **high**, counting **Restarts**.
   Effect of Gate:
   Low ➔ Disables Counting
   High ➔ Enables (Restarts) Counting
7) If the **count** is **ODD**, the **OUT** pin remains **high** for **(n+1)/2** and **low** for **(n-1)/2.**

♦ **Mode 4 --- Software Triggered Strobe**

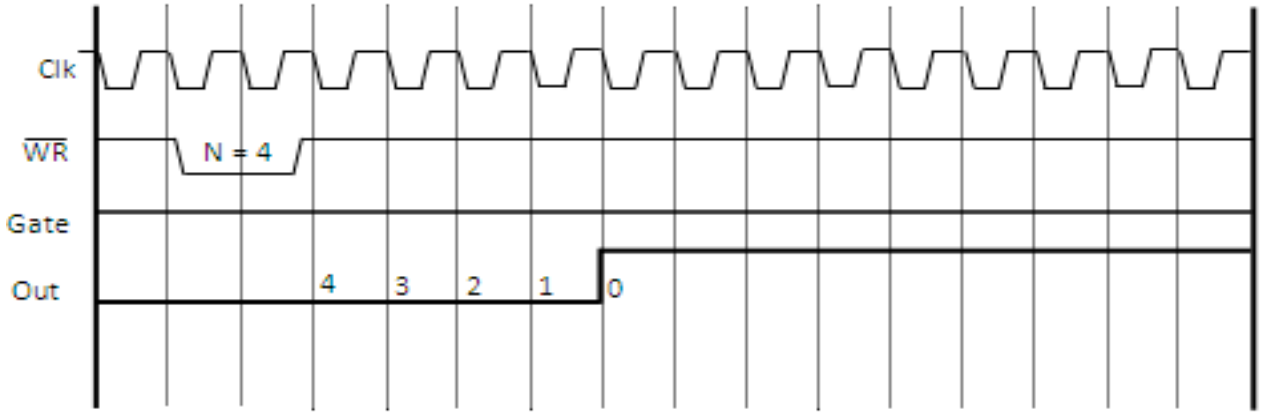
1) When this mode is selected **OUT** pin is **initially high**.
2) The **count** value is **loaded**.
3) **GATE** pin is made **high**, so **counting** is **enabled**.
4) **During counting, OUT** pin remains **high**.
5) The **OUT** pin goes **low** for **one** clock **cycle, just after TC**.
6) **After that OUT** pin goes **high** and remains high.
7) **During counting** if **GATE** is made **low**, it **disables counting**.
   When **GATE** is made **high**, counting **Restarts**.
   Effect of Gate:
   Low ➔ Disables Counting
   High ➔ Enables (Restarts) Counting

♦ **Mode 5 --- Hardware Triggered Strobe**

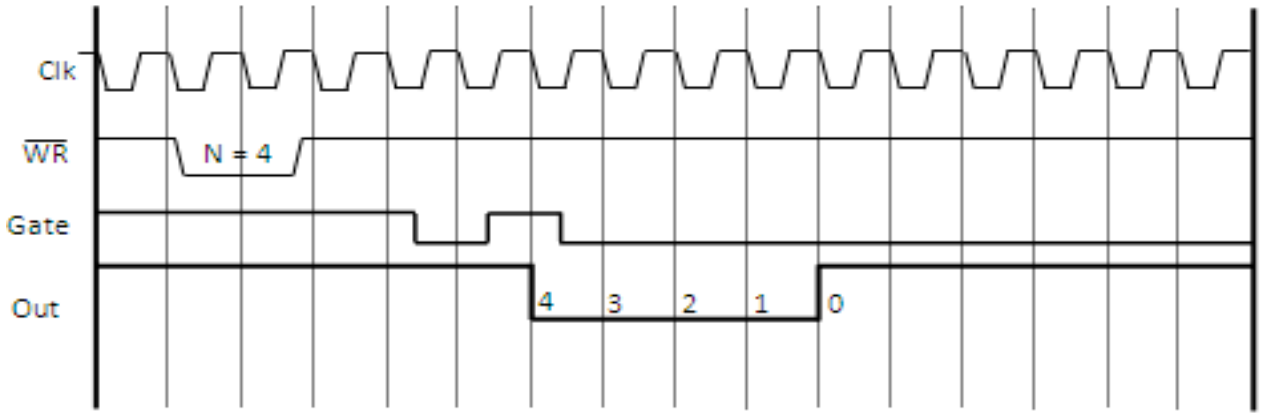
1) When this mode is selected **OUT** pin is **initially high**.
2) The **count** value is **loaded**.
3) Counting starts ONLY after a trigger is applied to the GATE pin.
4) Also, the GATE pin need not remain high for the counting to continue.
5) **During counting, OUT** pin remains **high**.
8) The **OUT** pin goes **low** for **one** clock **cycle, just after TC**.
9) **After that OUT** pin goes **high** and remains high.
6) **Thus GATE is used as a Trigger.**
   I.e. It has to be triggered to start counting.
   Effect of Gate:
   Low ➔ No Effect on counting
   High (Trigger) ➔ Starts Counting, can also re-rtigger it.

BHARAT ACHARYA EDUCATION
Cell: 98204 08217
BharatSir@hotmail.com

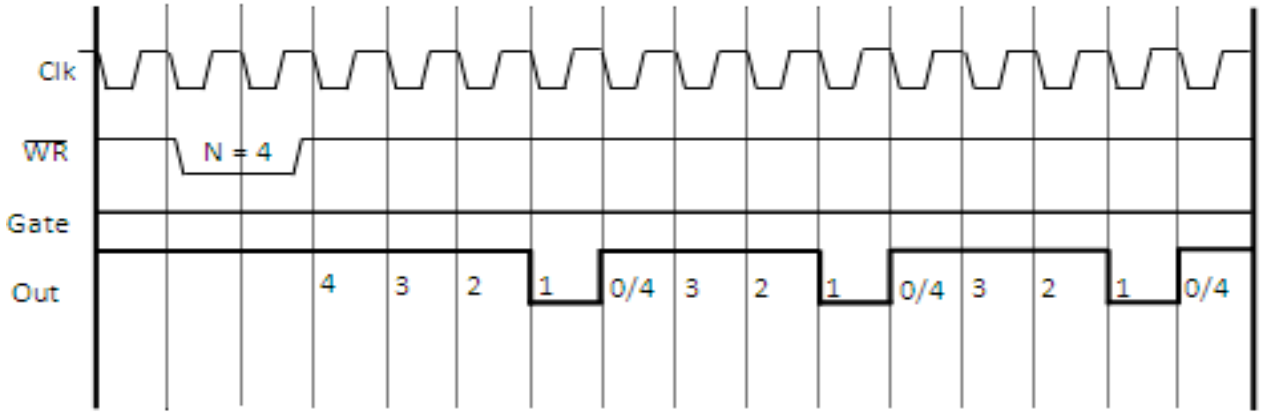
Mode 0: Interrupt On Terminal Count



Mode 1: Monostable Multivibrator

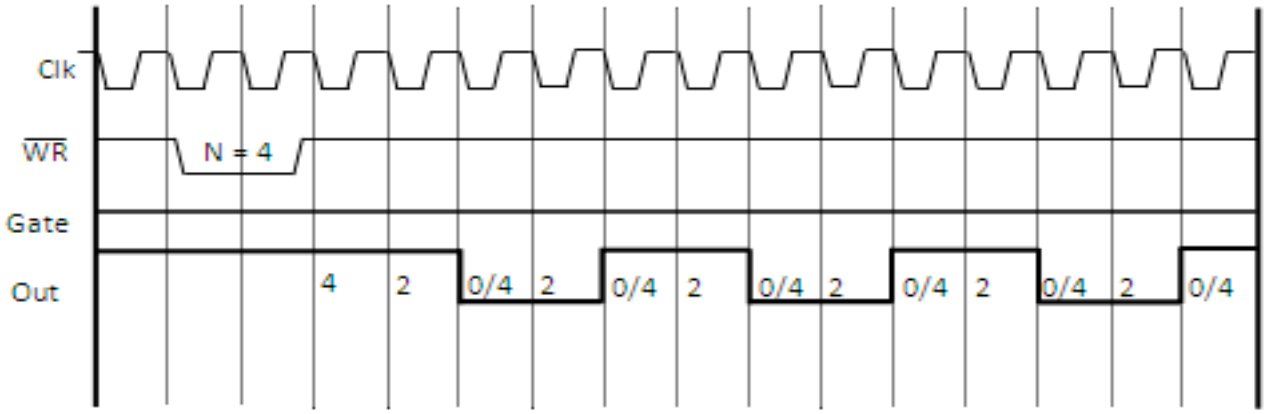


Mode 2: Rate Generator

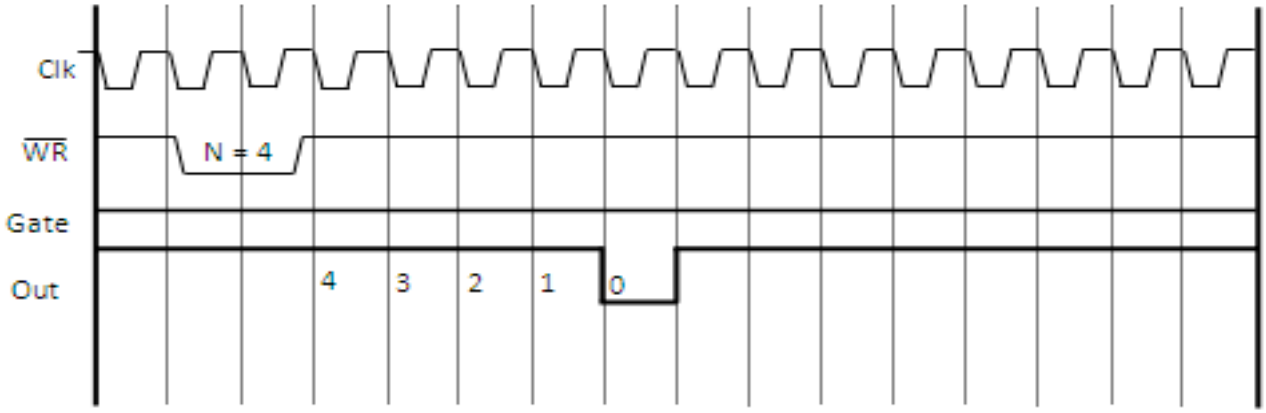


BHARAT ACHARYA EDUCATION
Cell: 98204 08217
BharatSir@hotmail.com

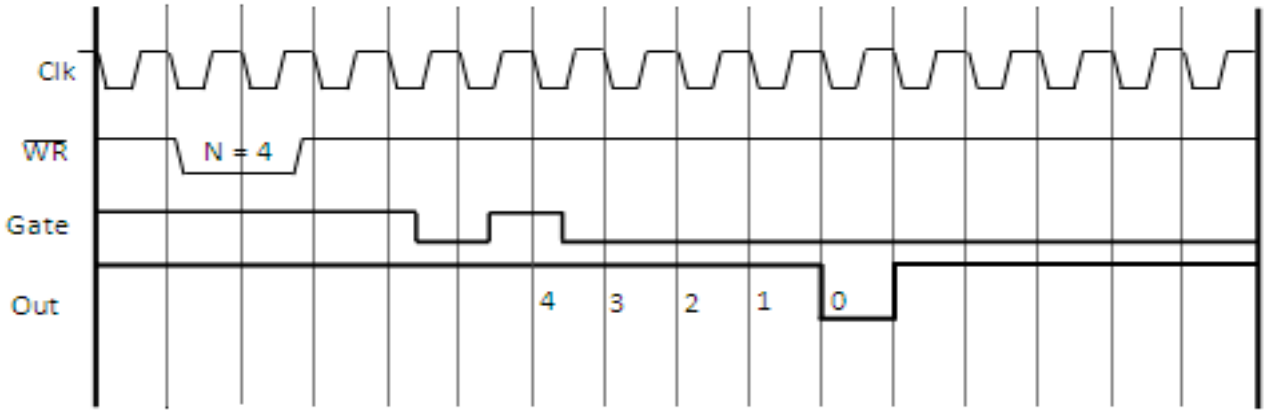
Mode 3: Square Wave Generator



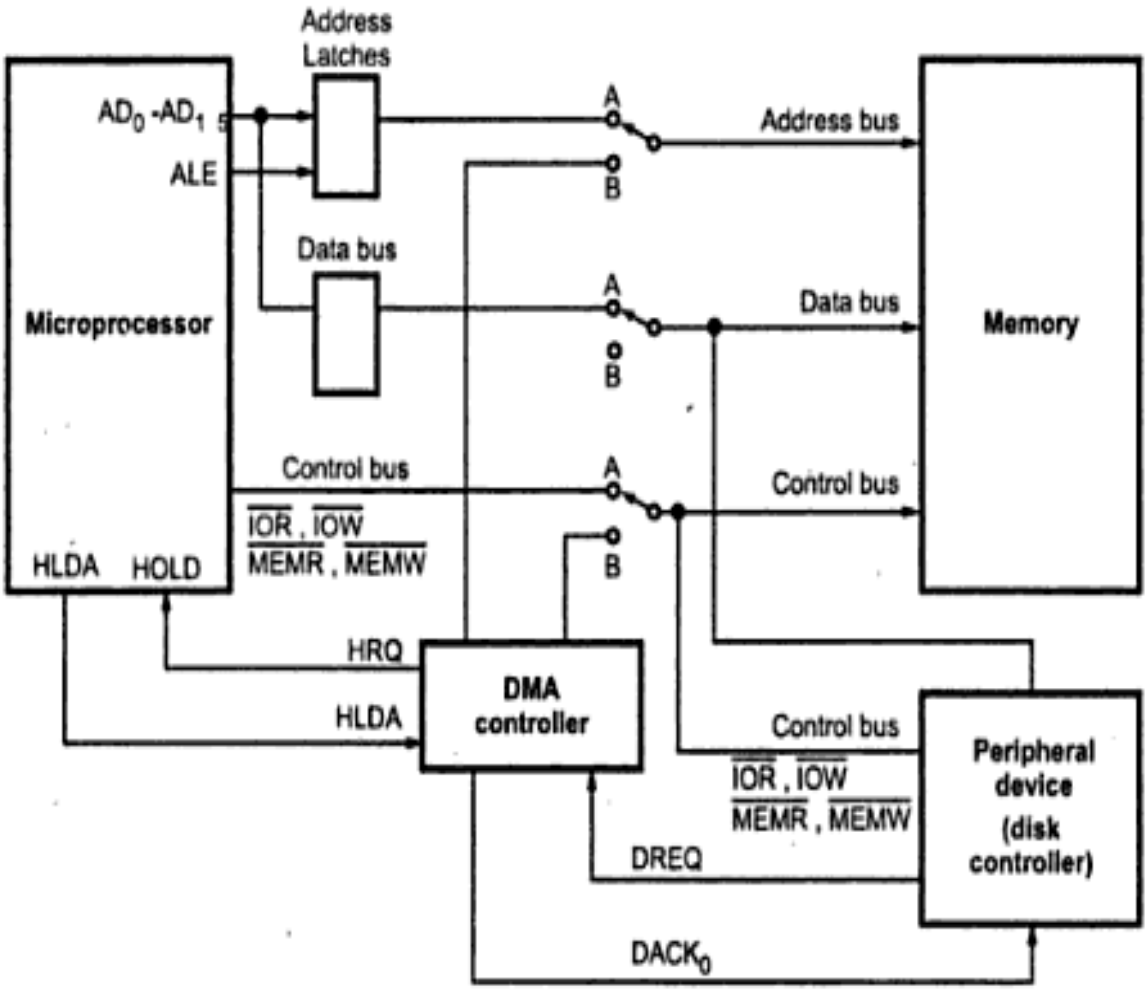
Mode 4: Software Triggered Strobe



Mode 5: Hardware Triggered Strobe



CONCEPT OF DMA



DMA Transfer is a **hardware controlled I/O** Transfer technique.
It is mainly used for **high-speed data transfer between I/O and Memory** where the speed of the peripheral is generally faster than the µP. **For doubts contact Bharat Sir on 98204 08217**
In Program Controlled I/O, Status or interrupt driven I/O the speed of transfer is **slow** mainly **because instructions** need to be **decoded** and **then executed** for the transfer.
DMA transfer is **software independent** and **hence much faster**.
A device known as the DMA Controller (DMAC) is responsible for the DMA transfer.

The **sequence of DMA transfer** is as follows:

1) µP initializes the DMAC by giving the starting address and the number of bytes to be transferred.
2) An **I/O device requests** the **DMAC**, to perform DMA transfer, **through the DREQ line**.
3) The **DMAC** in turn sends a **request signal to** the **µP**, through the **HOLD line**.
4) The **µP finishes the current machine cycle** and **releases** the **system bus** (gets disconnected from it).
   It also **acknowledges** receiving the HOLD signal through the **HLDA line**.
5) The **DMAC acquires control of the system bus.**
   The **DMAC sends** the **DACK signal** to the I/O peripheral and the **DMA transfer begins**.
6) After every byte is transferred, the Address Reg is incremented (or decremented) and the Count Reg is decremented.
7) This continues till the Count reaches zero (Terminal Count). Now the DMA transfer is completed.
8) **At the end** of the transfer, **the system bus is released by the DMAC** by making HOLD = 0.
   Thus **µP takes control of the system bus** and continues its operation.

The DMA Controller (DMAC) does DMA transfer through its channels.
The minimum requirements of each channel are:

i. **Address Register** (to store the starting address for the transfer).
ii. **Count Register** (to store the number of bytes to be transferred).
iii. **A DREQ signal** from the IO device**.**
iv. **A DACK signal** to the IO device.

Operation Cycles of DMAC

There are mainly two operation cycles of a DMAC.

i. Idle Cycle

After Reset, the DMAC is in idle state (idle cycle).
During idle state, **no DMA operation is taking place**.
No DMA requests are **active**.
The **initialization** of the DMAC takes place in the idle mode.

ii. Active Cycle

Once **DMA operation begins,** the **DMAC** is said to be **in active mode**.
Now the **DMAC controls** the **system bus**.
There are **three types of ACTIVE DMA Cycles** while performing DMA transfer:

1) DMA Read

The DMAC **reads** data **from** the **memory** and **writes into** to the **I/O device**.
Thus, **MEMR** and **IOW** signals are used.

2) DMA Write

The DMAC **reads** data **from** the **I/O device** and **writes into** to the **memory**.
Thus, **IOR** and **MEMW** signals are used.

3) DMA Verify

In this cycle, 8237 does not generate any control signals.
Hence, no data transfer takes place.
During this time, the peripheral and the DMAC verify the correctness of the data transferred, using some error detection method.

Transfer Modes of 8237

8237 has **four modes of data transfer**:

1) Single Byte Transfer Mode/ Cycle Stealing.

Once the DMAC becomes the bus master, it will transfer only **ONE BYTE** and return the bus back to the microprocessor. As soon as the microprocessor performs one bus cycle, DMAC will once again take the bus back from the microprocessor.
Hence both **DMAC and microprocessor** are **constantly stealing bus cycles** from each other.
It is the **most popular** method of DMA, because it keeps the **microprocessor active in the background**.
After a byte is transferred, the **CAR** and **CWCR** are **adjusted** accordingly.
The **system bus is returned** to the **µP.**
For further bytes to be transferred, **the DREQ line must go active again**, and then the entire operation is repeated.

2) Block Transfer Mode.

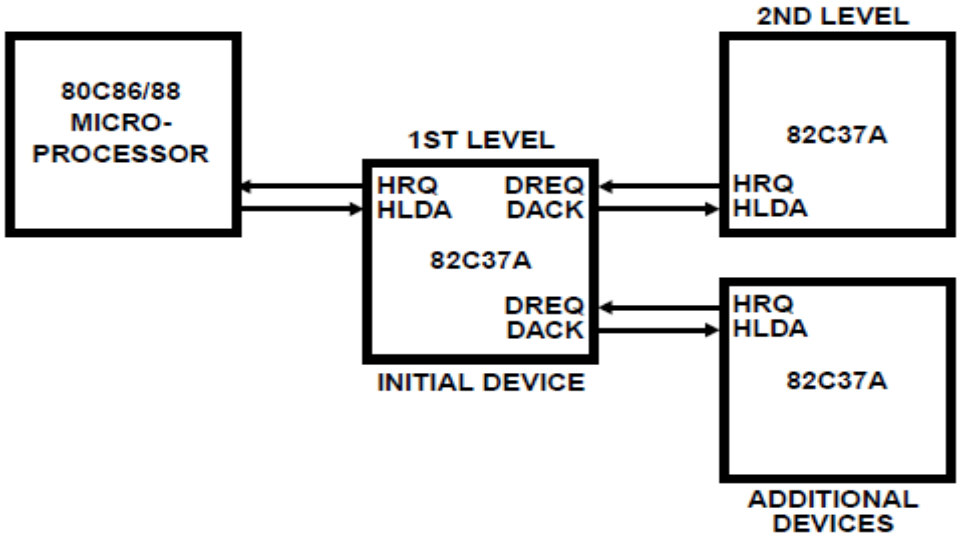
In this mode, the DMAC is programmed to **transfer ALL THE BYTES** in one complete DMA operation.
After a byte is transferred, the **CAR and CWCR are adjusted** accordingly.
The **system bus** is **returned** to the **µP, ONLY after all the bytes are transferred.** I.e. **TC** is **reached or EOP** signal is issued.
It is the **fastest** form of DMA but keeps the **microprocessor inactive** for a long time.
The **DREQ** signal **needs to be active only in the beginning** for requesting the DMA service initially.
Thereafter **DREQ can become low during the transfer**.

3) Demand Transfer Mode.

It is very **similar to Block Transfer**, **except** that the **DREQ must active throughout the DMA operation**.
If during the operation **DREQ goes low**, the **DMA operation is stopped** and the **busses** are **returned** to the **µP.** *#Please refer Bharat Sir's Lecture Notes for this ...*
In the meantime, the **µP** can **continue** with its own operations. **Once DREQ goes high again**, the **DMA operation continues** from where it had stopped.

4) Cascade Transfer Mode. Specific for 8237

In this mode, **more than one DMACs** are cascaded together.
It is used to **increase** the **number of devices interfaced** to the **µP**.
Here we have **one Master DMAC**, to which **one or more Slave DMACs** are connected. The **Slave gives HRQ** to the Master **on the DREQ of the Master**, and the **Master gives HRQ** to the **µP on the HOLD of the µP.**



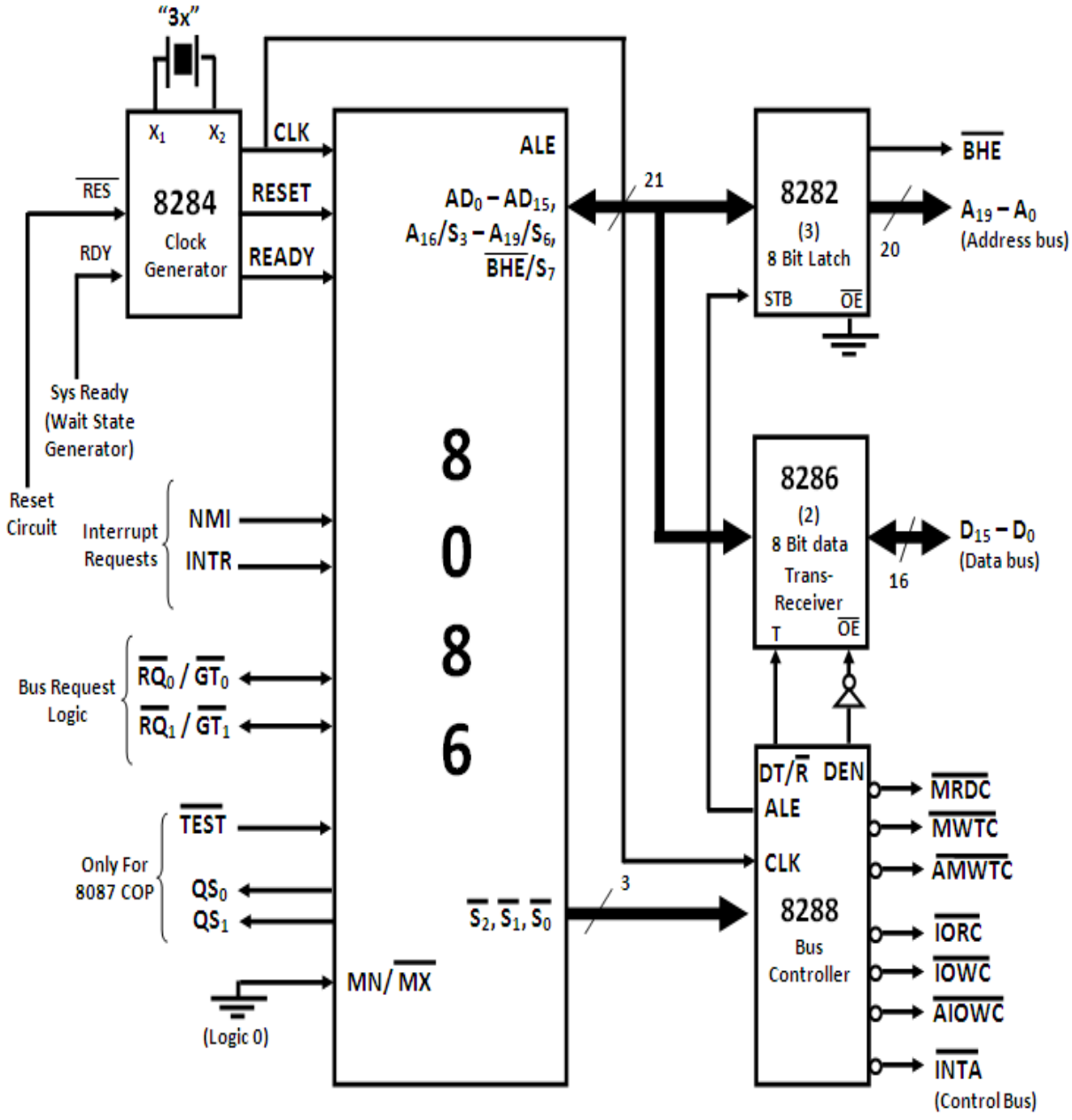
Single Board Computer Design

8086 DESIGNING

WWW.BHARATACHARYAEDUCATION.COM

Q1) Design an 8086 based Maximum Mode system working at 6 MHz having the following:
32KB EPROM using 16KB chips,
128KB RAM using 32KB chips,
Two 16-bit input and two 16-bit output ports all interrupt driven (20m)

Soln: Show 8086 max mode config with a crystal of 18 MHZ.



Memory Calculations:

EPORM:

Required = 32 KB, Available = 16 KB

No. of chips = 2 chips.

Starting address of EPROM is calculated as:

FFFFFH – (Space required by total EPROM of 32 KB)

```
  F F F F F H
-   7 F F F H
  F 8 0 0 0 H
```

Size of a single EPROM chip = 16 KB

$= 16 \times 1KB = 2^4 \times 2^{10}$
$= 2^{14}$
= 14 address lines

= **(A 14 … A1)**

RAM:

Required = 128 KB, Available = 32 KB

No. of chips = 4 chips.

Starting address of RAM is: 00000H

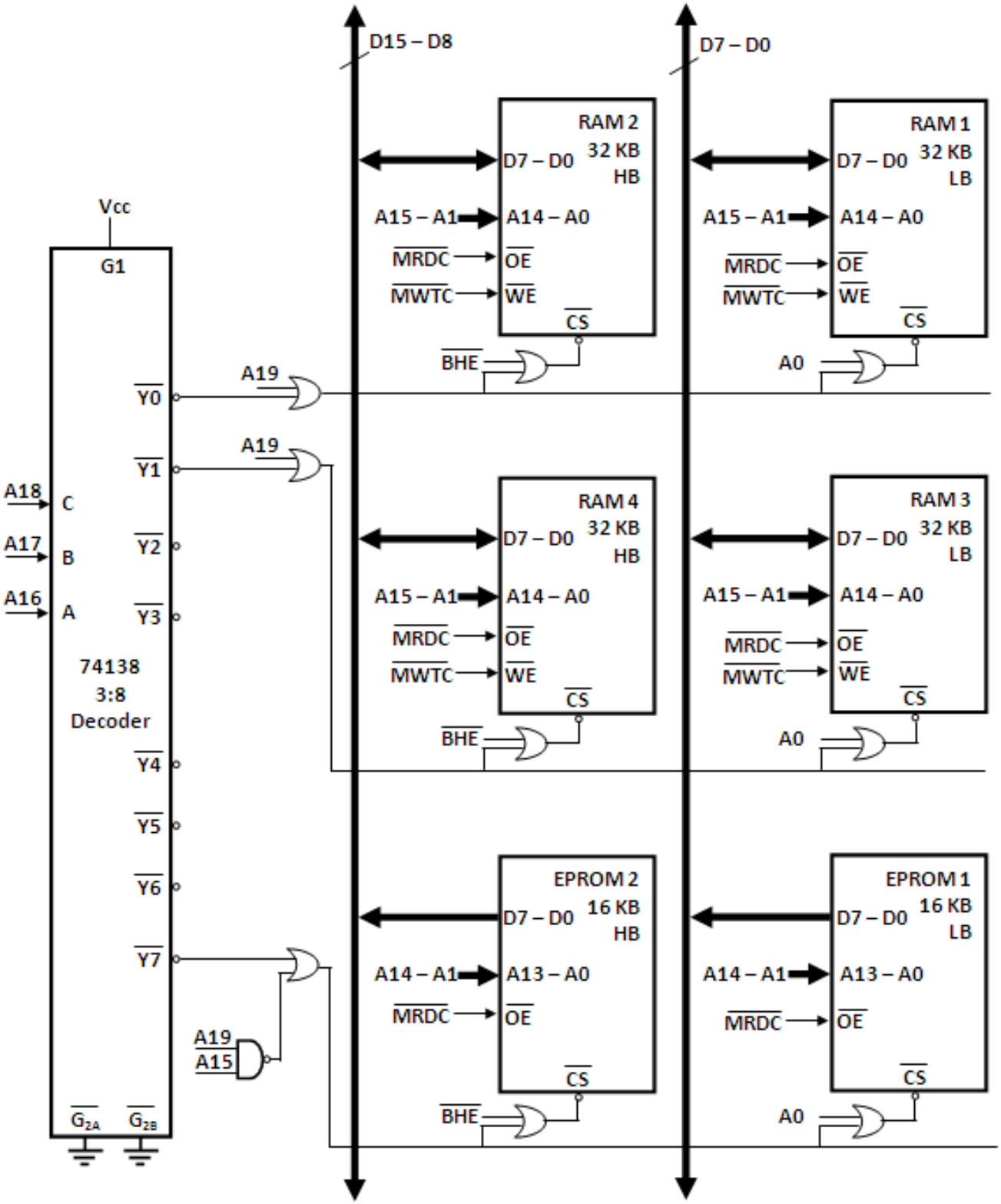
Size of a single RAM chip = 32 KB
$= 32 \times 1 \text{ KB} = 2^5 \times 2^{10}$
$= 2^{15}$
= 15 address lines

= **(A15 … A1)**

For doubts contact Bharat Sir at 98204 08217

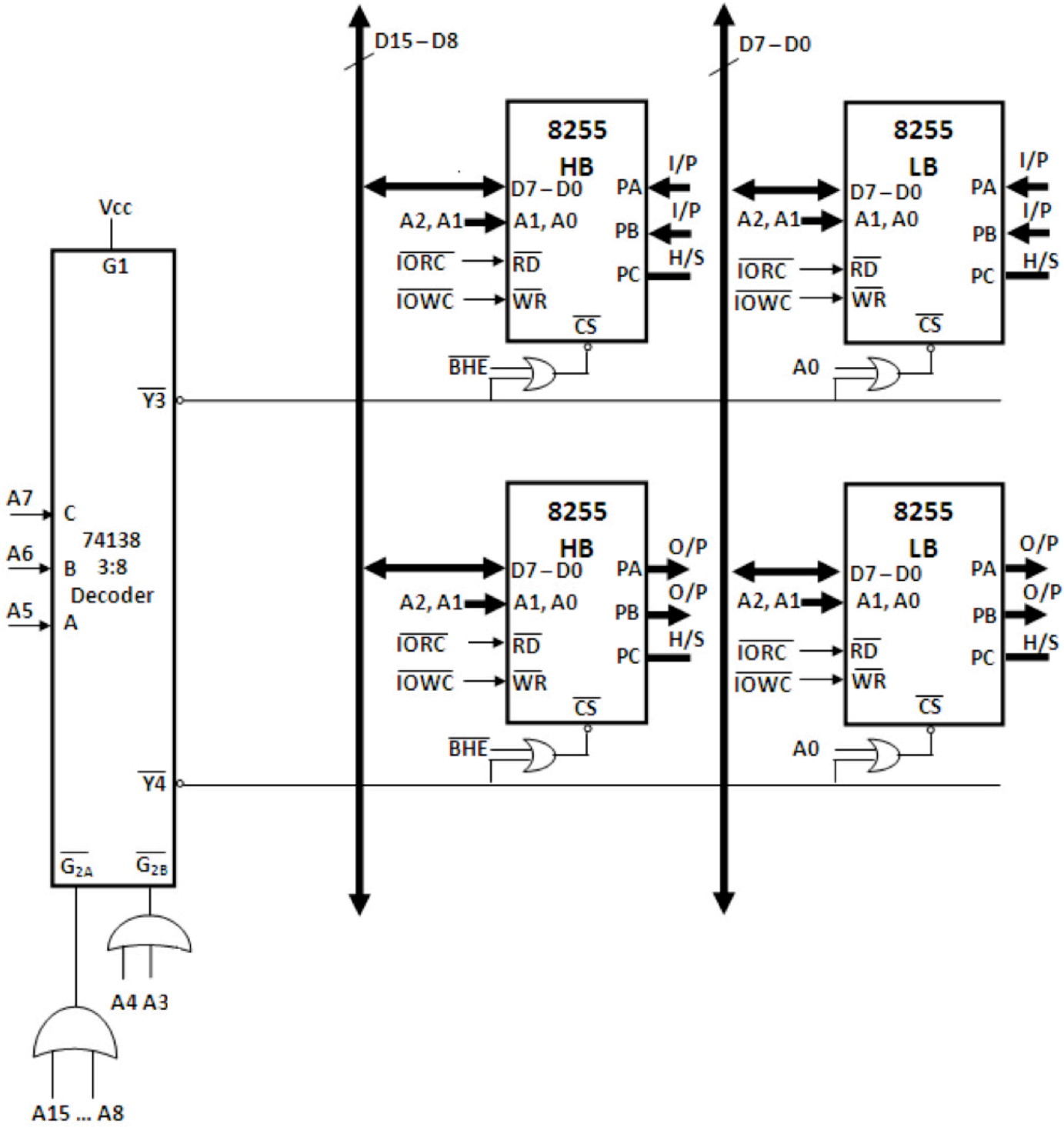
MEMORY MAP

Memory Chip	Address Bus																			Memory Address
	A19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	A0	
RAM 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	00000H
(LB)	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0FFFEH
RAM 2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	00001H
(HB)	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0FFFFH
RAM 3	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10000H
(LB)	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1FFFEH
RAM 4	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	10001H
(HB)	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1FFFFH
EPORM 1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	F8000H
(LB)	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	FFFFEH
EPORM 2	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	F8001H
(HB)	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	FFFFFH


D15 – D8
D7 – D0
RAM 2
32 KB
HB
RAM 1
32 KB
LB
D7 – D0
A15 – A1
A14 – A0
MRDC
MWTC
OE
WE
CS
BHE
A0
Vcc
G1
A19
Y0
Y1
A18
A17
A16
C
B
A
Y2
Y3
74138
3:8
Decoder
RAM 4
RAM 3
Y4
Y5
Y6
Y7
EPROM 2
16 KB
EPROM 1
A14 – A1
A13 – A0
A19
A15
G2A
G2B

I/O Map

I/O Port	Address Bus																I/O Address
	A1	14	13	12	11	10	9	8	7	6	5	4	3	2	1	A0	
8255 LB																	
Port A	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0060H
Port B	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0062H
Port C	0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0	0064H
C Word	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	0	0066H
8255 HB																	
Port A	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	0061H
Port B	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	1	0063H
Port C	0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	1	0065H
C Word	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	1	0067H
8255 LB																	
Port A	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0080H
Port B	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0082H
Port C	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0084H
C Word	0	0	0	0	0	0	0	0	1	0	0	0	0	1	1	0	0086H
8255 HB																	
Port A	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0081H
Port B	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	0083H
Port C	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	1	0085H
C Word	0	0	0	0	0	0	0	0	1	0	0	0	0	1	1	1	0087H


D15 – D8
D7 – D0
8255
HB
8255
LB
D7 – D0
A2, A1
A1, A0
IORC
IOWC
RD
WR
CS
PA
PB
PC
I/P
O/P
H/S
BHE
A0
Vcc
G1
Y3
Y4
A7
A6
A5
C
B
A
74138
3:8
Decoder
G2A
G2B
A4 A3
A15 ... A8

IMPORTANT POINTS TO REMEMBER FOR I/O DESIGNING

- **Normally** I/O devices are mapped using **I/O mapped I/O** which means I/O devices are given I/O addresses
- Here **I/O addresses** can be either **8-bit or 16 bit**.
- If the question says **direct addressing mode** or **fixed port addressing**,
  Then use an **8-bit address like 80H (A7-A0)**.
- If the question says **indirect addressing** or **variable port addressing**,
  Then use **16-bit address like 8000H (A15-A0)**.
- If nothing is mentioned, use any of the above techniques.
- If **memory mapped I/O** is asked (Very rare), then remember the **following changes**
  Give the I/O device a **20-bit unused memory address like 80000H (A19-A0)**
  Connect **MEMR# and MEMW#** signals to the I/O device instead of the usual IOR# and IOW# signals

Differentiate between

	I/O MAPPED I/O	MEMORY MAPPED I/O
1	I/O device is **treated as an I/O device** and hence **given an I/O address**.	I/O device is **treated like a memory device** and hence **given a memory address**.
2	I/O device has an **8 or 16 bit I/O address**.	I/O device has a **20 bit Memory address.**
3	I/O device is given **IOR# and IOW#** control signals	I/O device is given **MEMR# and MEMW#** control signals
4	**Decoding is easier** due to lesser address lines	**Decoding is more complex** due to more address lines
5	Decoding is **cheaper**	Decoding is more **expensive**
6	Works **faster due to less delays**	More gates add more delays hence **slower**
7	Allows **max 2^{16} = 65536 I/O devices**	Allows **many more I/O devices** as I/O addresses are now 20 bits.
8	I/O devices can **only** be accessed by **IN and OUT** instructions.	I/O devices can now be accessed using **any memory instruction**.
9	ONLY **AL/ AH/ AX registers** can be used to transfer data with the I/O device.	**Any register** can be used to transfer data with the I/O device.
10	**Popular** technique in **Microprocessors**.	**Popular** technique in **Microcontrollers**.

BUS CONTENTION / BUS ARBITRATION / PRIORITY RESOLVING SCHEMES

In a Loosely coupled system all processors can use their local bus simultaneously. But the system bus can be used by only one module at a time. Hence there is contest for the system bus. This is called bus contention. It is resolved buy various arbitration schemes having different priority methods.

A) Daisy Chain Method

- **All** bus **masters use** the **same** line for Bus **Request**.
- **If** the **Bus Busy** line is **inactive**, the **Bus Controller gives** the **Bus Grant signal**.
- Bus **Grant** signal is **propagated serially** through all masters **starting** from **nearest** one.
- The bus **master**, which requires the system bus, **stops this signal**, **activates** the Bus **Busy** line and **takes control** of the system bus.

Advantage:

i. **Design** is **simple**.
ii. The **number of control lines is less**. Also **adding** new bus masters is **easy**.

Disadvantage:

i. **Priority** of bus masters is **rigid** and **depends** on the **physical proximity** of the bus masters with the bus arbiter i.e. The one nearest to the Bus Arbiter gets highest priority.
ii. Bus is granted **serially** and hence a **propagation delay** is induced in the circuit.
iii. **Failure of one** of the devices may **fail** the entire **system**.

B) Polling Method

- Here also **all bus masters** use the **same** line for **Bus Request**.
- Here the **controller generates** binary **address** for the master.
  Eg: To connect 8 bus masters we need 3 address lines ($2^3 = 8$).
- In **response** to a **Bus Request**, the **controller "polls"** the bus masters by **sending** a **sequence** of bus master **addresses** on the address lines.Eg: 000,010,100,011 etc
- The selected **master activates** the **Bus Busy** line and **takes control** of the bus.

Advantage:

i. The **Priority** is **flexible** and can easily be **changed** by **altering** the **polling sequence**.
ii. If **one** module **fails**, the entire **system does not fail**.

Disadvantage:

i. **Adding** more bus masters is **difficult** as **increases** the number of **address lines** of the circuit. Eg: In the above circuit to add the 9^{th} Bus Master we need 4 address lines.

C) Independent Request Method

- Here, **all bus masters** have their **individual Bus Request** and **Bus Grant lines**.
- The **controller** thus **knows which master** has **requested**, so bus is granted to that master.
- **Priorities** of the masters are **predefined** so on **simultaneous** Bus **Requests**, the bus is **granted based** on the **priority**, provided the Bus Busy line is not active.
- The **Controller** consists of **encoder** and **decoder** logic for the priorities.

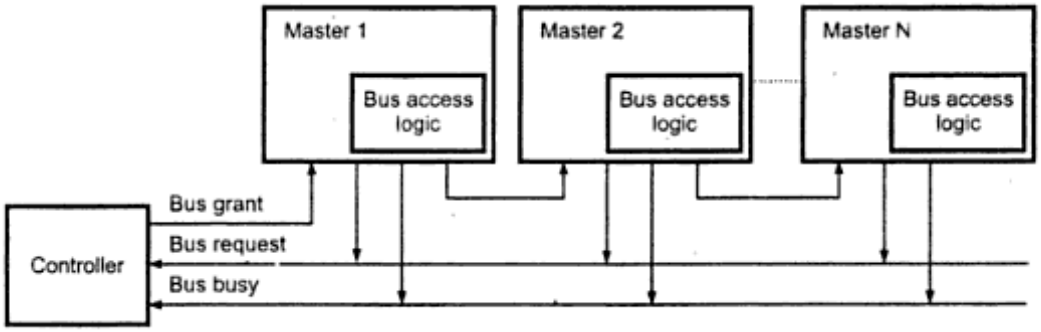
Advantage:

i. The Bus **Arbitration is fast**.
ii. The **speed of** Bus **Arbitration** is **independent** of the **number** of devices connected.

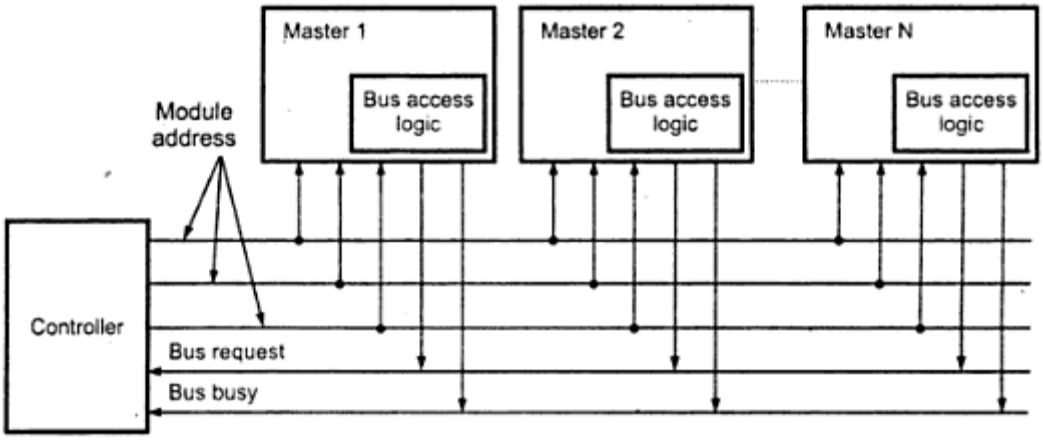
Disadvantage:

i. The **number of control lines required is more** (2n line required for n devices).

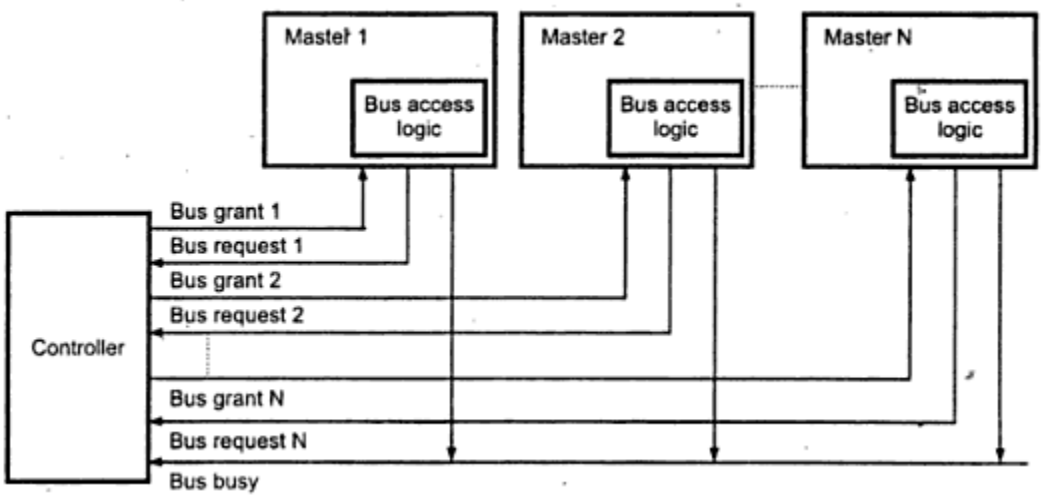
Daisy Chaining



Polling



Independent Requests



Sem V | Computers | Biomedical

Mumbai – 2017

VIVA QUESTIONS

WATCH ALL MY VIDEOS ON:

www.bharatacharyaeducation.com

BHARAT ACHARYA
E: bharatsir@hotmail.com | M: 9820408217

8085 Based Questions

Examiner
How many pins does 8085 have?
Q: 1

You
8085 is a 40 pin IC.

Examiner
8085 Address Bus and Memory?
Q: 2

You
8085 has a 16-bit address bus.
Therefore it can access 64 KB memory.

Examiner
8085 Data Bus and ALU?
Q: 3

You
8085 has an 8-bit data bus and an 8-bit ALU.
This means it can transfer 8-bits in one cycle and also operate on 8 bits in 1 cycle. Therefore 8085 is called an 8-bit microprocessor.

Examiner
Explain PC of 8085?
Q: 4

You
PC – Program Counter, is a 16-bit register. It contains address of the next instruction. It gets incremented as soon as an instruction is fetched.

Examiner
Explain SP and Stack of 8085?
Q: 5

You
SP – Stack Pointer, is a 16-bit register. It contains address of the top of stack. Stack is a data structure present in the memory. It operates in LIFO manner. During a Push, SP gets decremented and during a Pop, SP gets incremented.

Examiner
Explain role of incrementer/ decrementer?
Q: 6

You
INR/ DCR is used to increment the value of PC after every instruction is fetched and increment or decrement the value of SP after a POP or a PUSH operation Respectively.

Examiner
How many general purpose registers does 8085 have?
Q: 7

You
7 – A, B, C, D, E, H and L. They are all 8-bit registers.

Internal Examiner Joins the Viva!

Internal Examiner
How many register pairs does 8085 have?
Q: 8

You ~ Bharat Sir's Student
3 – BC, DE and HL. Additionally there is also WZ but it is not available to the programmer

Internal Examiner
What do you know about 8085 interrupts?
Q: 9

You ~ Bharat Sir's Student
8085 has 5 hardware interrupts.
In order of priority, they are:
Trap, RST 7.5, RST 6.5, RST 5.5, INTR.
Trap is edge & level triggered.
RST 7.5 is edge triggered. Others are level triggered.
Trap cannot be masked or disabled. Others can be disabled.
Additionally there are 8 software interrupts RST0… RST7.

Internal Examiner
What is the use of SID and SOD?
Q: 10

You ~ Bharat Sir's Student
Used for Serial Communication (Bit by Bit).
Serial communication is slow but is cheaper so is preferred for long distance communication.

Internal Examiner
Explain Clock of 8085?
Q: 11

You ~ Bharat Sir's Student
8085 operates at 3 Mhz.
It is connected to a crystal of 6 MHz on pins X1 & X2.
The frequency is divided by 2, using a T – Flip Flop.
This is done to produce a clock of 50% duty cycle.
The same clock is given to other devices using the Clkout signal.

Internal Examiner
Explain Ready Signal of 8085?
Q: 12

You ~ Bharat Sir's Student
Ready is used to synchronize the Microprocessor with slower peripherals. Devices inform microprocessor if they are ready or not. Ready = 1 means devices are ready so microprocessor continues. Ready = 0 means devices are not ready and hence microprocessor enters wait state.

Internal Examiner
Why is 8085 called "8085"?
Q: 13

You ~ Bharat Sir's Student
The 8 indicates that it is an 8-bit processor.
The 5 is because 8085 was the first microprocessor to operate on 5V Power supply. The remaining is the series number.

Internal Examiner
Explain multiplexing in 8085?
Q: 14

You ~ Bharat Sir's Student
Multiplexing is done to reduce the number of lines.
In 8085, A0-A7 and D0-D7 are multiplexed, to form AD0-AD7.
ALE is used to identify whether the bus carries address or data.
If ALE = 1, bus carries address else data.

8086 Based Questions

External Examiner
How many pins does 8086 have?
Q: 15

You
8086 is a 40 pin IC.

Internal Examiner
8086 Address Bus and Memory?
Q: 16

You
8086 has a 20-bit address bus.
Therefore it can access 1 MB memory.

External Examiner
8086 Data Bus and ALU?
Q: 17

You
8086 has a 16-bit data bus and a 16-bit ALU.
This means it can transfer 16-bits in one cycle and also operate on 16-bits in 1 cycle. Therefore 8086 is called a 16-bit microprocessor.

Internal Examiner
Explain pipelining?
Q: 18

You
Fetching the next instruction while executing the current instruction is called pipelining. It saves time. It fails whenever there is a branch. The instruction fetched is no longer valid and hence has to be discarded.

Internal Examiner
Explain GPRs of 8086. Do they also have some special uses?
Q: 19

You ~ Bharat Sir's Student
There are 4 16-bit GPRs called AX, BX, CX & DX. They can also be used as 8 independent 8-bit GPRs: AL, AH; BL, BH…
They also have some special uses:
AX – Accumulator; BX – Memory pointer;
CX – Count Register; DX – 32-bit Accumulator along with AX.

External Examiner
How Many Segments are in 8086?
Q: 20

You ~ Bharat Sir's Student
There are 4 segments: Code, Stack, Data and Extra.

Internal Examiner
What is the max and min size of a segment and why?
Q: 21

You ~ Bharat Sir's Student
Max size: 64 KB. This is because offset addresses are of 16 bits. 16 Bytes. This is because formula of Physical address is segment address x 10h plus offset address. This puts a minimum gap of 10h bytes between two segments, which means 16 bytes.

Guess Who? #9820408217
Good Going… Keep it up ☺
All the best!

External Examiner
What are the advantages of segmentation?
Q: 22

You ~ Bharat Sir's Student
We can access 1 MB memory using 16 bit addresses. Moreover, the memory becomes more organized and overlaps are prevented between code, stack and data.

External Examiner
Name Segment and offset registers?
What is the difference between SP and BP?
Q: 23

You ~ Bharat Sir's Student
Segment registers: CS, SS, DS, ES. Offset Registers: IP, SP & BP, SI, DI.
SP and BP, both are for Stack Segment.
SP gives offset address of the top of stack. It is used for Push and Pop.
BP gives offset address of any location of the stack. It is used for random access of the stack.

External Examiner
Why is memory banking done? How many banks are there?
Q: 24

You ~ Bharat Sir's Student
8086 has a 16-bit data bus.
Memory banking is done to access 16-bit data, which is stored in two consecutive locations. Hence memory is divided into 2 banks, Even bank and Odd Bank, also called Lower and Higher banks respectively.
8085 had an 8-bit data bus so it did not require memory banking.

Internal Examiner
How are the memory banks selected?
Q: 25

You ~ Bharat Sir's Student
Draw A0 and BHE table from #BharatSir notes.

Internal Examiner
Can we access locations 24000H and 24001H simultaneously?
Q: 26

You ~ Bharat Sir's Student
Yes. By Ignoring A0, both these locations have the same bit pattern on A1-A19. Locations are selected on the basis of A1-A19, whereas A0 and BHE are used to select the banks.
Such data is called "Aligned Data".

Internal Examiner
Can we access locations 24001H and 24002H simultaneously?
Q: 27

You
No. Even after Ignoring A0, these two locations don't have the same address. So they will have to be accessed in two cycles.
Such data is called "Misaligned Data".

Bharat Sir
Shabbaash! I was sure you could answer that ☺
Keep it up!

External Examiner
Give advantages of 8086 ALU over 8085 ALU?
Q: 28

You ~ Bharat Sir's Student
8086 has a 16-bit ALU compared to 8085, which had an 8-bit ALU.
8086 ALU can perform MUL and DIV, which 8085 could not.
8086 ALU does not depend upon Accumulator for most of its operations, whereas 8085 had an Accumulator based ALU.

External Examiner
Explain Flag register of 8086?
Q: 29

You ~ Bharat Sir's Student
Please refer #BharatSir notes for diagram and answer.
Show examples from #BharatSir lecture notes if asked.

External Examiner
How is overflow determined?
Q: 30

You ~ Bharat Sir's Student
OF is determined by Ex-Or of C7 and C6.
It basically means that the result is out of range for a signed number.
Range for 8-bit signed number (-80h…00h…+7Fh).
Range for 16-bit signed number (-8000h…0000h…+7FFFh).

External Examiner
Why are DF, IF and TF called Control Flags?
Q: 31

You ~ Bharat Sir's Student
They are used by the programmer to control different operations.
DF: Controls direction for String Instructions.
IF: Controls enabling and disabling of interrupts.
TF: Controls Single Stepping used for debugging of programs.

External Examiner
What is the size of the Prefetch queue? Why? When is the Queue Refilled? Why?
Q: 32

You ~ Bharat Sir's Student
The Prefetch Queue is of 6-bytes as it is the max size of an instruction. It is refilled whenever two bytes are empty as 8086 has a 16-bit data bus.

External Examiner
What is the advantage of using String Instructions?
Q: 33

You ~ Bharat Sir's Student
String instructions are fetched and decoded only once but executed repeatedly. So they can perform large transfers very fast.

External Examiner
Difference between MUL and IMUL?
Q: 34

You ~ Bharat Sir's Student
MUL is used for unsigned numbers. IMUL for signed.

External Examiner
Difference between SUB and CMP?
Q: 35

You ~ Bharat Sir's Student
Both perform Subtraction. SUB will store the result and affect the flags. CMP will not store the result. Will only affect the flags.

External Examiner
Where is TEST instruction used?
Q: 36

You ~ Bharat Sir's Student
To check a Semaphore in Multiprocessor Systems for principle of Mutual Exclusion.

External Examiner
How do you interchange the nibbles of a number?
Q: 37

You ~ Bharat Sir's Student
Rotate it 4 times: MOV CL, 04h, ROL AL, CL; This will Interchange nibbles of AL.

External Examiner
Where is ADC used?
Q: 38

You ~ Bharat Sir's Student
To add 2 large numbers.
Lower Bytes are added using ADD, higher Bytes using ADC.

External Examiner
Where is DAA used?
Q: 39

You ~ Bharat Sir's Student
To add two BCD numbers (Decimal Numbers). First enter the numbers, Add them, Perform DAA, by its adjustment logic, it gives the correct BCD answer.

External Examiner
MOV AL, 25H, DAA;
Show the working of DAA in this example?
Q: 40

You ~ Bharat Sir's Student
Won't work. DAA stands for Dec Adjust "After Addition"!

External Examiner
How can we make any bit of a register "0"?
Q: 41

You ~ Bharat Sir's Student
To make any bit 0: AND that bit with "0" & remaining bits with "1"
To make any bit 1: OR that bit with "1" & remaining bits with "0"
To complement a bit: XOR that bit with "1" & remaining bits with "0"

External Examiner
What is the use of various String Instructions?
Q: 42

You ~ Bharat Sir's Student
MOVS: To transfer a block as it is.
LODS & STOS: To Manipulate and Transfer a block
CMPS: To Compare two strings
SCAS: To Search for a number in a string.

External Examiner
Transfer block form 12345H to 54678H using MOVS?
Q: 43

You ~ Bharat Sir's Student
Initialize as follows: DS: 1000H, SI: 2345H, ES: 5000H & DI: 4678H

External Examiner
How do we perform NAND, NOR or XNOR?
Q: 44

You ~ Bharat Sir's Student
There are no direct instructions for these operations.
AND followed by NOT is a NAND.
OR followed by NOT is a NOR.
XOR followed by NOT is a XNOR.

External Examiner
When is CBW used?
Q: 45

You ~ Bharat Sir's Student
To expand an 8-bit signed number into 16 bits by retaining its MSB.

External Examiner
Can the binary number (1000 0011) can have two values?
Q: 46

You ~ Bharat Sir's Student
Yes. Depending upon whether we are working in signed or unsigned system. Unsigned: 83H. Signed: -7DH.

Bharat Sir ~ For Doubts Call Me @ 9820408217
Excellent answer… Most students get this one wrong!
Keep it up!

External Examiner
Difference between Control Flags and Status Flags?
Q: 47

You ~ Bharat Sir's Student
Status flags give status of Current result and are changed by the ALU.
Control flags are changed by the programmer & are used to control different operations.

External Examiner
Perform (-25H) + (-35H). Show effect on Flags?
Q: 48

You ~ Bharat Sir's Student
Answer: -5AH. Flags: OF = 0, SF = 1, ZF = 0, AC = 1, PF = 1, CF = 1.
In case of doubts please refer #BharatSir classroom examples.

External Examiner
What is the use of Control Flags?
Q: 49

You ~ Bharat Sir's Student
DF: Decides Direction for String Operations (0- decrement, 1-increment)
IF: Enables or Disables INTR interrupt (0 – disable, 1 - enable)
TF: Perform single stepping (1- perform, 0 – don't perform)

External Examiner
Why does IF affect only INTR interrupt?
Q: 50

You ~ Bharat Sir's Student
There are 256 interrupts. We can't disable software interrupts.
We can't disable NMI. That leaves only INTR interrupt.

External Examiner
What are Vectored and Non Vectored interrupts?
Q: 51

You ~ Bharat Sir's Student
Interrupts that have a Fixed Vector Number are called Vectored.
All Software interrupts and NMI are Vectored.
An interrupt whose Vector Number is Not Fixed is called Non-Vectored.
INTR is the only Non Vectored Interrupt of 8086.

External Examiner
Why does 8086 give INTA#?
Q: 52

You ~ Bharat Sir's Student
INTA# is given in response to INTR to obtain the Vector Number as INTR is Non Vectored. 8086 gives two INTA#. On first INTA# device generates the value of Vector Number. On 2nd INTA# device sends the value of Vector Number to 8086.

External Examiner
What are INT 0… INT 4 called?
Q: 53

You ~ Bharat Sir's Student
These are called Dedicated Interrupts as their functions are dedicated.
INT 0: Divide Error.
INT 1: Single Stepping.
INT 2: NMI.
INT 3: Breakpoint.
INT 4: Interrupt on Overflow.

External Examiner
What are INT 5… INT 31 called?
Q: 54

You ~ Bharat Sir's Student
These are called Reserved Interrupts.
They are reserved by Intel, for Future Processors.

External Examiner
What are INT 32… INT 255 called?
Q: 55

You ~ Bharat Sir's Student
These are called User Defined Interrupts.
They are available to the programmer and can be used to service any user-defined condition.

External Examiner
Define an interrupt?
Q: 56

You ~ Bharat Sir's Student
An interrupt is a condition that can make the microprocessor execute an ISR

External Examiner
What are Assembler Directives?
Q: 57

You ~ Bharat Sir's Student
These are statements given to the assembler so that it can convert the program into machine language. They are not instructions so they don't have any opcode.

External Examiner
What .ASM file?
Q: 58

You ~ Bharat Sir's Student
It is an Assembly language file
The Assembler converts into .OBJ file, which is in machine language.
The Linker converts it into a .EXE file that can be executed..

External Examiner
What is the use of DB, DW, DD, DQ, DT directives?
Q: 59

You ~ Bharat Sir's Student
These are data types used for declaring variables.
DB: 8-bit. DW: 16-bit. DD: 32-bit. DQ: 64-bit. DT: 80-bits.

External Examiner
What is the use of Assume Directive?
Q: 60

You ~ Bharat Sir's Student
Informs the Assembler, which are the segments. E.g.:: Assume CS: Code, DS: Data.

External Examiner
What is the use of .model Small directive?
Q: 61

You ~ Bharat Sir's Student
It reduces the memory consumed by our program by allowing the Assembler to overlap the segments instead of reserving 64 KB space for every segment.

External Examiner
What is the use of .stack 100 directives?
Q: 62

You ~ Bharat Sir's Student
Allocates 100 bytes to stack. Can be increased if needed.

External Examiner
What is the use of INT 21H?
Q: 63

You ~ Bharat Sir's Student
It is a DOS Interrupt. It performs several functions depending upon the value in AH.

To Accept a Value from User
MOV AH, ***01H***
INT 21H; user given value comes in AL.

To Accept a String from User
LEA DX, String
MOV AH, ***0AH***
INT 21H; user given string stored at location pointed by DX.

To Display a the result on Screen
MOV DL, value
MOV AH, ***02H***
INT 21H; value in Dl will be displayed

To Display a message on Screen
Msg DB 'Hello World' --- do this in data segment
LEA DX, Msg
MOV AH, ***09H***
INT 21H; "Hello World" will be displayed on the screen.

To Terminate the Program
MOV AX, ***0046H***
INT 21H; Termination Code.

External Examiner
What is the use 13, 10, (or 0DH, 0AH) while declaring a string?
Q: 64

You ~ Bharat Sir's Student
It is the ASCII for Carriage Return, Line Feed.
This simply means the string will be displayed on a new line, from the left most position.
This is the same as putting "\n" in C Programming.

External Examiner
Why do we put a $ at the end of a string?
Q: 65

You ~ Bharat Sir's Student
For the Assembler to understand that the string has ended.

External Examiner
Why do we write, "Start" at the beginning?
Q: 66

You ~ Bharat Sir's Student
This indicates the beginning of the first executable statement in the program. Its like "main" of a C program.

External Examiner
Why do we write, "Org" assembler directive?
Q: 67

You ~ Bharat Sir's Student
Org stands for Origin.
It gives the absolute location where the data (following Org) will be stored.

External Examiner
What is 8282? Where is it used?
Q: 68

Bharat Sir's Student
It is an 8-bit latch. It is used to latch the address from the multiplexed address – data bus. It captures address when ALE is 1.

External Examiner
What is 8286? Where is it used?
Q: 69

Bharat Sir's Student
It is an 8-bit data trans-receiver. It is a bi-directional buffer. It passes data. It is enabled by DEN# (DEN bar) and its direction is decided by DT/R# signal.

External Examiner
What is 8284? Where is it used?
Q: 70

Bharat Sir's Student
It is a clock generator. It is used to generate the system clock.
It is connected to a crystal oscillator.
The Crystal frequency is divided by 3, to produce a clock of 33 % duty cycle.
If 8086 operates at 6 MHz we connect a crystal of 18 MHz
8284 also synchronizes Reset and Ready for the entire system.

External Examiner
What is 8288? Where is it used?
Q: 71

Bharat Sir's Student
It is a Bus Controller. It is used to generate control signals in Max Mode. It decodes S2#, S1#, S0# and generates all the control signals.

External Examiner
How do you Generate Control Signals in Min Mode?
Q: 72

You ~ Bharat Sir's Student
In Min Mode, 8086 Produces M/IO#, RD# and WR#. They Are decoded by a 3:8 decoder to produce MEMR#, MEMW#, IOR# and IOW#.

External Examiner
What is the difference between Min Mode and Max Mode?
Q: 73

You ~ Bharat Sir's Student
In Min Mode, there is only one processor that is 8086.
In Max Mode 8086 can work along-with other processors like 8087 and 8089.
8087 is called NDP (Numeric data processor), also called Math Co Processor.
8089 is called I/O Processor.

External Examiner
What is 8288? Where is it used?
Q: 74

You
It is a Bus Controller. It is used to generate control signals in Max Mode.
It decodes S2#, S1#, S0# and generates all the control signals.

External Examiner
What is a Machine Cycle/ Bus Cycle? Example?
Q: 75

You
One operation performed on the system bus is called a Machine Cycle.
Example: Memory Read, Memory Write etc.
In 8086 every Machine cycle requires 4 t-states.

External Examiner
In a Read Machine cycle, why is there a time gap between Address and Data?
Q: 76

You
It a Read Cycle, Data has to travel from Memory (or I/O) to the Microprocessor. This travel takes time. It is called propagation delay.
It is not required in a Write Cycle as data is sent by the microprocessor itself.

External Examiner
Why are Read or Write signals activated in T2?
Q: 77

You ~ Bharat Sir's Student
Because in T1, the multiplexed bus carries address.

External Examiner
Instruction Cycles are Interrupt Breakpoints, Machine Cycles are DMA Breakpoints. Justify?
Q: 78

You ~ Bharat Sir's Student
When an Interrupt occurs microprocessor finishes the current instruction before executing the ISR. In case of doubts call #BharatSir on: 982040 08217.
When a Bus Request (DMA) occurs, microprocessor simply finishes the current machine cycle and releases the bus.

External Examiner
Why does 8086 have two GND pins?
Q: 79

You ~ Bharat Sir's Student
GND is the pin from where all the Current is drained out of the chip.
In order to avoid over-heating on a single ground pin, 8086 has 2 GNDs.

External Examiner
If RESET, HOLD, NMI and INTR occur simultaneously, what is their priority order of service?
Q: 80

You ~ Bharat Sir's Student
The strongest signal of microprocessor is RESET, and will be serviced first.
Then Bus Request Signals (Hold or RG#/GT#). Then NMI and Finally INTR.

External Examiner
When does 8086 check Ready Signal?
Q: 81

You ~ Bharat Sir's Student
Ready is checked during T3 of any machine cycle.
If it is "0", it means device is not ready. Refer #BharatSir lecture notes for more!
"Wait States" are inserted between T3 and T4 till Ready becomes "1"

External Examiner
What are Advanced Write signals?
Q: 82

You ~ Bharat Sir's Student
They get activated 1 T-state in advance. It gives slower devices more time to accept the data. Please refer #BharatSir printed notes for more on this!

External Examiner
What is the use of 8087 NDP?
Q: 83

You ~ Bharat Sir's Student
It is used to perform powerful arithmetic like Trigonometry, Log, Square-Root etc. It operates on Floating Point numbers up to 80 bits. It can perform multiprocessing with 8086 to give higher performance.

External Examiner
What are the data formats of 8087?
Q: 84

You ~ Bharat Sir's Student
Integers: Words Int (16). Short Int (32). Long Int (64 bits).
BCD: Packed BCD (80bits).
Floating Point: Short Real (32). Long Real (64). Temp Real (80) bits.
Draw Floating Point formats from Class Notes.

External Examiner
How does 8087 know if host is 8086 or 8088?
Q: 85

You ~ Bharat Sir's Student
By checking BHE# pin on reset. If it is 0, then it is 8086 else 8088.

External Examiner
What is Normalization of a Floating Point No?
Q: 86

You ~ Bharat Sir's Student
Normalization means converting a Floating Pt Number to Fixed Point.

External Examiner
What are the errors/ exceptions of 8087?
Q: 87

You ~ Bharat Sir's Student
8087 interrupts 8086 due to 6 different errors/ exceptions.
Precision: Mantissa is too large to be stored
Underflow: Exponent is too small to be stored
Overflow: Exponent is too large to be stored.
Zero Divide: When we divide by zero
De-Normal Operand: When operand is not normalized
Invalid Operation: When we perform an invalid operation

External Examiner
What are the types of multi processor systems?
Q: 88

You ~ Bharat Sir's Student
Closely Coupled: Multiple Processors use the same, SHARED Memory, I/O & Buses.
Loosely Coupled: Multiple Processors use their own LOCAL Memory, I/O & Buses, and are connected to each other by a SHARED bus.

External Examiner
What is the Role of 8289 Bus Arbiter?
Q: 89

You ~ Bharat Sir's Student
8289 is used in Loosely Coupled systems to resolve priorities when different modules request for the SHARED system bus.

External Examiner
What are different Arbitration Schemes?
Q: 90

You ~ Bharat Sir's Student
Daisy Chaining, Polling and Independent Requests.

External Examiner
What are the instruction prefixes used in 8086?
Q: 91

Bharat Sir's Student
REP: Used to repeat string instructions "CX" times.
LOCK: Prevents the microprocessor from releasing the bus during an instruction.
ESC: Used to identify an 8087 instruction. (ESC is "11011" in binary)

External Examiner
What is the role of WAIT instruction in 8086?
Q: 92

You ~ Bharat Sir's Student
WAIT makes 8086 check TEST# signal.
TEST# comes from BUSY signal of 8087. If 8087 is BUSY, TEST# will be "1".
Now 8086 enters Wait State, and will resume after TEST# becomes "0".

Bharat Sir ~ Winter is coming.
Well done… Keep it up!
Kuch bhi Karna magar NO-BALL mat dalna!

External Examiner
What is the role of 8259 PIC?
Q: 93

You ~ Bharat Sir's Student
It is used to increase the number of interrupts.
8259 has 8 interrupt pins IR0 – IR7.
If cascaded, 8259 can provide up to 64 interrupts using 1 master and 8 slaves.

External Examiner
What are the priority modes of 8259?
Q: 94

Bharat Sir's Student
FNM: Default mode. IR0 Highest… IR7 Lowest. Used in Single 8259.
SFNM: IR0 Highest… IR7 Lowest. Used in Cascaded 8259.
Rotating Priority: After any interrupt is serviced it becomes lowest priority.

External Examiner
What is SMM in 8259?
Q: 95

Bharat Sir's Student
Any interrupt can interrupt the current ISR, irrespective of their priority, except the interrupt currently being serviced.

External Examiner
What are EOI modes in 8259?
Q: 96

Bharat Sir's Student
Normal EOI Mode: Programmer gives EOI command at the end if ISR to clear bit from in service register.
Auto EOI Mode: 8259 automatically clears bit from in service register after 2nd INTA# cycle.

External Examiner
What is the use of CAS lines in 8259?
Q: 97

Bharat Sir's Student
$CAS_{0,1,2}$ are used by the master to select the slave during 1st INTA#.

External Examiner
What is Polling in 8259?
Q: 98

Bharat Sir's Student
In Poll Mode, instead of 8259 interrupting the microprocessor, 8086 will periodically POLL 8259 to know which interrupt has occurred.

External Examiner
What are the Commands of 8259?
Q: 99

Bharat Sir's Student
There are two types of Commands:
ICW1, ICW2, ICW3 and ICW4: Initialization Command Words.
OCW1, OCW2 and OCW3: Operational Command Words.
These are commands given by the programmer to 8259 to program various properties such an Vector Numbers, Triggering, Masking, Priority etc.

External Examiner
What is the Role of 8255 PPI?
Q: 100

Bharat Sir's Student
8255 provides 3, 8-bit bidirectional I/O ports for connection I/O devices to 8086. It can perform Handshake based data transfer to increase reliability.

External Examiner
What is data transfer Modes of 8255?
Q: 101

Bharat Sir's Student
Mode 0: Simple I/O. All ports transfer data but no handshaking.
Mode1: Handshake I/O. Port A, B transfer data, Port C does Handshaking.
Mode 2: Bidirectional Handshaking. Only Available for Port A.

External Examiner
What are the Commands of 8255 PPI?
Q: 102

Bharat Sir's Student
I/O Command: Used to select the Modes and Directions of all ports of 8255.
BSR Command; Used to set or reset a single line of Port C.

External Examiner
How can we generate a square wave using 8255?
Q: 103

Bharat Sir's Student
Using BSR Command, we can make any line of port C alternate between 1 and 0 (Set and Reset) with a delay after each. Connect this output to a CRO and we get a square wave. By manipulating delay we can also produce Rectangular wave.

Bharat Sir
Superb answer! Say the same thing if asked to blink an LED.
Keep it up!

External Examiner
What is the role of 8253/8254 PIT?
Q: 104

Bharat Sir's Student
It is used to produce hardware delays using three 16-bit Counters: C0, C1 & C2.

External Examiner
What are the 6 Counting Modes of 8254 PIT?
Q: 105

You ~ Bharat Sir's Student
Mode 0: Interrupt on Terminal Count
Mode 1: Mono-stable Multi-vibrator
Mode 2: Rate Generator
Mode 3: Square Wave Generator
Mode 4: Software Triggered Strobe
Mode 5: Hardware Triggered Strobe

External Examiner
What are the methods of reading the count?
Q: 106

Bharat Sir's Student
Ordinary Read: Counting has to be stopped.
Counter Latch Command: Can read the Count without stopping.
Read Back Command: Can read Count and Status without stopping.

External Examiner
What is the difference between 8253 & 8254?
Q: 107

Bharat Sir's Student
8253 does not have a Read Back Command

External Examiner
What if we give an ODD count for Mode 3 Sq. Wave Gen?
Q: 108

You ~ Bharat Sir's Student
The Extra pulse is always given on the HIGH side.
Count = 6: 3-High, 3-Low pulses. Count = 7: 4-High, 3-Low pulses

Bharat Sir
That was a tricky one… Well Done!
Call: 9820408217

External Examiner
What is DMA?
Q: 109

You ~ Bharat Sir's Student
Direct Memory Access means transferring data directly between Memory and I/O without involving the microprocessor. It is extremely fast due to two reasons: The transfer is direct and it is hardware based so no time is wasted in fetching and decoding instructions.

External Examiner
Explain DMA operation in brief?
Q: 110

You ~ Bharat Sir's Student
DMA Controller (8237/ 8275) makes Hold = 1. 8086 Releases system bus and makes HLDA = 1. DMAC becomes bus master and transfers data directly between Memory and I/O. At the end DMAC makes Hold = 0. 8086 becomes bus master again.

External Examiner
What are DMA Data Transfer Modes?
Q: 111

You ~ Bharat Sir's Student
Block transfer also called Burst Mode: DMAC performs full data transfer and only then return the bus back to 8086.
Cycle Stealing also called Single Byte transfer: Both 8086 and DMAC perform Single Cycles alternately.
Demand transfer: Transfer stops whenever DREQ becomes low, and resumes when DREQ becomes high again (transfer happens only on demand)
Transparent Mode also called Hidden Mode: Transfer done only when 8086 is idle.

External Examiner
What is Cascaded mode of DMA?
Q: 112

You ~ Bharat Sir's Student
DMAC has 4 Channels.
If we want more devices then we can connect DMAC in a cascaded form.

External Examiner
What is DMA Read and DMA Write Cycle?
Q: 113

You ~ Bharat Sir's Student
DMA Read: When DMAC transfers data from Memory to I/O.
DMA Write: When DMAC transfers data from I/O to Memory.

External Examiner
Explain DMAC Priority Modes?
Q: 114

You ~ Bharat Sir's Student
Fixed Priority: Channel 0 Highest… Channel 3 Lowest.
Rotating Priority: When a channel is serviced it gets lowest priority and others come up in the order in a circular manner.

External Examiner
Give some features of 80386?
Q: 115

You ~ Bharat Sir's Student
Operating frequency: 16 - 33 MHz
Address Bus: 32 bits , Physical Memory: 4GB
Data Bus: 32 bits , Memory Banks: 4
ALU: 32 bits, Classification: 32-bit microprocessor
Pipeline Stages: 3 stages – Fetch, Decode, Exec

External Examiner
Give some features of 80586 - Pentium?
Q: 116

You ~ Bharat Sir's Student
Operating frequency: 66 - 99 MHz
Address Bus: 32 bits, Physical Memory: 4GB
Data Bus: 64 bits , Memory Banks: 8 banks
ALU: 32 bits, Classification: 32-bit microprocessor
Pipeline Stages: 5 stages – Fetch, Decode, Operand Addr Calc, Exec, Write Back

External Examiner
What are the different types of Non – Volatile Memories?
Q: 117

You ~ Bharat Sir's Student
***ROM**: Read Only Memory. This is the original type of ROM. Cannot be written.*
***PROM**: Programmable ROM. Can be written but only once. Also called OTPROM.*
***EPROM**: Erasable PROM. Can be erased by exposure to UV rays.*
***EEPROM**: Electrically EPROM. Can be erased by applying higher voltage.*
***Flash ROM**. Special type of EEPROM. Allows block-wise erasure so erasure is faster.*

External Examiner
What are the different types of Volatile Memories?
Q: 118

You ~ Bharat Sir's Student
***RAM**: Random Access Memory. It is of two types.*
***SRAM**: Static RAM. Uses Flip-Flops to store data so works faster. Very Expensive.*
***DRAM**: Dynamic RAM. Uses capacitors to store data. Slower. Cheaper.*
***SDRAM**: Synchronous Dynamic RAM. Uses common clock with CPU to synchronize.*
***RDRAM**: Rambus RAM. Uses dedicated bus with CPU to transfer data..*

External Examiner
What memories are used in your mobile phone?
Q: 119

You ~ Bharat Sir's Student
It uses DRAM for primary memory, and Flash for secondary storage. Even the memory card is Flash ROM.

External Examiner
What memories are used in your computer?
Q: 120

You ~ Bharat Sir's Student
Primary Memory: DRAM ~ 4GB
Secondary Memory: Hard Disk ~ 1TB
Cache Memory: SRAM ~ 3MB
Portable Memory: CD – 700MB, DVD – 4.7GB. Pen Drives: Flash ROM of any size.