Dear students,

Welcome to the world of Microprocessors.

You are about to learn, how these little chips became the central force behind the computer revolution. Brain child of Alan Turing, nurtured by genius minds like Von Neumann, Maurice Wilkes, and materialized by corporations like Intel, Samsung, Apple… these chips transformed the way the world computed.

From traffic signals to air traffic control, drones to satellites, fitness bands to pace makers, Microprocessors have brought to you, ever improving standards of health, travel, entertainment, communication etc.

In this book, you begin with an introduction to the 8085 Microprocessor.
You then learn the 8086 Microprocessor in depth and then go on to explore new avenues introduced by several advanced processors all the way up to Pentium.
You also get a glimpse of interfaces with the keyboard, displays, ADCs and DACs.

Before we start, allow me to take the opportunity to thank my mother,
Prof. Veena D. Acharya.
A teacher all her life, she was the primary inspiration for me to pursue this noble profession.
Her blessings are reflected in the enthusiasm shown in this book and in the classroom lectures.

Bharat D. Acharya.
B. E. Computer Science.
Founder, Bharat Academy.
Teaching Microprocessors since 2000.

8085 MICROPROCESSOR

8086 MICROPROCESSOR

8259 PROGRAMMABLE INTERRUPT CONTROLLER

8255 PROGRAMMABLE PERIPHERAL INTERFACE

8254 PROGRAMMABLE INTERVAL TIMER

8237/ 8257 DMA CONTROLLER

MULTIPROCESSOR SYSTEMS

8086 DESIGNING

KEYBOARD AND LED DISPLAY (ETRX ONLY)

ADC DAC INTERFACING (EXTC ONLY)

ADVANCED MICROPROCESSORS (EXTC ONLY)

SYLLABUS AND QUESTION PAPERS

8085

MICROPROCESSOR

Important Features of 8085

1) Bus:

A bus is a **collection of lines**, which perform the same logical task.
The **size** of a bus **indicates** the **number of lines** in it, and **hence** the **number of bits** the bus can carry – as one line carries one bit of information.
There are three types of buses:

- **Address bus:**
  This bus **carries the address** of a particular location, for a transfer.
  It is very important to know that the size of the address bus decides the maximum number of locations the processor can address.
  i.e. n-bit address bus ➔ 2^n memory locations.
  8085 has a 16-bit address bus.
  ∴ It can access 2^{16} = 65536 memory locations ∴ **64 KB memory.**
  The memory **address range** is from **0000H … FFFFH**.

- **Data bus:**
  This bus **carries the data** to be transferred.
  8085 has an 8-bit data bus, which means it can transfer 8-bits in one operation and hence it is called as an **8-bit microprocessor**.

- **Control bus:**
  This bus **carries the control signals** that would cause any kind of an operation.
  The basic control signals are RD, WR etc.

Together these three buses are called as the **SYSTEM BUS**.

2) It can generate **8-bit IO Address** hence it can access 2^8 I/O Ports i.e. 256 I/O Ports. The **I/O Address** ranges from **00H … FFH.**

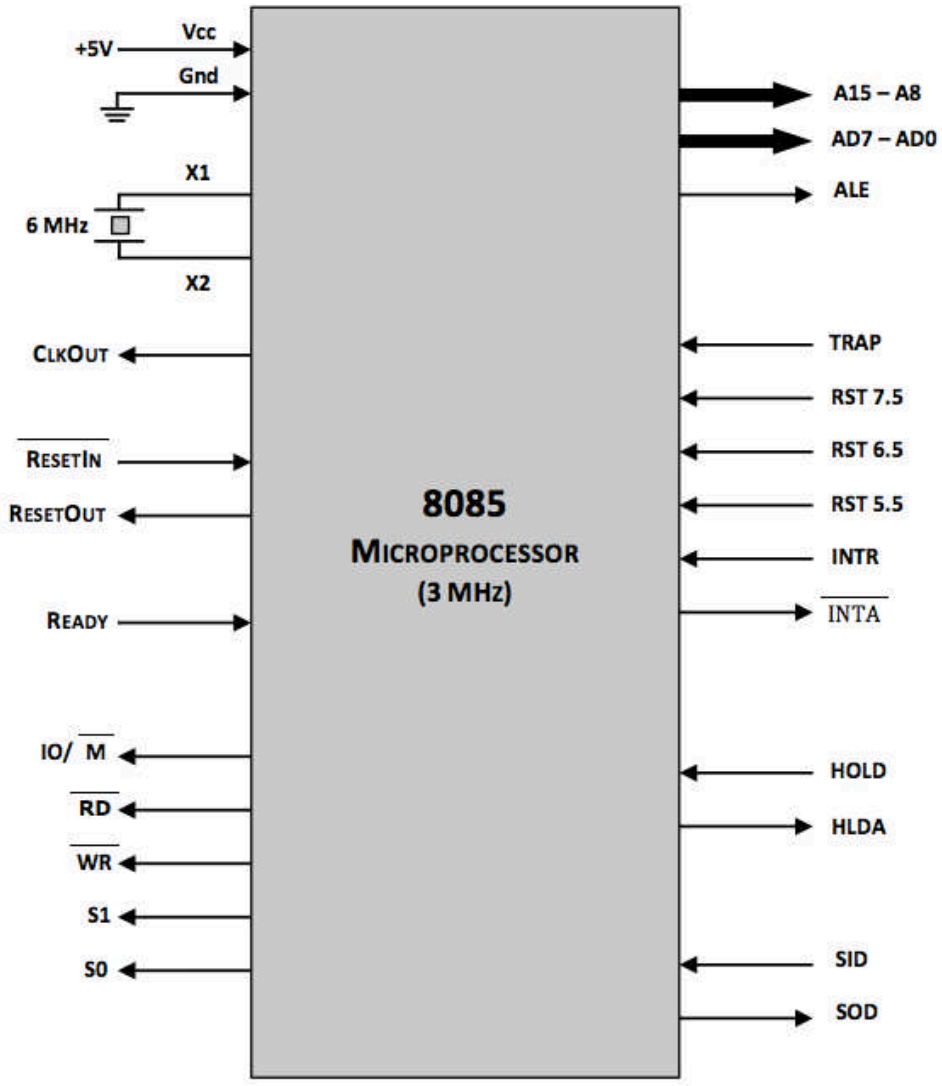
3) It works on +5 V power supply.

4) It provides 74 different instructions.

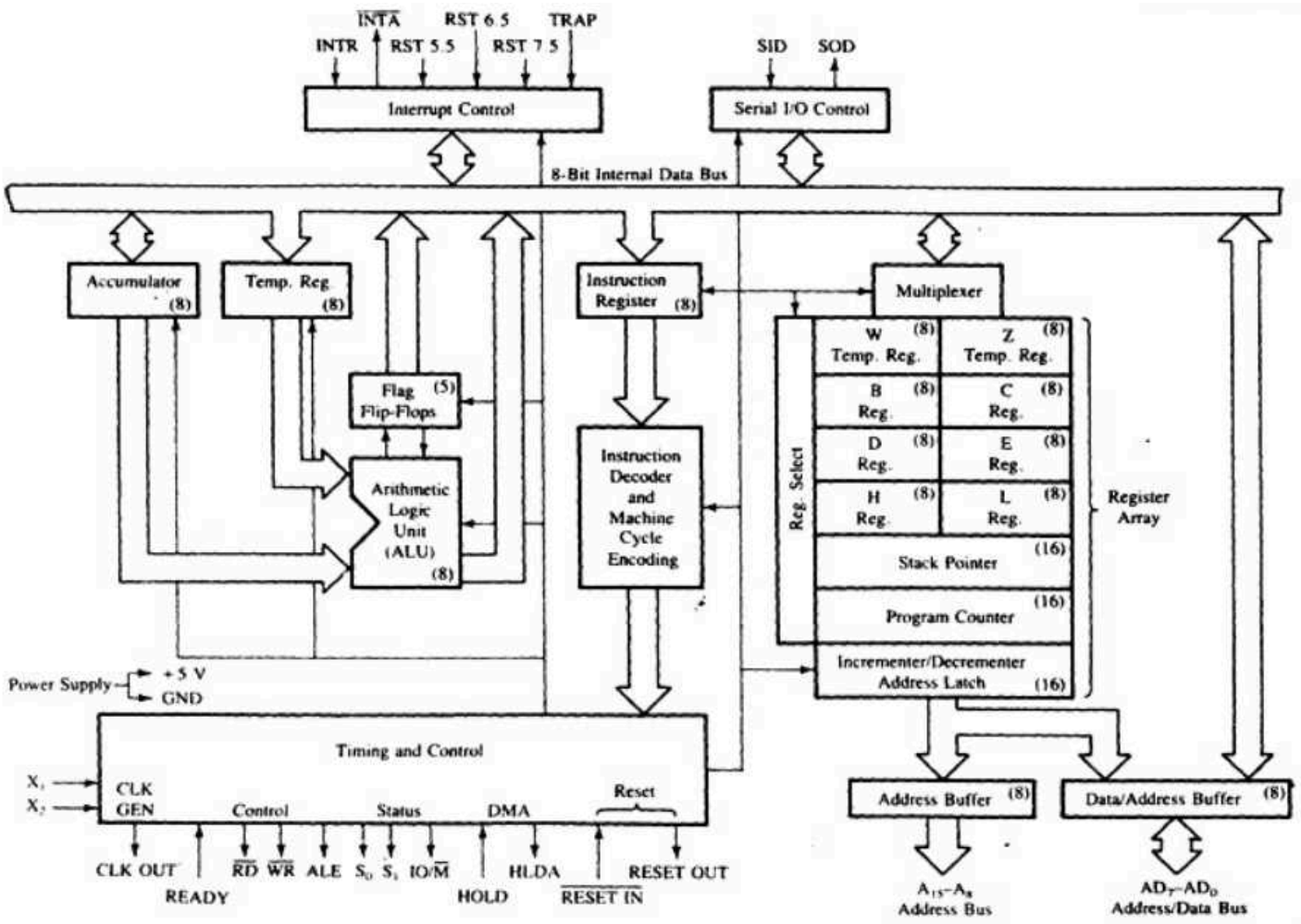
5) It has 40 external pins.

6) It accepts **5 external hardware interrupts**.

PIN CONFIGURATION OF 8085



ARCHITECTURE OF 8085



Registers

Program Counter (PC, 16-bits):

It is a 16-bit Special-Purpose register. It holds **address** of the **next instruction**.
PC is incremented by the INR/DCR after every instruction byte is fetched.

Stack Pointer (SP, 16-bits):

It is a 16-bit Special-Purpose register. It holds **address** of the **top of the Stack**.
Stack is a set of memory locations operating in LIFO manner.
SP is **decremented** on every **PUSH** operation and **incremented** on every **POP**.

B, C, D, E, H, L registers 8-bits each:

These are 8-bit General-Purpose registers**.**
They can also be used to store 16-bit data in register pairs.
The possible register **pairs** are **BC** pair, **DE** pair and **HL** pair.
The **HL** pair also holds the **address** for the Memory Pointer **"M"**.

Temporary Registers (WZ, 16-bits):

This is a 16-bit register pair.
It is **used by µP** to hold **temporary** values in some instructions like CALL/JMP etc.
The **programmer** has **no access** to this register pair.

INR/DCR Register (16-bits):

This is a 16-bit shift register.
It is used to **increment PC after every instruction byte is fetched** and **increment** or **decrement SP** after a Pop or a Push operation respectively.
It is not available to the programmer.

A - Accumulator (8-bits):

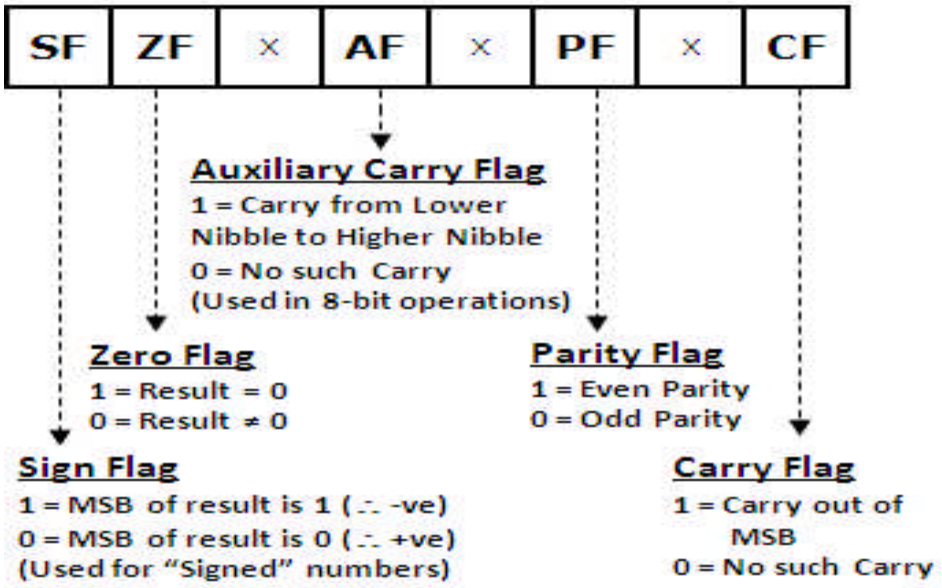
It is an 8-bit programmable register.
The user can read or write this register.
It has two **special properties**:

- It **holds one** of the **operands** during most of the arithmetic operations.
- It **holds** the **result** of most of the arithmetic and logic operations

Temp Register (8-bits):

This is an 8-bit register.
It is **used by µP** for storing one of the operands during an operation.
The **programmer** has **NO ACCESS** to this register.

FLAG REGISTER OF 8085



S - Sign Flag:

It is **set** (1) when **MSB** of the result is **1** (i.e. result is a **-**VE number).
It is reset (0) when MSB of the result is 0 (i.e. result is a **+**VE number).

Z - Zero Flag:

It is **set** when the **result** is **= zero**.
It is reset when the result is not = zero.

AC - Auxiliary Carry Flag:

It is **set** when an **Auxiliary Carry** / Borrow is **generated**.
It is reset when an Auxiliary Carry / Borrow is not generated.
Auxiliary Carry is the Carry generated **between** the **lower nibble and the higher nibble** for an 8-bit operation. It is not affected after a 16- bit operation. It is used only in DAA operation.

P - Parity Flag:

It is **set (1)** when result has **even parity**. It is reset when result has odd parity.

C - Carry Flag:

It is **set** when a **Carry / Borrow** is **generated from the MSB**.
It is reset when a Carry / Borrow is not generated from the MSB.

 # In the exam, Show at least 2 examples from Bharat Sir's lecture notes

INTERRUPTS OF 8085

This Block is responsible for controlling the **hardware interrupts** of 8085.
8085 supports the following hardware interrupts:

TRAP:

This is an **edge as well as level triggered**, **vectored** interrupt.
It cannot be masked by SIM instruction and can neither be disabled by DI instruction.
It has the **highest priority**.
Its vector address is **0024H**.

RST 7.5:

This is an **edge triggered**, **vectored** interrupt.
It can be masked by SIM instruction and can also be disabled by DI instruction.
It has the **second highest priority**.
Its vector address is **003CH**.

RST 6.5:

This is a **level triggered**, **vectored** interrupt.
It can be masked by SIM instruction and can also be disabled by DI instruction.
It has the **third highest priority**.
Its vector address is **0034H**.

RST 5.5:

This is a **level triggered**, **vectored** interrupt.
It can be masked by SIM instruction and can also be disabled by DI instruction.
It has the **fourth highest priority**.
Its vector address is **002CH**.

INTR:

This is a **level triggered**, **non-vectored** interrupt.
It cannot be masked by SIM instruction but can be disabled by DI instruction.
It has the **lowest priority**.
It has an **acknowledgement signal $\overline{\text{INTA}}$** .
The address for the ISR is **fetched from external hardware**.

$\overline{\text{INTA}}$:

This is an **acknowledgement** signal **for INTR** (only).
This signal is used to **get** the Op-Code (and hence the ISR address) from External hardware in order to execute the ISR. ☺ In case of doubts, contact Bharat Sir: - 98204 08217.

ALL Interrupts **EXCEPT TRAP** can be **disabled** though the **DI** instruction.
These interrupts can be **enabled** again by the **EI** Instruction.
Interrupts can be individually **masked or unmasked by SIM instruction**.
TRAP and INTR are not affected by SIM instruction.

SERIAL CONTROL

This Block is responsible for transferring data Serially to and from the µP.

SID - Serial In Data:
µP receives data, bit-by-bit through this line.

SOD - Serial Out Data:
µP sends out data, bit-by-bit through this line.
Serial transmission can be done by **RIM** and **SIM** Instructions.

ALU – ARITHMETIC LOGIC UNIT

8085 has an **8-bit ALU**.
It performs 8-bit arithmetic operations like Addition and Subtraction.
It also performs logical operations like AND, OR, EX-OR NOT etc.
It takes **input** from the **Accumulator** and the **Temp** register.
The **output** of most of the ALU operations is stored back **into** the **Accumulator**.

INSTRUCTION REGISTER AND DECODER

Instruction Register:
The 8085 places the contents of the PC onto the Address bus and fetches the instruction.
This fetched instruction is stored into the Instruction register.

Instruction Decoder:
The fetched instruction from the Instruction register enters the Instruction Decoder.
Here the instruction is decoded and the decode information is given to the Timing and Control Circuit where the instruction is executed..

TIMING AND CONTROL CIRCUIT

The timing and control circuit issues the various internal and external control signals for executing and instruction.
The external pins connected to this circuit are as follows:

X1 and X2:
These pins provide the **Clock Input to the µP**.
Clock is provided from a crystal oscillator.

ClkOut:
8085 provides the **Clock input** to all **other peripherals** through the ClockOut pin.
This takes care of **synchronizing** all peripherals with 8085.

$\overline{\text{ResetIn}}$:
This is an active low signal activated when the manual reset signal is applied to the µP. This signal **resets the µP**. On Reset PC contains **0000H**. Hence, the **Reset Vector Address** of 8085 is 0000H.

ResetOut:

This signal is connected to the reset input of all the peripherals.
It is used to **reset the peripherals once** the **µP** is **reset**.

READY:

This is an active high input.
It is used to **synchronize** the µP with **"Slower"** Peripherals.
The **µP samples** the **Ready** input in the beginning of every Machine Cycle.
If it is found to be **LOW**, the **µP executes** one **WAIT CYCLE** after which it re-samples the ready pin till it finds the Ready pin HIGH.
∴ The **µP remains** in the **WAIT STATE until** the **READY** pin becomes **high** again.
Hence, **if** the **Ready** pin is **not required** it should be **connected** to the **Vcc,** and not, left unconnected, **otherwise** would cause the µP to execute **infinite wait cycles**. *#Please refer Bharat Sir's Lecture Notes for this ...*

ALE - Address Latch Enable:

This signal is **used to latch address** from the multiplexed Address-Data Bus (**AD0-AD7**). When the Bus contains **address**, **ALE** is **high**, **else** it is **low**.

IO/ $\overline{M}$:

This signal is used to distinguish between an **IO** and a **Memory operation**.
When this signal is high it is an IO operation else it is a Memory operation.

$\overline{RD}$:

This is an active low signal used to indicate a **read operation**.

$\overline{WR}$:

This is an active low signal used to indicate a **write operation**.

S_1 and S_0:

These lines denote the status of the µP

S1	S0	STATUS
0	0	Idle
0	1	Write
1	0	Read
1	1	Opcode fetch

HOLD and HLDA:

The Hold and Hold Acknowledge signals are used for **Direct Memory Access** (DMA).
The **DMA Controller issued** the **Hold** signal to the µP.
In response the **µP releases** the **System bus**.
After releasing the system bus the **µP acknowledges** the Hold signal with **HLDA** signal.
The **DMA Transfer** thus **begins**.
DMA **Transfer** is **terminated** by **releasing** the **HOLD** signal.

*Note: **"Programmer's Model"** simply means all registers in the architecture that are available to the programmer. So if they ask Programmers model, draw the internal part of the architecture without the pins, and explain all registers.*

8086

MICROPROCESSOR

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.

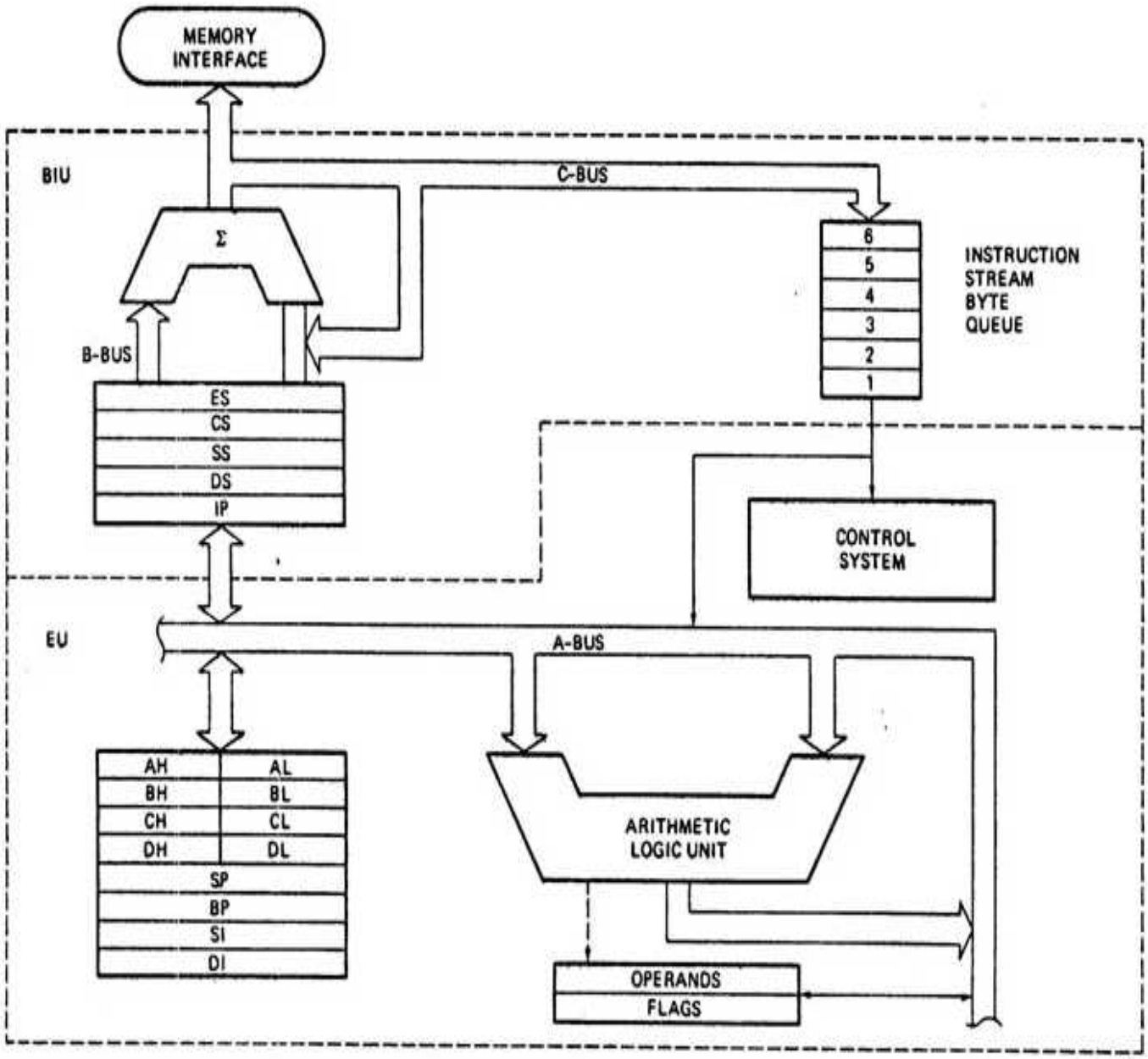
4) 8086 provides **Memory Banks**.
The entire memory of 1 MB is **divided into 2 banks of 512KB each**, in order to transfer 16-bits in 1 cycle. The banks are called **Lower Bank** (even) **and Higher Bank** (odd).

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:

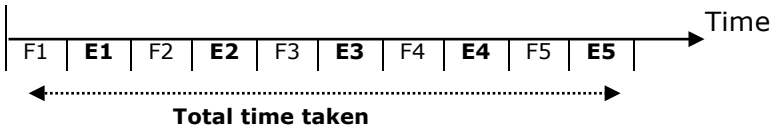
Physical address = Segment Address x 10h + Offset Address

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.

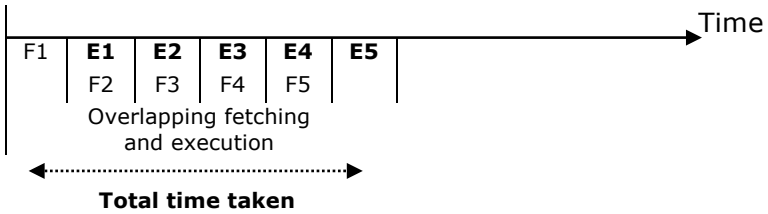
d) 6-Byte Pre-Fetch Queue {Pipelining – 4m}

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.**

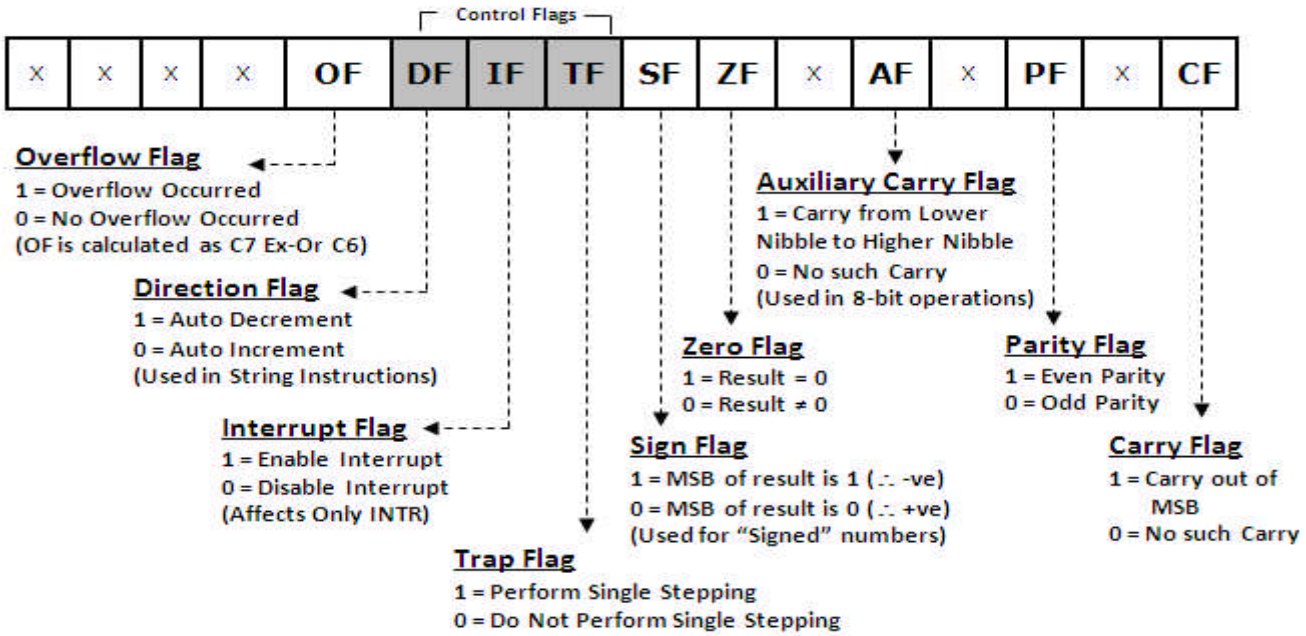
c) ALU (16-Bits)
It has a **16-bit ALU**. It performs 8 and 16-bit arithmetic and logic operations.

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 …

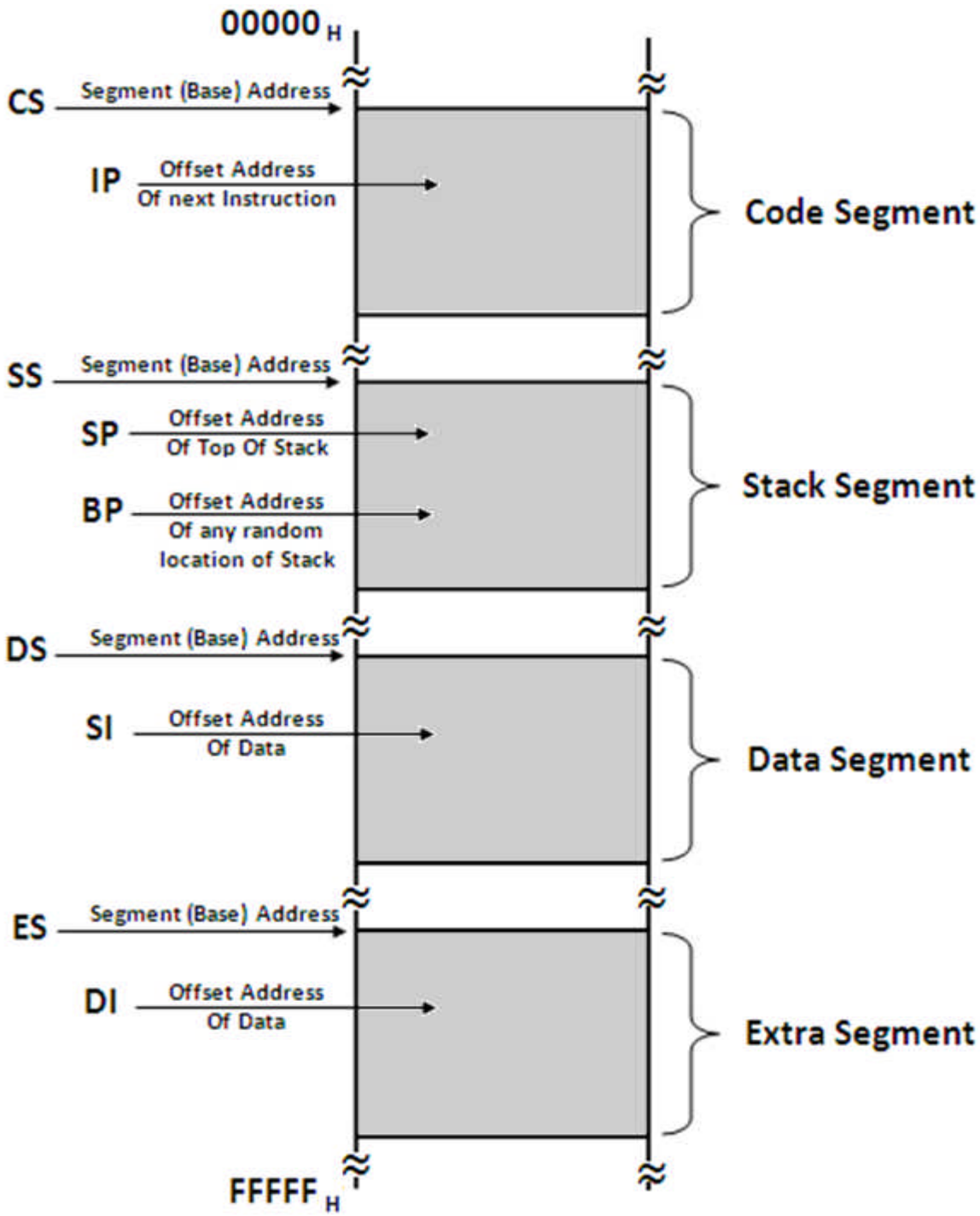
CONTROL FLAGS

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 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**.

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 $\overline{\text{BHE}}$ = 0

Even Bank

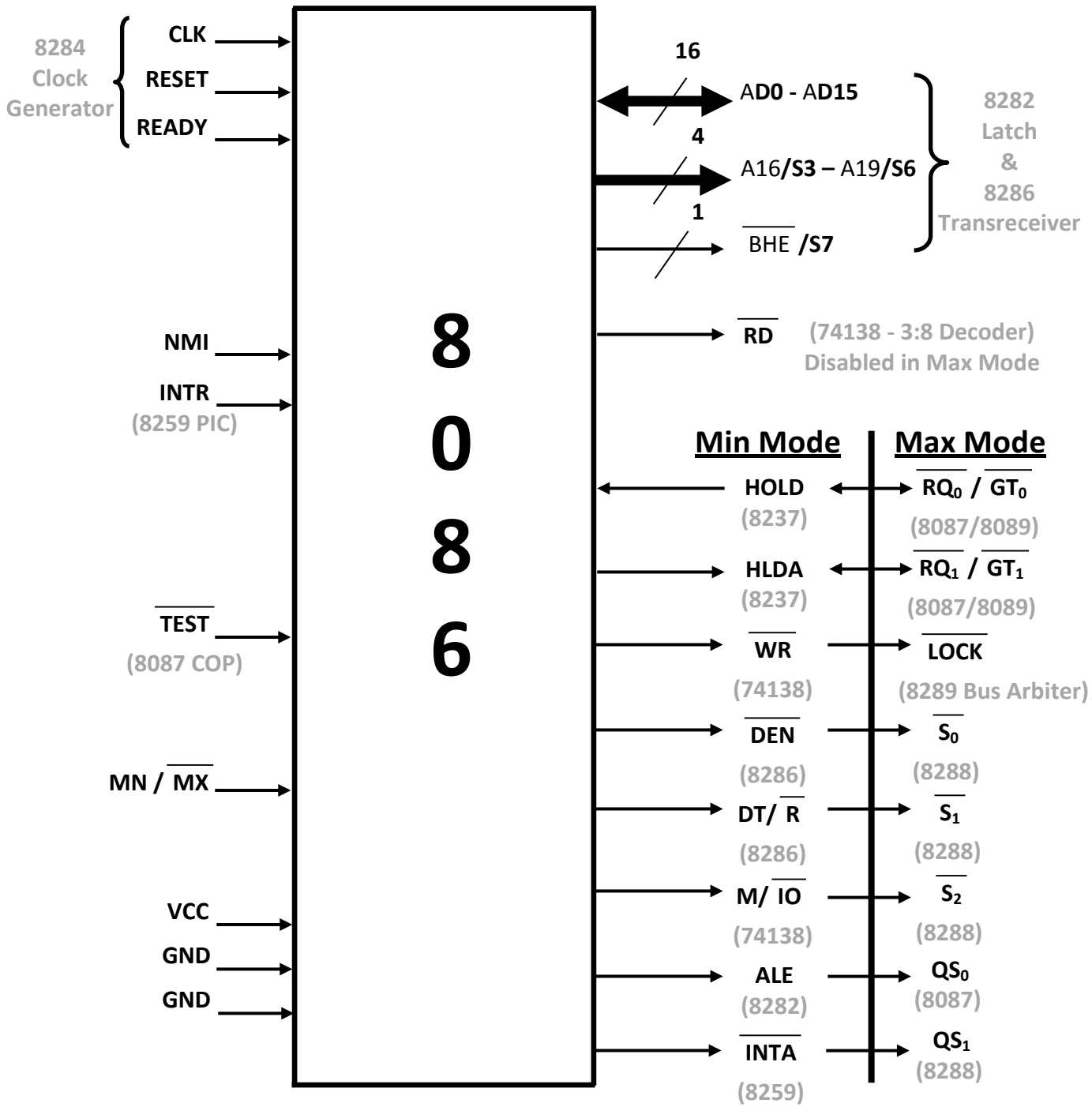
- Also called as "Lower bank"
- Address range:

00000H
00002H
00004H
.
.
.
FFFFEH

- Selected when A_0 = 0

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

PIN DIAGRAM OF 8086



Machine Cycles

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

Segment Selection

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

Queue Synchronization

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

PIN DESCRIPTIONS

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

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

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

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

- **MN/ $\overline{\text{MX}}$**
  This is an **input** signal to 8086.
  If this signal is **HIGH,** 8086 is in **Minimum** mode i.e. Uni-Processor system.
  If this signal is **Low,** 8086 is in **Maximum** mode i.e. Multiprocessor system.

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

- ## INTR

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

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

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

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

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

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

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

- ## VCC and GND

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

- ## AD15 - AD0

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

- ## A16/S3 - A19/S6

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

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

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

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

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

- **HOLD --- $\overline{RQ}_0 / \overline{GT}_0$**

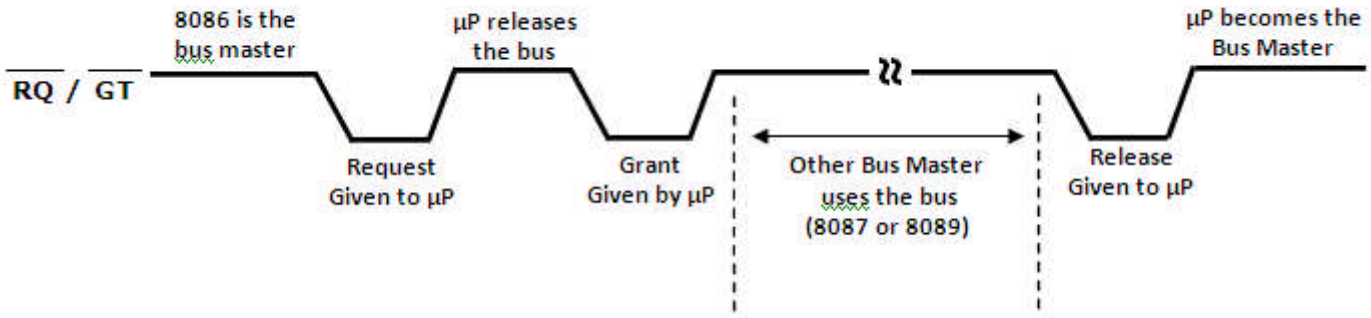
  In **Minimum** Mode this line carries the **HOLD** input signal.
  The **DMA Controller issues** the HOLD signal to request for the system bus.
  In response 8086 completes the current bus cycle and releases the system bus.

  *In **Maximum** Mode it carries the bi-directional $\overline{RQ}_0 / \overline{GT}_0$ signal (Request/Grant).The **external bus master** (eg: 8087) **sends an active low pulse** to request for the sys bus.*
  *In **response** the **8086** completes the current bus cycle, releases the system bus and **sends an active low Grant pulse** on the same line to the external bus controller.*
  ***8086 gets back** the system bus only after external **bus master sends an active low release pulse** on the same line.*

- **HLDA --- $\overline{RQ}_1 / \overline{GT}_1$**

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

  *In **Maximum** Mode it functions as $\overline{RQ}_1 / \overline{GT}_1$, which is the same as $\overline{RQ}_0 / \overline{GT}_0$ but is of **lower priority**.*



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

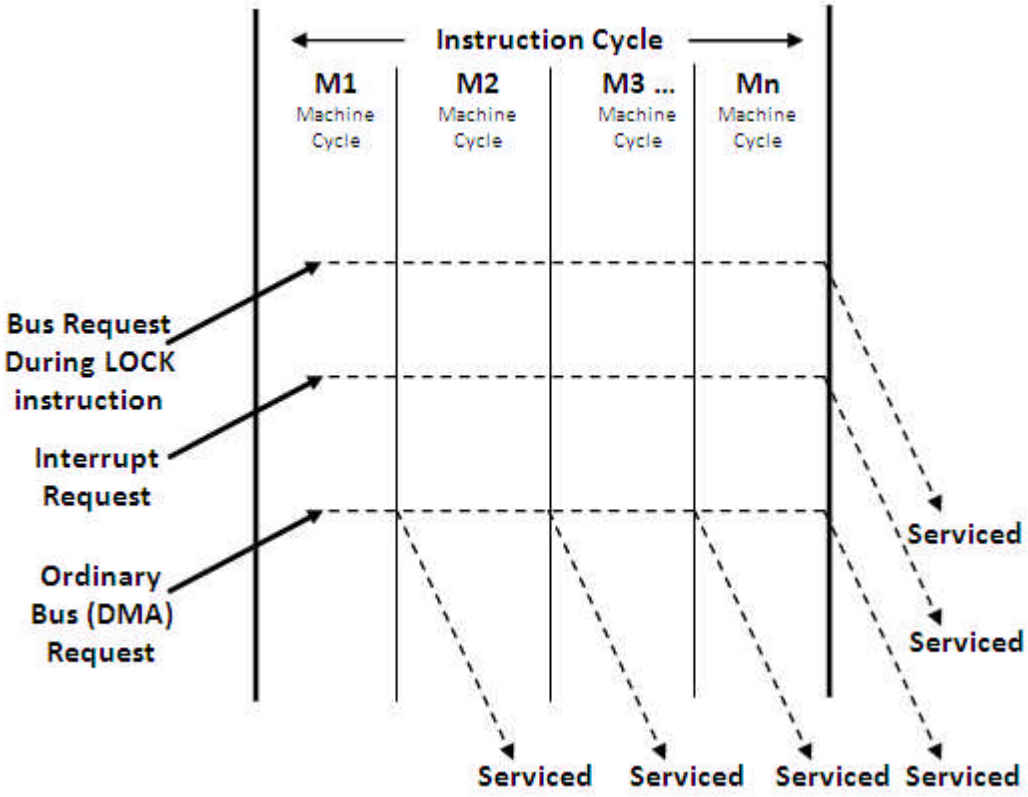
In **Minimum** Mode this line carries the **$\overline{WR}$** signal.

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

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

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

***LOCK Prefix**: Normally a bus request is serviced after the current machine cycle and an interrupt request is serviced after the current instruction cycle.*
But if we write LOCK prefix before any instruction, then even if there is a bus request, the bus will be released only after the current instruction. Hence the bus is said to be locked during the instruction.
*µP will maintain **$\overline{LOCK}$** signal low throughout the instruction to indicate that it is performing an instruction with LOCK prefix. **$\overline{LOCK}$** signal is given to 8289 Bus Arbiter in Loosely Coupled Systems, to prevent 8289 from releasing the system bus to other bus masters.*



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

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

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

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

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

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

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

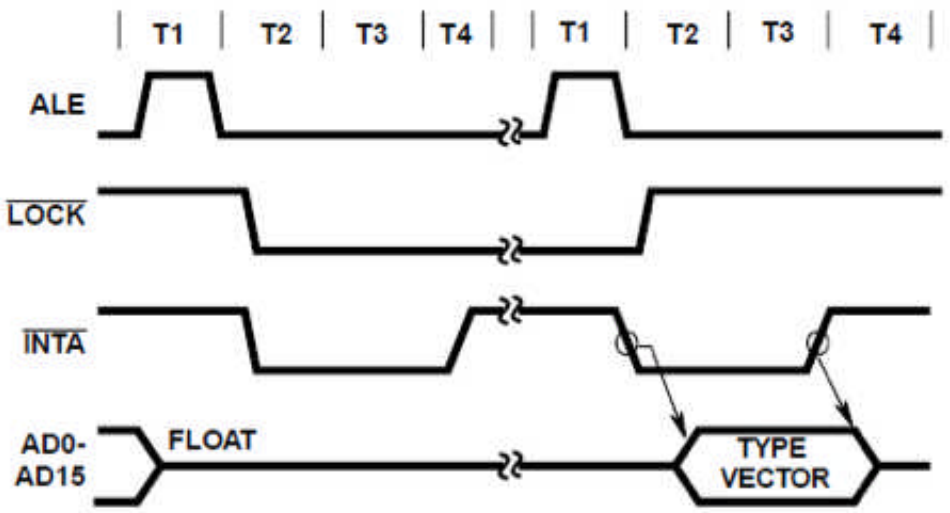
M/ $\overline{IO}$	$\overline{RD}$	$\overline{WR}$	Action
1	0	1	Memory Read
1	1	0	Memory Write
0	0	1	I/O Read
0	1	0	I/O Write

- **ALE --- QS0**
  In **Minimum** Mode it carries the **ALE** signal, which is used to latch the address.
  *In **Maximum** Mode it carries the **QS_0** signal.*
  *It is used with **QS_1** to indicate the Instruction Queue Status.*
  ***In Maximum Mode, Bus Controller gives the ALE signal**.*

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

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

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



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

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 * 10H + 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.*

ADDRESSING MODES FOR PROGRAM MEMORY

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

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

INTRA SEGMENT DIRECT ADDRESSING MODE

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

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

                  Or

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

INTER SEGMENT DIRECT ADDRESSING MODE

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

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

INTRA SEGMENT INDIRECT ADDRESSING MODE

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

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

INTER SEGMENT INDIRECT ADDRESSING MODE

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

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

ADDRESSING MODES FOR STACK MEMORY (optional, to be written only if asked)

REGISTER ADDRESSING MODE

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

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

REGISTER INDIRECT ADDRESSING MODE

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

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

1) Flag Addressing Mode

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

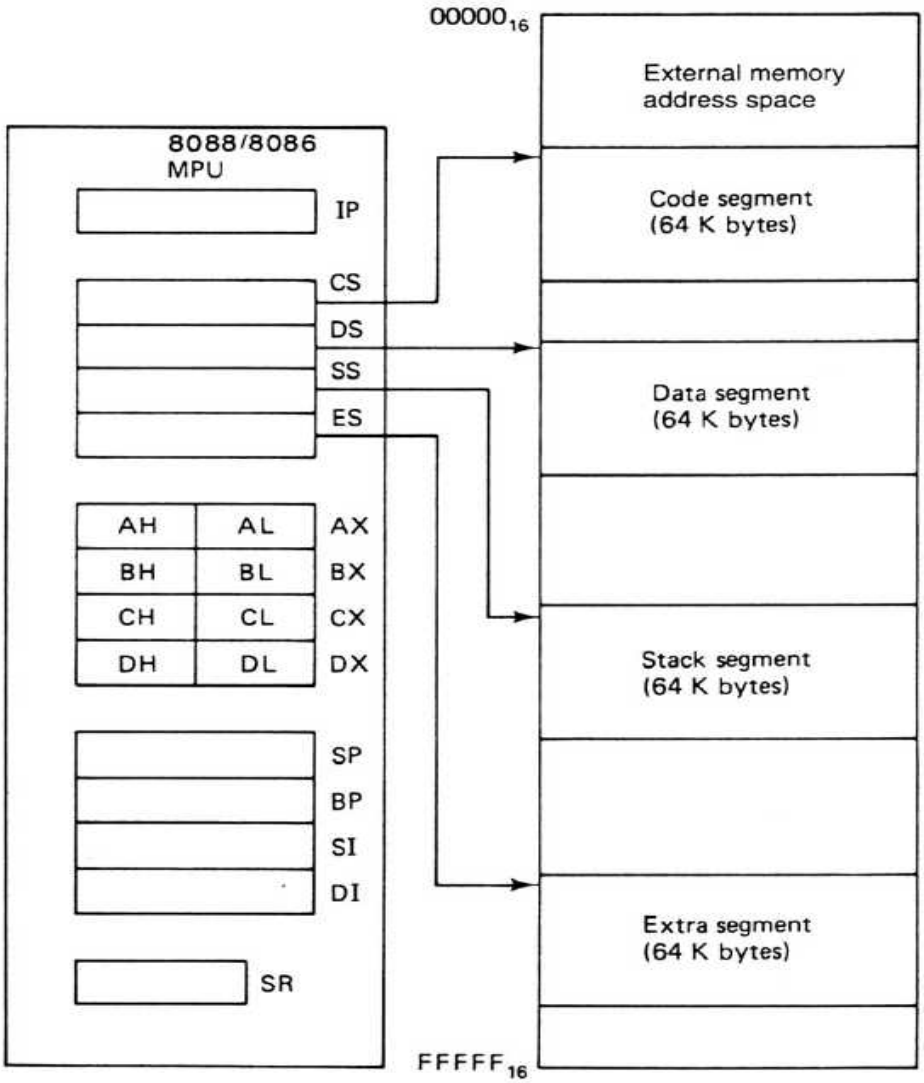
Eg: PUSHF *; Transfers higher byte of Flag register to SP-1*
; and lower byte to SP-2 in the Stack Segment.

2) Segment Register Addressing Mode

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

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

8086 SOFTWARE MODEL



Dear Students,

*Software Model, also called **Programmers model,** Means all the registers available to the programmer. So if they ask Programmers Model, draw the above diagram and explain all the registers from the architecture answer. It must include **all GPRs: AX, BX, CX, DX. Segment Registers: CS, SS, DS, ES. All Offset Registers: IP, SP, BP, SI, DI and Flag Register**. Additionally give a very brief note on segmentation.*

-Bharat Sir.

In case of doubts, just call me on my cell phone… 9820408217.

8086 INSTRUCTION SET

CLASSIFICATION OF INSTRUCTION SET OF 8086

1) **DATA TRANSFER INSTRUCTIONS**
   E.g.:: MOV, PUSH, POP
2) **ARITHMETIC INSTRUCTIONS**
   E.g.:: ADD, SUB, MUL
3) **LOGIC INSTRUCTIONS (BIT MANIPULATION INSTRUCTIONS)**
   E.g.:: AND, OR, XOR
4) **SHIFT INSTRUCTIONS & ROTATE INSTRUCTIONS**
   E.g.:: ROL, RCL, ROR, SHL
5) **PROGRAM EXECUTION AND TRANSFER INSTRUCTIONS (BRANCH INSTRUCTIONS)**
   E.g.:: JMP, CALL, JC
6) **ITERATION CONTROL INSTRUCTIONS (LOOP INSTRUCTIONS)**
   E.g.:: LOOP, LOOPZ, LOOPNE
7) **PROCESSOR CONTROL INSTRUCTIONS (INSTRUCTIONS OPERATING ON FLAGS)**
   E.g.:: STC, CLC, CMC
8) **EXTERNAL HARDWARE SYNCHRONIZATION INSTRUCTIONS**
   E.g.:: LOCK, ESC, WAIT
9) **INTERRUPT CONTROL INSTRUCTIONS**
   E.g.:: INT n, IRET, INTO (Interrupt on overflow)
10) **STRING INSTRUCTIONS**
    E.g.:: MOVSB, LODSB, STOSB

Dear Students,

Instruction Set was asked in a recent paper as a theory question.
If asked again, Give two instructions under each of the above headings as an example.
There are umpteen examples in the following pages.
Choose your favorites and write them.
Bharat Sir.

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
```

Segment Overriding

In every instruction, a particular segment register is accessed for the base address.
Eg: **MOV CL, [5000H]** *; CL ← DS:[5000H] as Data Seg is accessed by default*

However, we can also **override the segment** as follows:
Eg: **MOV CL, CS:[5000H]** *; Here CL ← CS:[5000H], this is **Segment Overriding**.*

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

Another example:

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

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.
$\therefore$ 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.

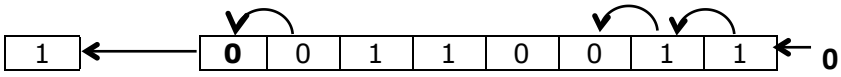
Destination: Register, Memory Location. #Please refer Bharat Sir's Lecture Notes for this ...

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

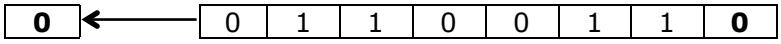
Assume:

Before Operation: BL = 0011 0011 and CF = 1

Carry **Destination**



After Operation: BL = 0110 0110 and CF = 0



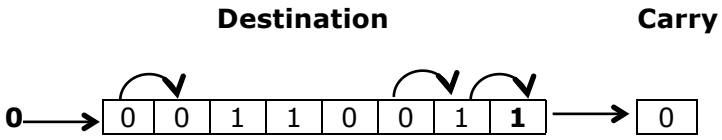
More examples:
MOV CL, 05H *; Load number of shifts in CL register.*
SAL BL, CL *; Left-Shift BL bits CL (5) number of times.*

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

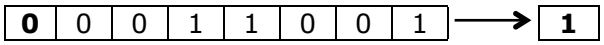
Bits are shifted 'count' number of times.
If count is 1, it is directly specified in the instruction.
If count > 1, it has to be given using CL register.
Eg: **SHR BL, 1** *; Right-Shift BL bits, once.*

Assume:

Before Operation: BL = 0011 0011 and CF = 0



After Operation: BL = 00011 1001 and CF = 1

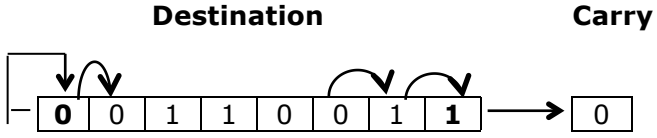


3) SAR destination, count
RIGHT-Shifts the bits of destination.
MSB placed **in MSB itself** (∴ Sign is preserved).
LSB shifted **into** the **CARRY**.

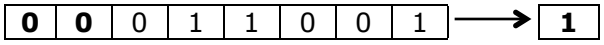
Bits are shifted 'count' number of times.
If count is 1, it is directly specified in the instruction.
If count > 1 it has to be given using CL register. ☺ For doubts contact Bharat Sir on 98204 08217
Destination: Register, Memory Location

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

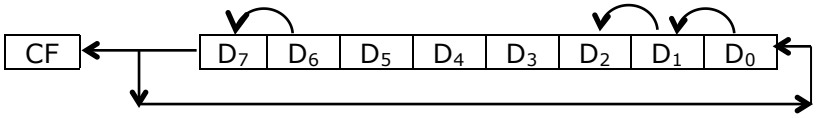


ROTATE INSTRUCTIONS

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**

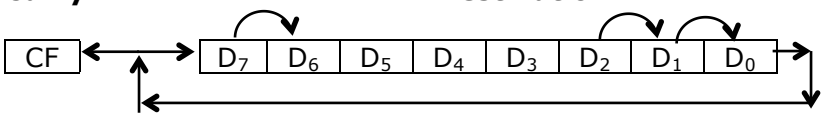


More examples:
MOV CL, 05H *; Load number of shifts in CL register.*
ROL BL, CL *; Left-Shift BL bits CL (5) number of times.*

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**

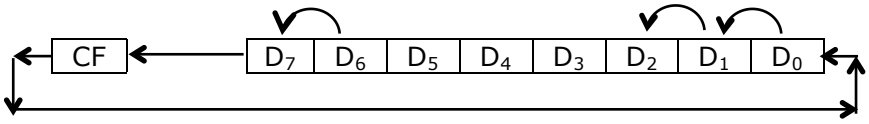


3) RCL destination, count

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.*

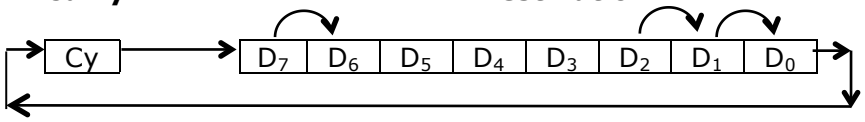
Carry **Destination**



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 2) **Processor Control / Machine Control Instructions**
(these are instructions that directly operate on Flag Reg)

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

For Carry Flag

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

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

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

For Direction Flag

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

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

For Interrupt Enable Flag

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

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

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

To set TF:

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

To reset TF:

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

Type 3) # External Hardware Synchronization Instructions

1) ESC

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

2) WAIT

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

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

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

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

i. **$\overline{\textbf{TEST}}$ input is made low**
i.e. 8087 is no longer busy.
This takes 8086 completely out of the IDLE state.

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

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

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

3) LOCK

This is an 8086 **instruction prefix.**

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

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

4) NOP

There is **no operation performed** while executing this instruction.
8086 requires 3 T-States for this instruction.
It is mainly used to insert time delays, and can also be used while debugging.

5) HLT

This instruction causes 8086 to **stop fetching any more instructions**.
8086 enters **Halt state**.
8086 can **come out** of this halt state only if there is a **valid hardware interrupt** (NMI or INTR) or **by reset**.

Type 4) # Interrupt Control Instructions

1) INT Type

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

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

2) INTO (Interrupt on Overflow)

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

3) IRET (Return from ISR)

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

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

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

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

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

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

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

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

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

INSTRUCTION FORMAT TEMPLATE OF 8086

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

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

Opcode Byte

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

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

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

Addressing Mode Byte

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

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

reg (3 bits):

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

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

r/m (3 bits):

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

R/M	MOD				
	00	01	10	11	
				W=0	W=1
000	[BX]+[SI]	[BX]+[SI]+d8	[BX]+[SI]+d16	AL	AX
001	[BX]+[DI]	[BX]+[DI]+d8	[BX]+[DI]+d16	CL	CX
010	[BP]+[SI]	[BP]+[SI]+d8	[BP]+[SI]+d16	DL	DX
011	[BP]+[DI]	[BP]+[DI]+d8	[BP]+[DI]+d16	BL	BX
100	[SI]	[SI]+d8	[SI]+d16	AH	SP
101	[DI]	[DI]+d8	[DI]+d16	CH	BP
110	16-bit address	[BP]+d8	[BP]+d16	DH	SI
111	[BX]	[BX]+d8	[BX]+d16	BH	DI

ASSEMBLER DIRECTIVES / PSEUDO OPCODES

Assembly language has 2 types of statements:

1. **Executable**: Instructions that are translated into Machine Code by the assembler.

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

1. DB (Define Byte) ; Used to define a Byte type variable.
Eg: SUM DB 0 ; Assembler reserves 1 Byte of memory for the variable
; named SUM and initialize it to 0.

2. DW (Define Word) ; Used to define a Word type variable (2 Bytes).

3. DD (Double Word) ; Used to define a Double Word type variable (4 Bytes).

4. DQ (Quad Word) ; Used to define a Quad Word type variable (8 Bytes).

5. DT (Ten Bytes) ; Used to define 10 Bytes to a variable (10 Bytes).

6. DUP() ; Copies the contents of the bracket followed by this
; keyword into the memory location specified before it.
Eg: LIST DB 10 DUP (0) ; Stores LIST as a series of 10 bytes initialized to Zero.

7. SEGMENT ; Used to indicate the beginning of a segment.

8. ENDS ; Used to indicate the end of a segment.

9. ASSUME ; Associates a logical segment with a processor segment.
Eg: Assume CS:Code ; Makes the segment "Code" the actual Code Segment.

10. PROC ; Used to indicate the beginning of a procedure.

11. ENDP ; Used to indicate the end of a procedure.

12. END ; Used to indicate the end of a program.

13. EQU ; Defines a constant
E.g.:: AREA EQU 25H ; Creates a constant by the name AREA with a value 25H
Do remember, in the class, you have been clearly made to understand the difference between using a variable and using a constant.

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

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

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

17. Model Directives

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

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

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

Combined Example:

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

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

Code    ENDS
        END Start
```

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

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 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 5) **WAP to transfer a block of 10 bytes from Data Segment to Extra Segment**

```
Data  SEGMENT

      Block1 DB 05H,36H,01H,09H,78H,14H,96H,78H,15H,12H

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

      MOV   CX, 000AH

      CLD

REP   MOVSB

      INT3

Code ENDS

      END
```

Q 6) **WAP to invert a block of 10 bytes from Data Segment to Extra Segment.**

```
Data SEGMENT

      Block1 DB 00H,01H,02H,03H,04H,05H,06H,07H,08H,09H

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

      INT3

Code  ENDS

      END
```

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

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 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 12) **WAP to find the factorial of a number stored at 24,000H in data segment. Store the result at 24,001H and 24,002H.**

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

      MOV   BL, [4000H]
      MOV   AH, 00H
      MOV   AL, 01H

Back: MUL   BL
      DEC   BL
      JNZ   Back

      MOV   [4001H], AX

      INT3
Code ENDS
      END
```

Q 13) **WAP to Convert a Hex number into Decimal Number.**
Assume the hex number stored at 24000H.
Store the result at 24001H and 24002H.

Note:
Same question can be asked as **Convert a Binary number into BCD.**

Dear Students,
To understand the below mentioned program, with example, keep a calculator handy.
Try it with your own examples. You will always get the correct answer, trust me!
*Since **YOU** are familiar with **OUR** notes, just focus on the **"BOLD"** numbers*
In case of doubts, You know whom to call ;-)

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

      MOV   AL, [4000H] ; Assume number is 86H. Answer should be: 01 34
      MOV   AH, 00H     ; Now AX in our example is 0086H
      MOV   BL, 0AH     ; BL ← 0AH which means BL is 10.

      DIV   BL          ; AX÷BL i.e. 0086H ÷ 0AH gives 0DH (Quotient) in AL
                        ; And 04H (Remainder) in AH
      MOV CH, AH        ; The remainder in AH (04H) is the lowest nibble of the final answer.
                        ; Lets move it to CH for now

      MOV AH, 00H       ; Now AX = 000DH
      DIV BL            ; AX÷BL i.e. 000DH ÷ 0AH gives 01H (Quotient) in AL
                        ; And 03H (Remainder) in AH
                        ; The quotient in AL is the highest nibble of the answer
                        ; In our example it is 01H

      MOV [4002H], AL   ; Put this (01H) at [4002H]
      MOV AL, AH        ; Now AL gets 03H from AH

      MOV CL, 04H
      SHL AL, CL        ; Shift left AL 4 times, that's the same as Multiplying by 16.
                        ; So AL which was 03H becomes 30H

      ADD AL, CH        ; AL ← 30H + 04H = 34H

      MOV   [4001H], AL ; Store 34H at [4001H]
                        ; So finally [4002H] and [4001H] have the final
                        ; Result = 01 34

      INT3
Code ENDS
      END
```

Q 14) **WAP to Convert a Decimal Number into Hexadecimal.**
Assume the Decimal Number is stored at 24000H.
Store the result at 24001H.

Note:
Same question can be asked as **Convert a BCD number into a Binary.**

Dear Students,
To understand the below mentioned program, with example, keep a calculator handy.
Try it with your own examples. You will always get the correct answer, trust me!
*Since **YOU** are familiar with **OUR** notes, just focus on the **"BOLD"** numbers*
In case of doubts, You know whom to call ;-)

Assume the following example: Converted (67)D into (43)H

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

      MOV   AL, [4000H] ; Assume number is 67. Answer should be: 43
      AND   AL, 0F0H    ; AL will become 60H

      MOV   CL, 04H
      SHR   AL, CL      ; Shift right AL, 4 times. That's the same as Dividing AL by 16.
                        ; So AL which was 60H becomes 06H

      MOV   AH, 00H     ; Now AX becomes 0006H
      MOV   BL, 0AH     ; BL gets 0AH which means BL gets 10.

      MUL   BL          ; AX becomes 0006H x 0AH = 003CH
                        ; We are basically interested in AL which is now 3CH

      MOV   CL, [4000H] ; Again access the original number 67 from [4000H]
      AND   CL, 0FH     ; CL will become 07H

      ADD   AL, CL      ; AL will get 3CH + 07H = 43H ➔ Final result!

      MOV   [4001H], AL ; Store the final result 43H from AL to [4001H]


      INT3
Code ENDS
      END
```

Q 15) **WAP to determine how many times "e" exists in "exercise"**

```
Data  SEGMENT
      String      DB    "exercise"
      Count       DB    00H
Data  ENDS

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

      LEA   SI, String  ; Get the offset address of "String" into SI
      MOV   CX, 08H     ; Count of 8 as "exercise" has 8 characters
      MOV   BL, 00H     ; No of time "e" is encountered
      MOV   AL, "e"     ; Assembler substitutes the ASCII value of "e" in AL

Back: CMP   AL, [SI]    ; Compare AL (e) with every character of the string (exercise)
      JNZ   Skip        ; If the current character is not equal to "e" then skip
      INC   BL          ; As current character is equal to "e", increment the counter
Skip: INC   SI          ; In any case, increment the string pointer SI
      LOOP  Back        ; Decrement loop counter CX and keep looping till CX becomes 0

      MOV   Count, BL   ; Store the count from BL to variable "Count"

      INT3
Code ENDS
      END
```

PARAMETER PASSING TECHNIQUES USED IN ASSEMBLY LANGUAGE PROGRAMMING

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

1. USING REGISTERS

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

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

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

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

2. USING MEMORY LOCATIONS DIRECTLY

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

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

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

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

3. USING MEMORY LOCATIONS INDIRECTLY

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

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

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

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

4. USING STACK

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

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

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

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

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):

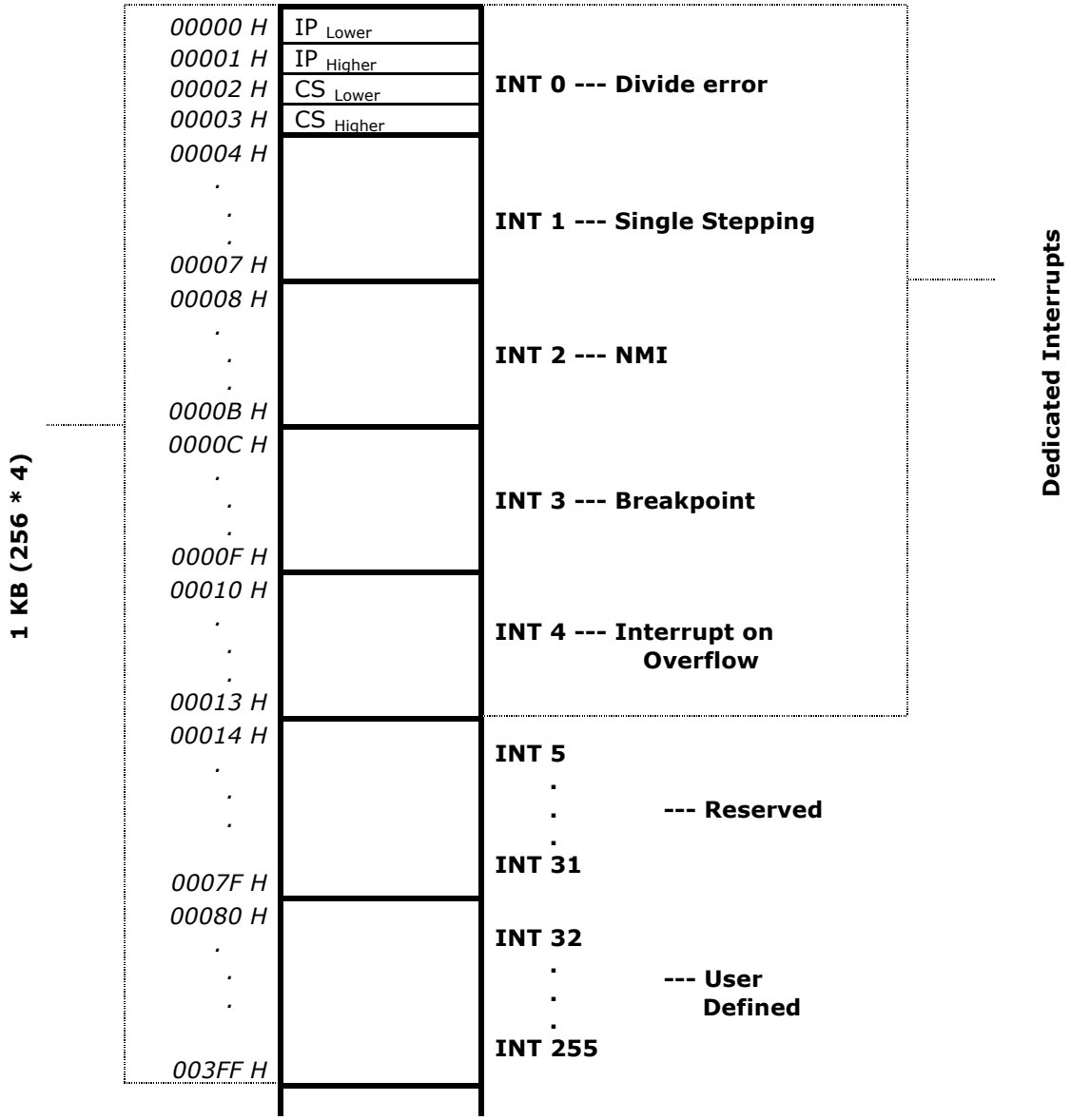
These interrupts are caused by writing the software interrupt instruction INTn where "n" can be any value from 0 to 255 (00H to FFH).
Hence all 256 interrupts can be invoked by software.

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.

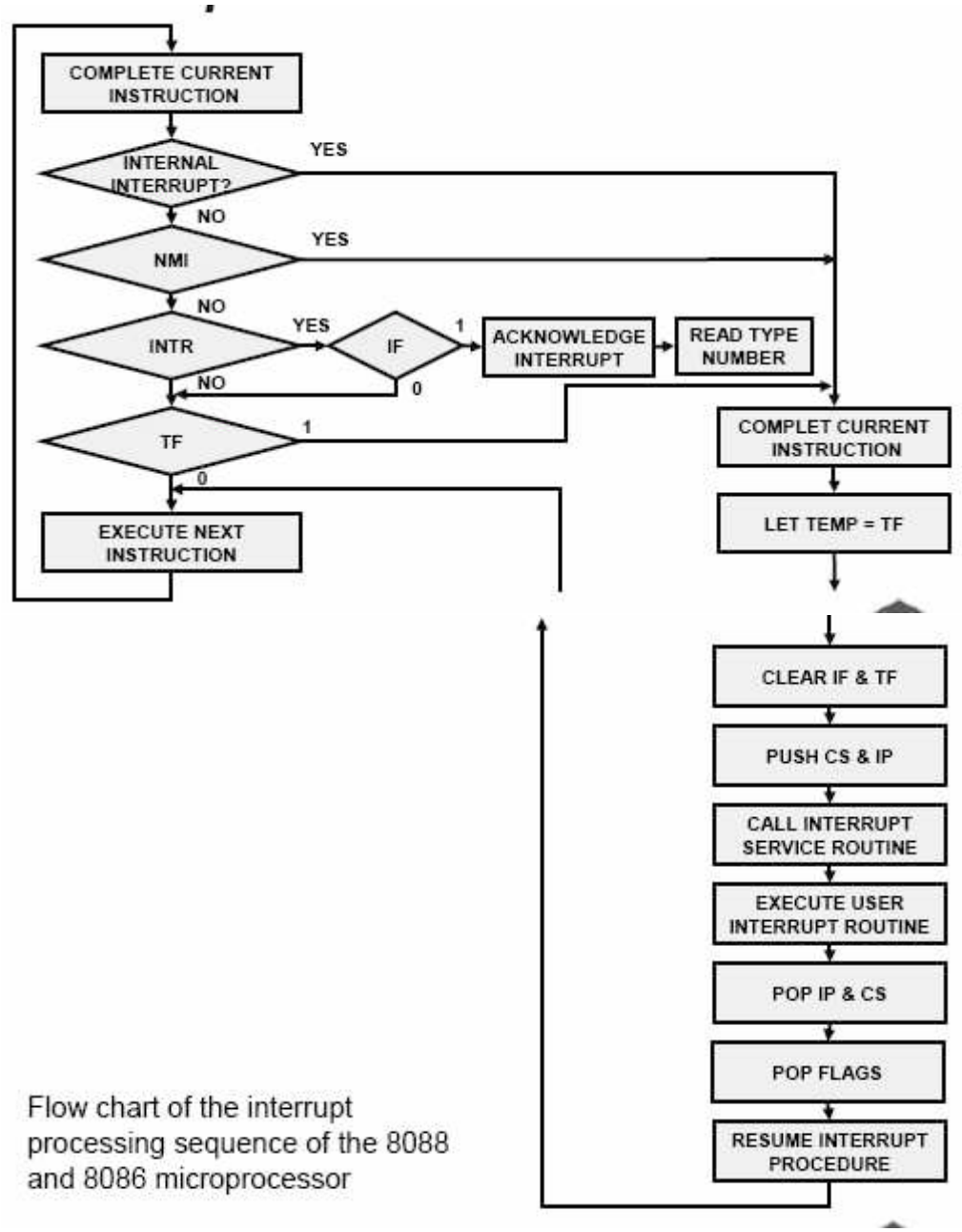
Eg: Assume the µP is executing a **DIV** instruction that causes a **division error** and **simultaneously INTR** occurs.
Here **INT 0** (Division error) will be **serviced first** i.e. its ISR will be executed, as it has higher priority, and **then INTR** will be **serviced**. #Please refer Bharat Sir's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

Interrupt priority Flowchart {Optional – Only for reference}



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

8086 CONFIGURATIONS

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

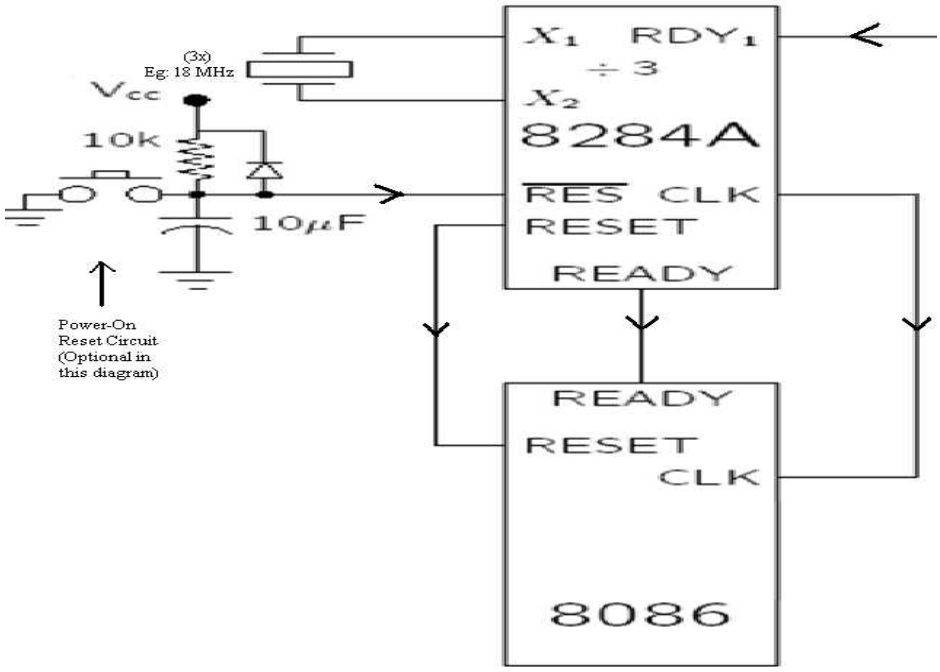
8282 – 8-bit (Octal) Latch

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

8286 – 8-bit Data Trans-receiver

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

8284 – Clock Generator

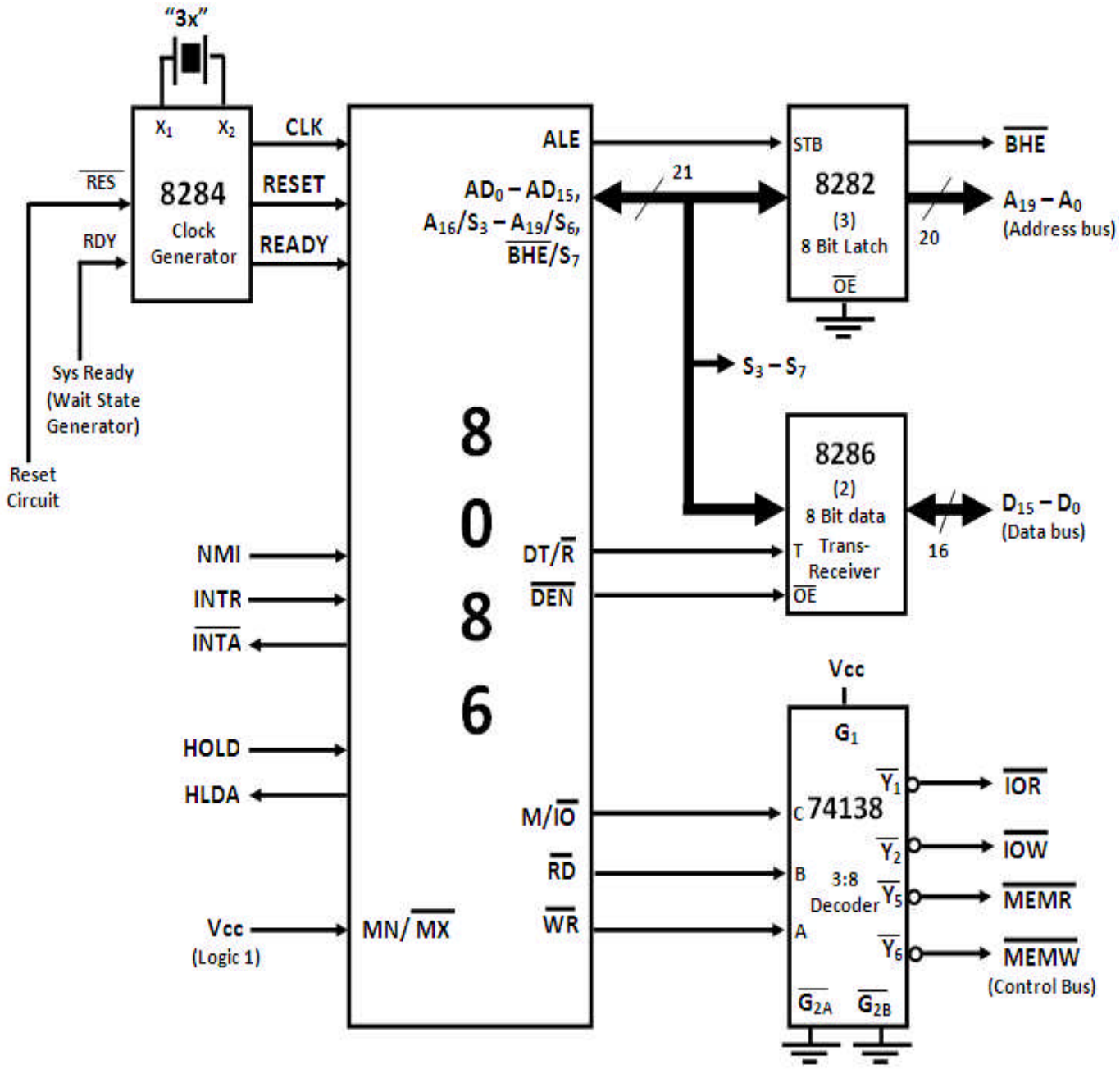


1) 8284 is a *Clock Generator* IC.
2) It provides the C **LOCK** (CLK) signal, a train of pulses at a constant freq, to the entire circuit.
3) It synchronizes the READY (**RD** Y) signal which indicates that an interface is ready for data.
4) It also synchronizes the RESET (RST) signal which is used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

   F/ C = 1 ➔ Input Clock given **through EFI** pin.

   F/ C = 0 ➔ Input Clock given **through** Oscillator inputs **X1, X2** pins.

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

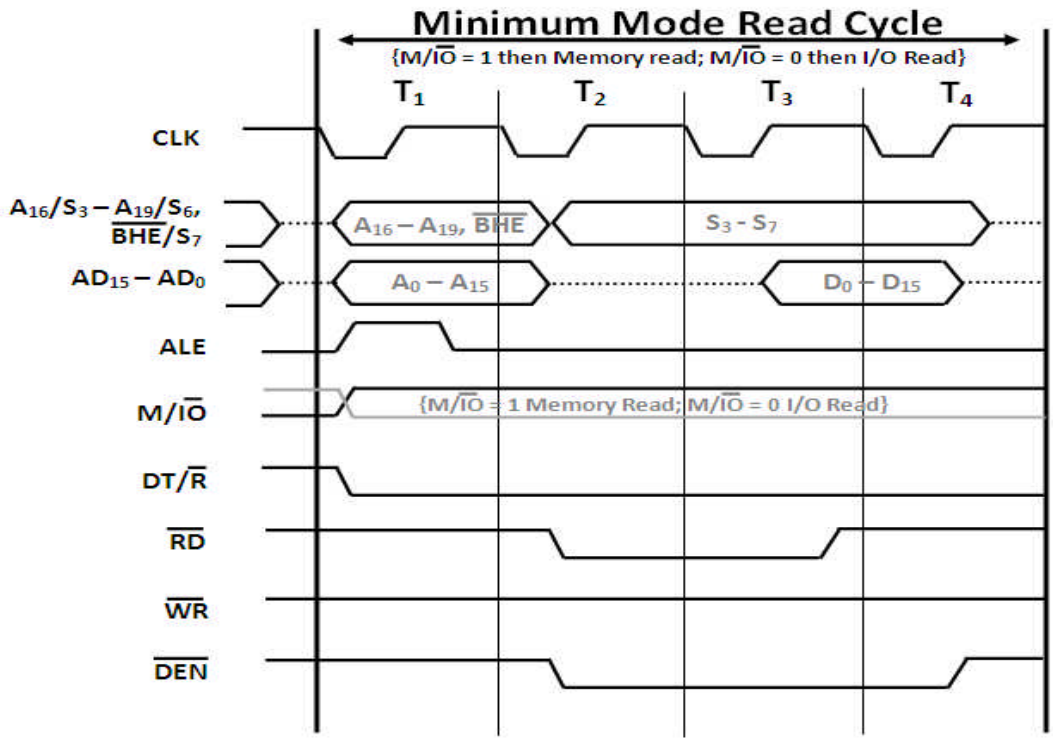
7) **M/ $\overline{IO}$, $\overline{RD}$, $\overline{WR}$** are **decoded by a 3:8 decoder like IC 74138.**

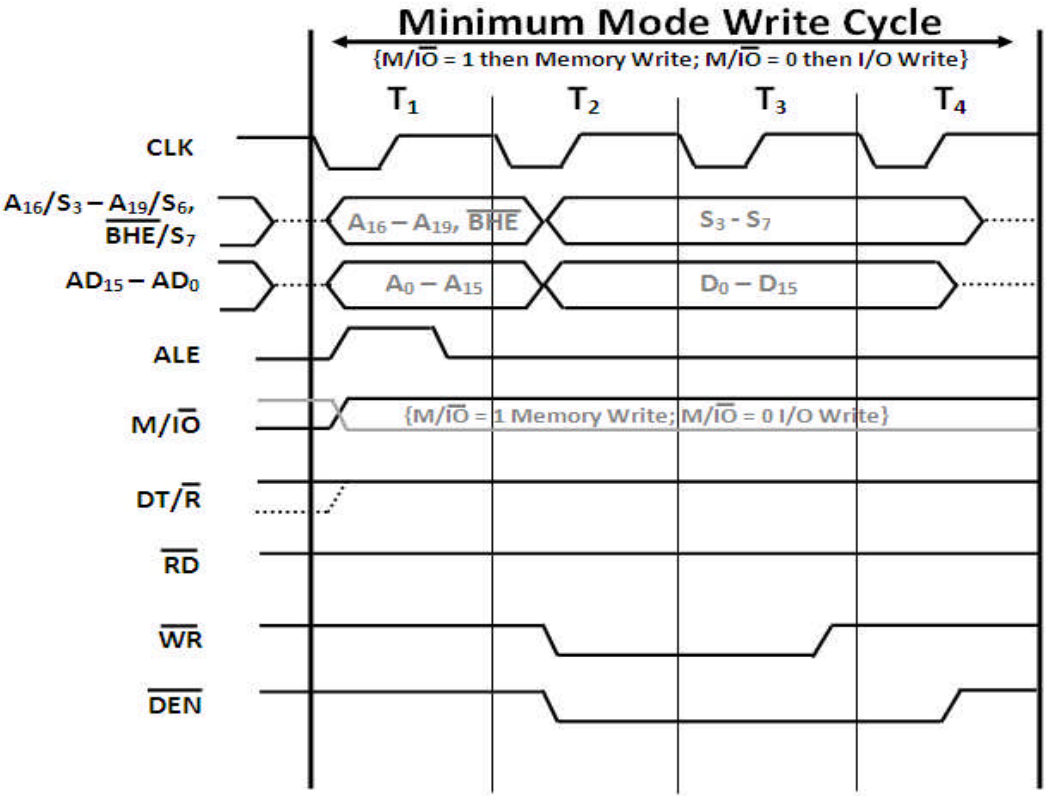
8) **Bus Request (DMA) is done using the HOLD and HLDA signals.**

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

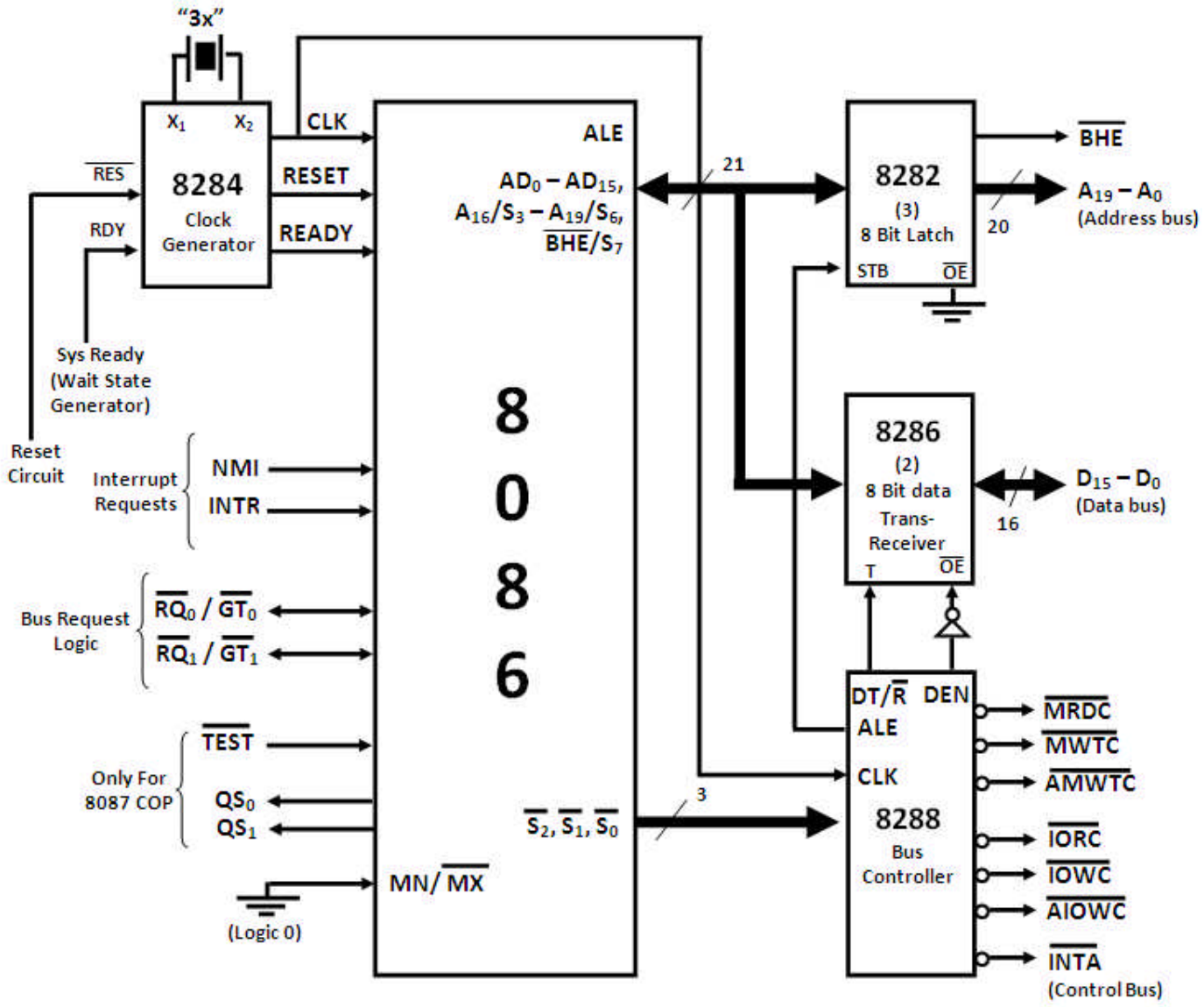
10) **The Circuit is simpler than Maximum Mode but does not support multiprocessing.**

Timing Diagrams:





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.

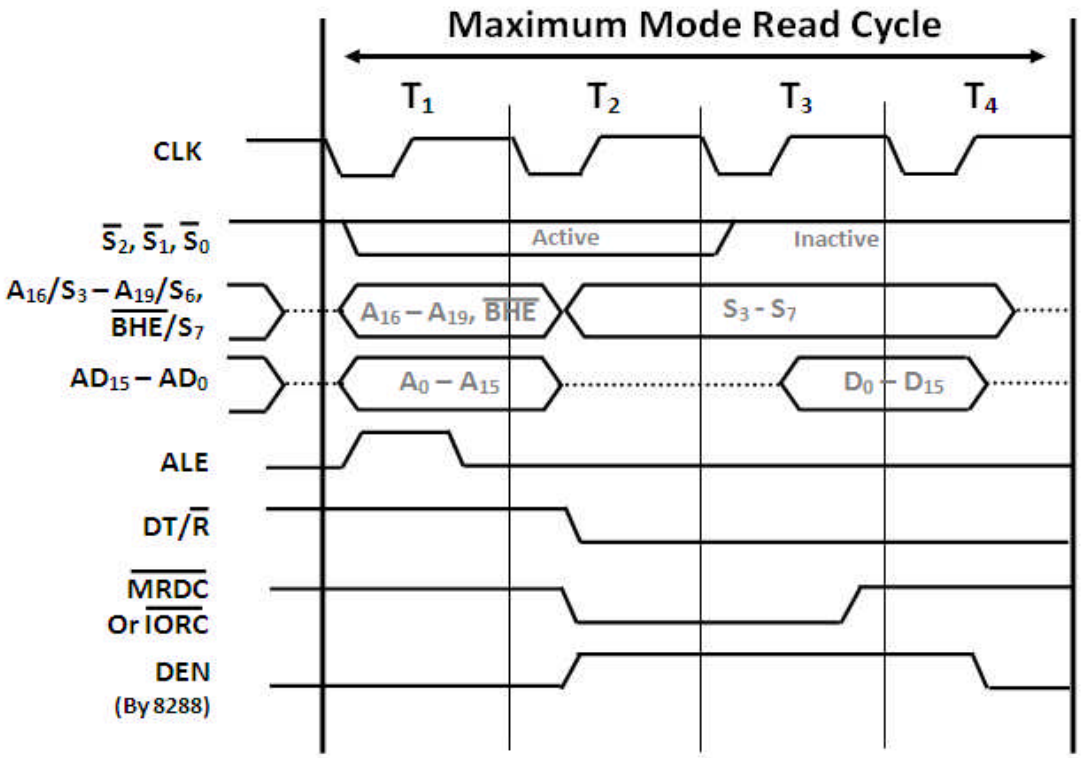
9) Bus request is done using $\overline{RQ}$ / $\overline{GT}$ lines interfaced with 8086.
RQ_0/GT_0 has higher priority than RQ_1/GT_1. ☺ For doubts contact Bharat Sir on 98204 08217

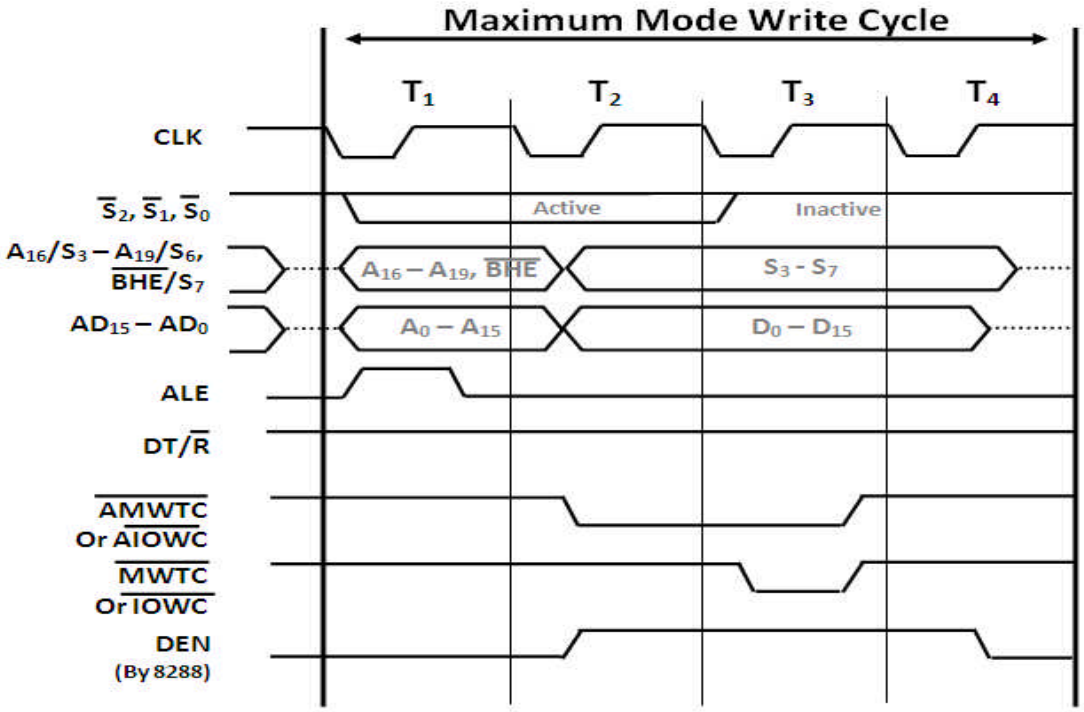
10) $\overline{INTA}$ is given by 8288 Bus Controller, in response to an int. on INTR line of 8086.

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".

TIMING DIAGRAMS





Differentiate between

	MIN MODE	MAX MODE
1	It is a **uniprocessor mode**. 8086 is the only processor in the circuit.	It is a **multiprocessor mode**. Along with 8086, there can be other processors like 8087 and 8089 in the circuit.
2	Here **MN/ $\overline{\text{MX}}$** is connected to **Vcc.**	Here **MN/ $\overline{\text{MX}}$** is connected to **Ground.**
3	**ALE** for the latch is **given by 8086** itself.	As there are multiple processors, **ALE** for the latch is **given by 8288 bus controller**.
4	**$\overline{\text{DEN}}$** and **DT/ $\overline{\text{R}}$** for the transreceivers **are given by 8086 itself.**	As there are multiple processors, **DEN** and **DT/ $\overline{\text{R}}$** for the transreceivers is **given by 8288 bus controller**.
5	Direct control signals like **M/ $\overline{\text{IO}}$, $\overline{\text{RD}}$ and $\overline{\text{WR}}$ are produced by 8086 itself.**	Instead of control signals, all processors produce status signals **$\overline{\text{S}_2}$, $\overline{\text{S}_1}$ and $\overline{\text{S}_0}$**
6	Control signals **M/ $\overline{\text{IO}}$, $\overline{\text{RD}}$ and $\overline{\text{WR}}$ are decocded by a 3:8 decoder IC 74138.**	Status signals **$\overline{\text{S}_2}$, $\overline{\text{S}_1}$ and $\overline{\text{S}_0}$ require special decoding** are decoded by **8288 bus controller**.
7	**$\overline{\text{INTA}}$** for interrupt acknowledgement **is produced by 8086.**	**$\overline{\text{INTA}}$** for interrupt acknowledgement **is produced by 8288 Bus Controller.**
8	**Bus request are grant** is handled using **HOLD and HLDA** signals.	**Bus request are grant** is handled using **$\overline{\text{RQ}}$ / $\overline{\text{GT}}$** signals.
9	Since **74138** does not independently generate any signals, it **does not need a CLK**.	Since **8288** independently generates control signals, it **needs a CLK from 8284** clock generator.
10	The circuit is **simpler but does not support multiprocessing**.	The circuit is **more complex but supports multiprocessing**.

Differentiate between

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

Differentiate between

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

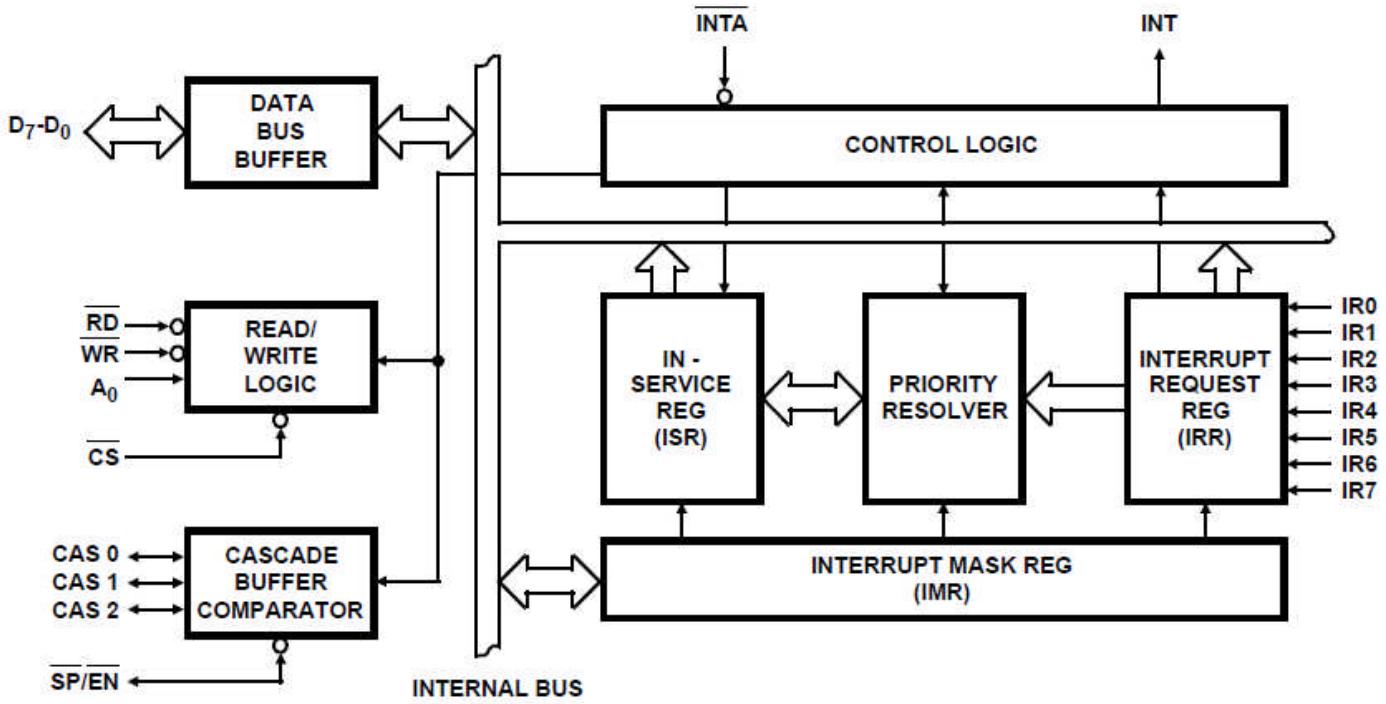
8259

PROGRAMMABLE INTERRUPT CONTROLLER

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**.

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.

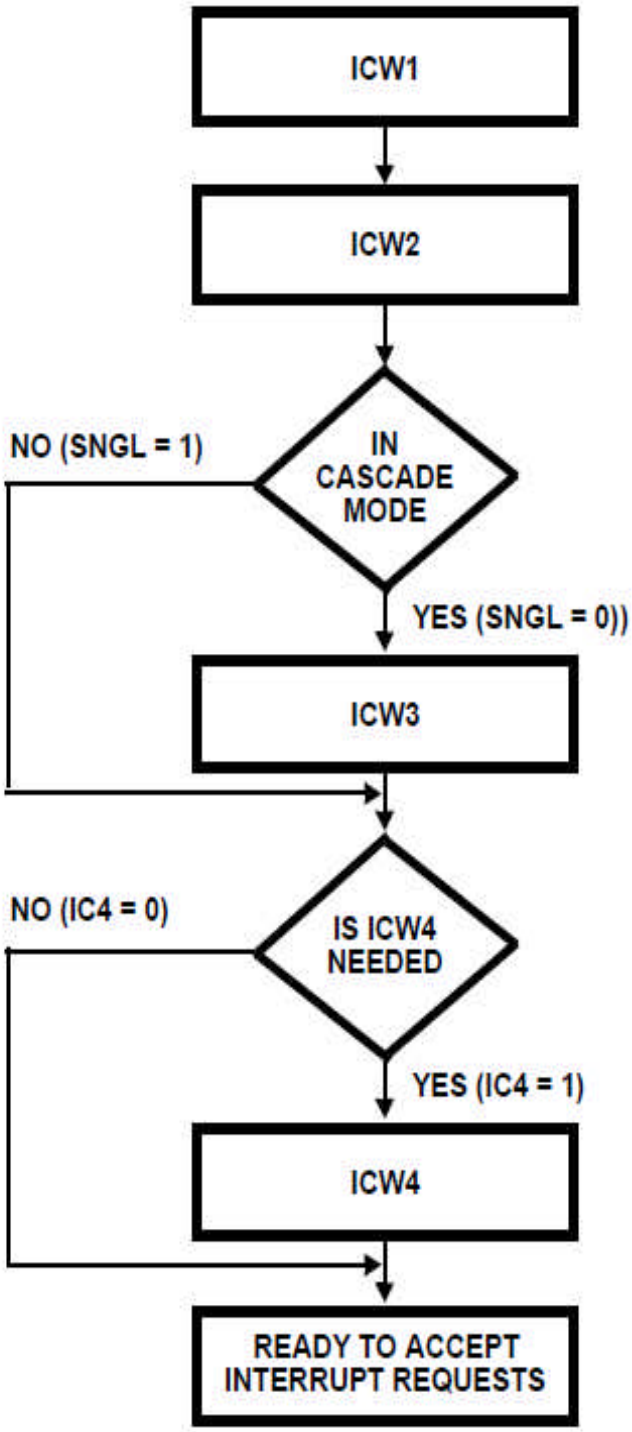
Specific EOI Command:

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

- IC4: 1 = ICW4 needed, 0 = No ICW4 needed
- SNGL: 1 = Single, 0 = Cascade Mode
- ADI: CALL address interval, 1 = Interval of 4, 0 = Interval of 8
- LTIM: 1 = Level triggered mode, 0 = Edge triggered mode
- A_7 - A_5: 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

- S_7 - 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

- µPM: 1 = 8086/8088 mode, 0 = MCS-80/85 mode
- AEOI: 1 = Auto EOI, 0 = Normal EOI
- BUF / M/S:

BUF	M/S	
0	X	- Non buffered mode
1	0	- Buffered mode slave
1	1	- Buffered mode master

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

† 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	

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 $\overline{\text{INTA}}$ pulse**.

4) On receiving the first $\overline{\text{INTA}}$ 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 $\overline{\text{INTA}}$ pulse** to 8259.

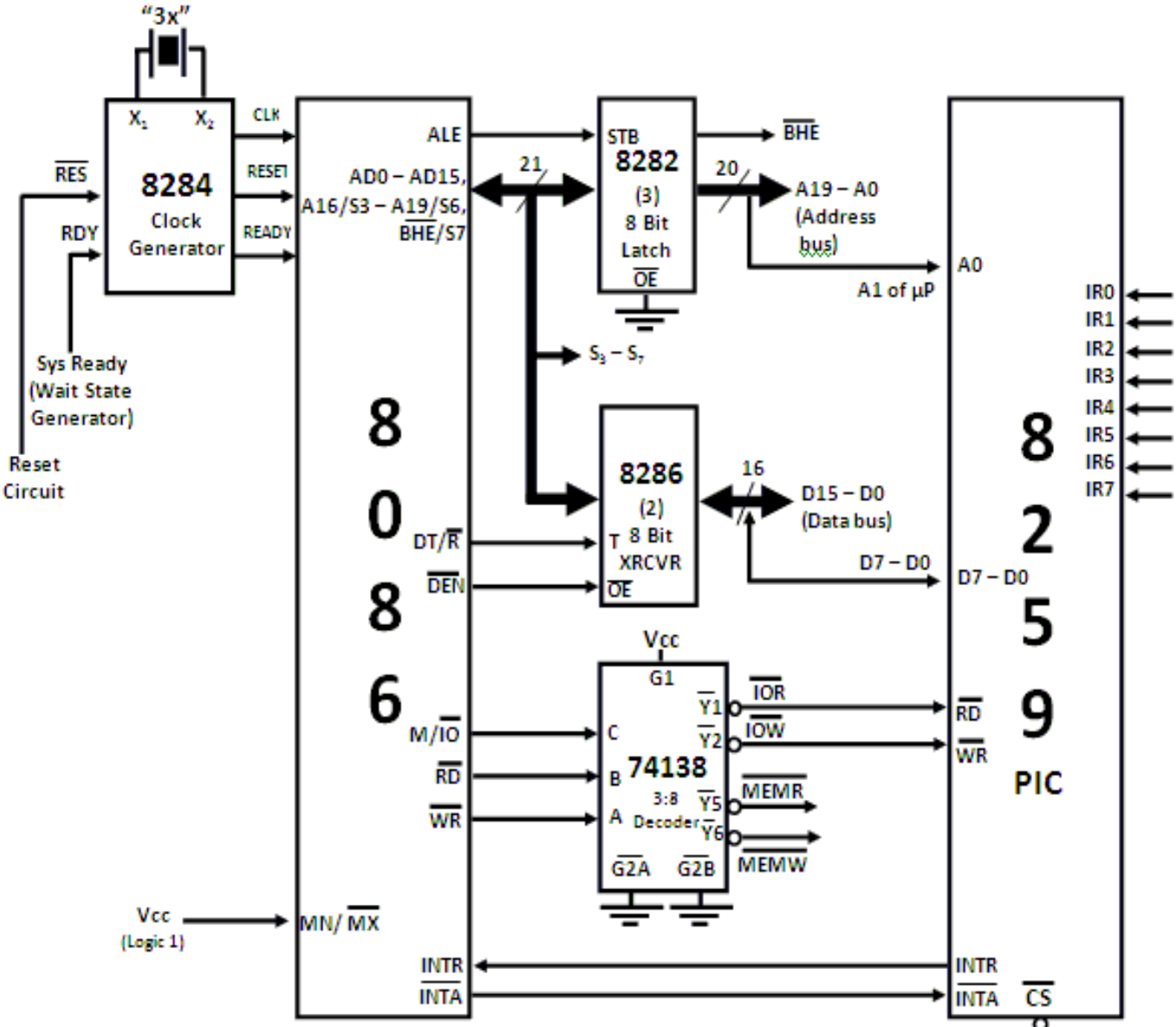
6) In response to the 2nd $\overline{\text{INTA}}$ pulse, **8259 sends the one byte Vector Number N to µP**.

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 ...*

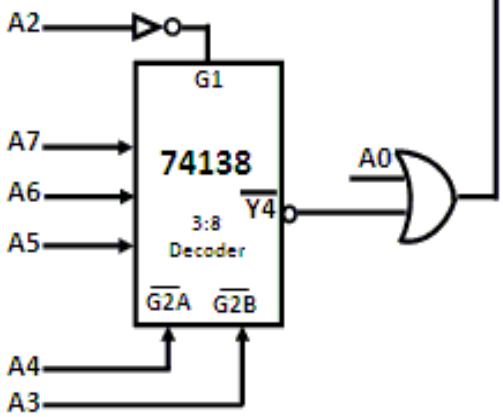
10) **The ISR thus begins**.



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**.

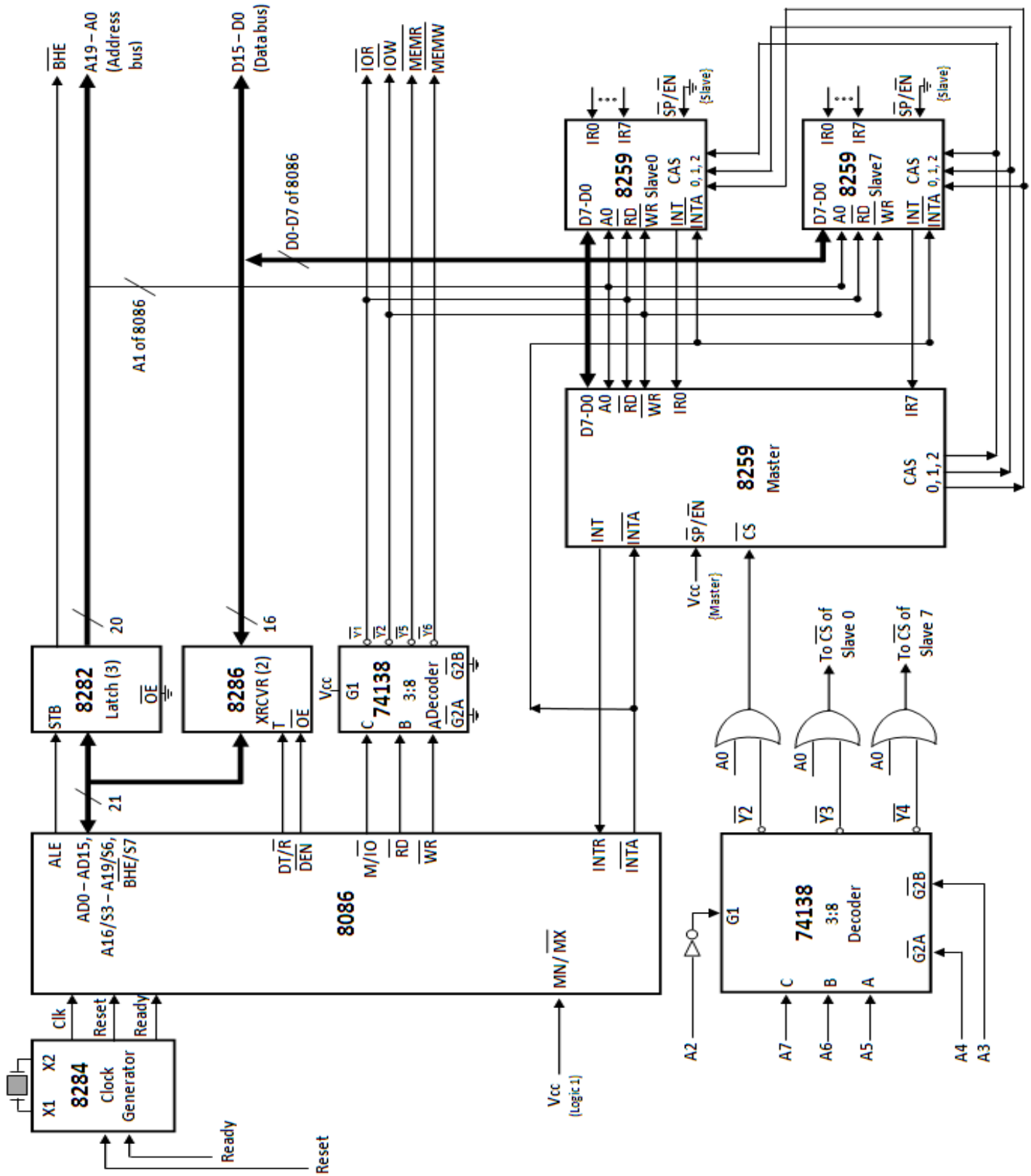
   iii. The **Slave identifies** its number on the **CAS lines** and **sets** the **corresponding bit in its InSR**.

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
8259 Master
8259
Slave0
Slave7
{Slave}
{Master}
8282 Latch (3)
8286 XRCVR (2)
74138 3:8 Decoder
8086
8284 Clock Generator
To CS of Slave 0
To CS of Slave 7
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
Vcc
(Logic 1)
A2
A7
A6
A5
A4
A3

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**

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
```

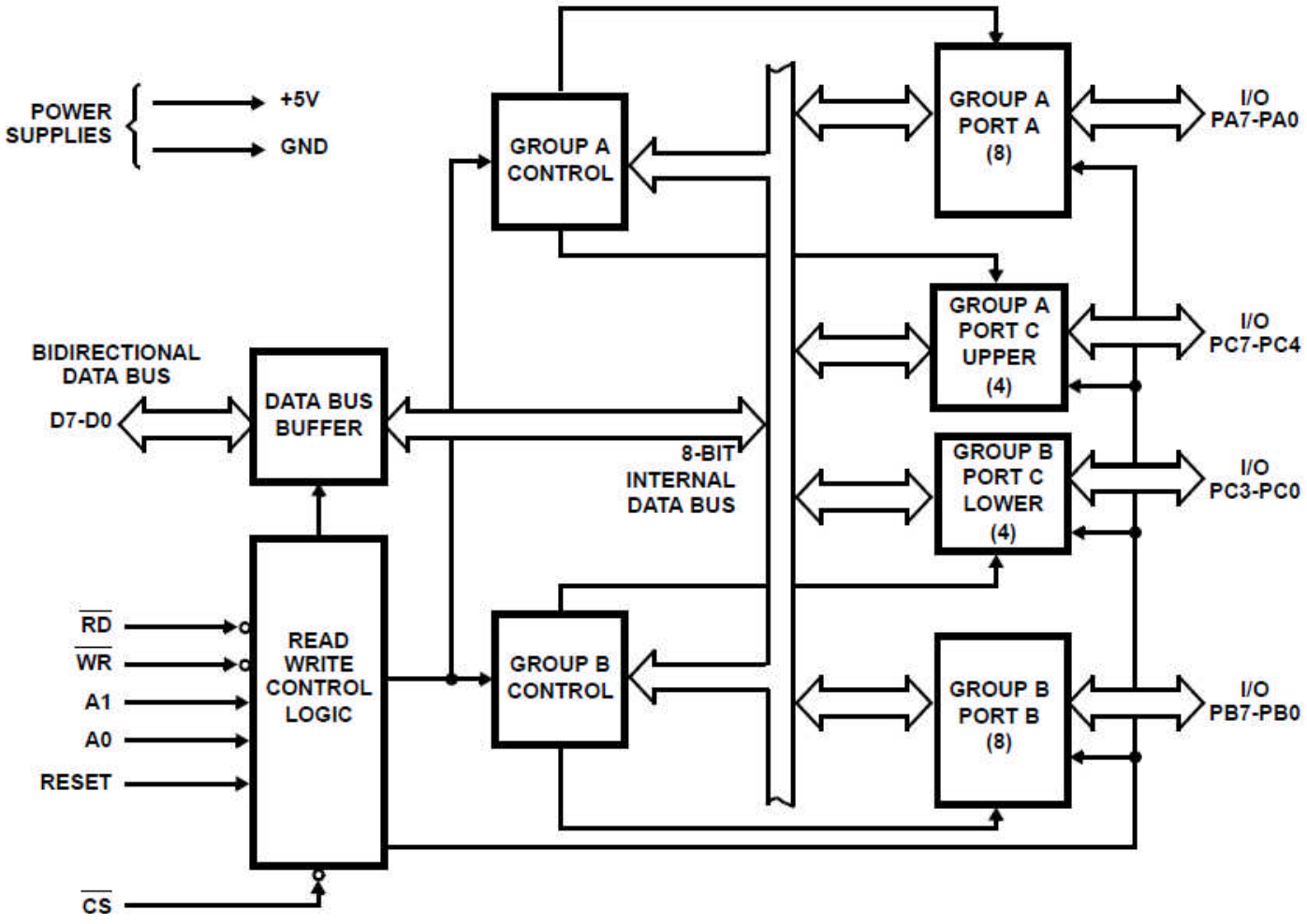
8255

PROGRAMMABLE PERIPHERAL INTERFACE

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

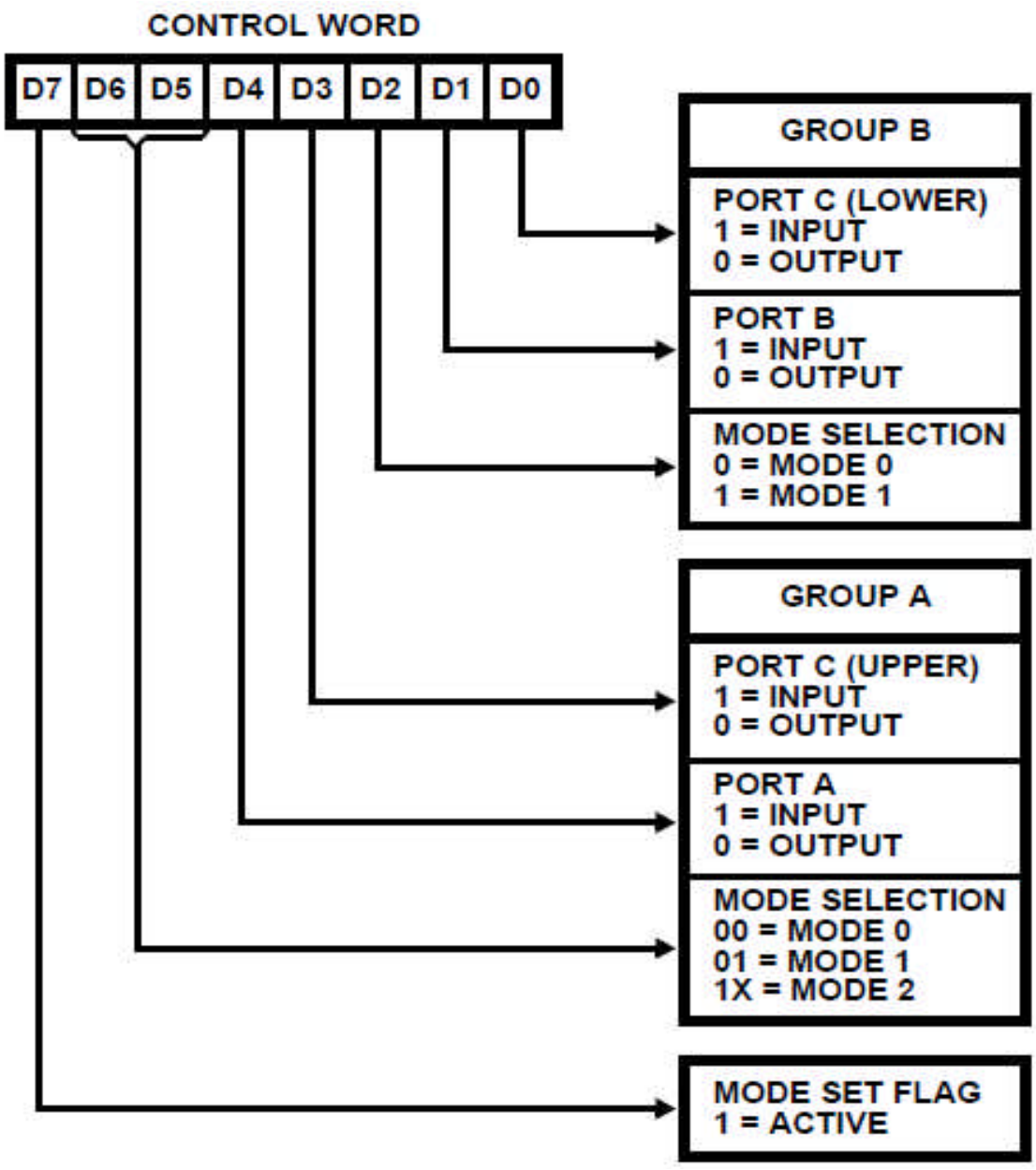
These are 8-bit Bi-directional Ports.
They can be programmed to wok in the various modes as follows:

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.

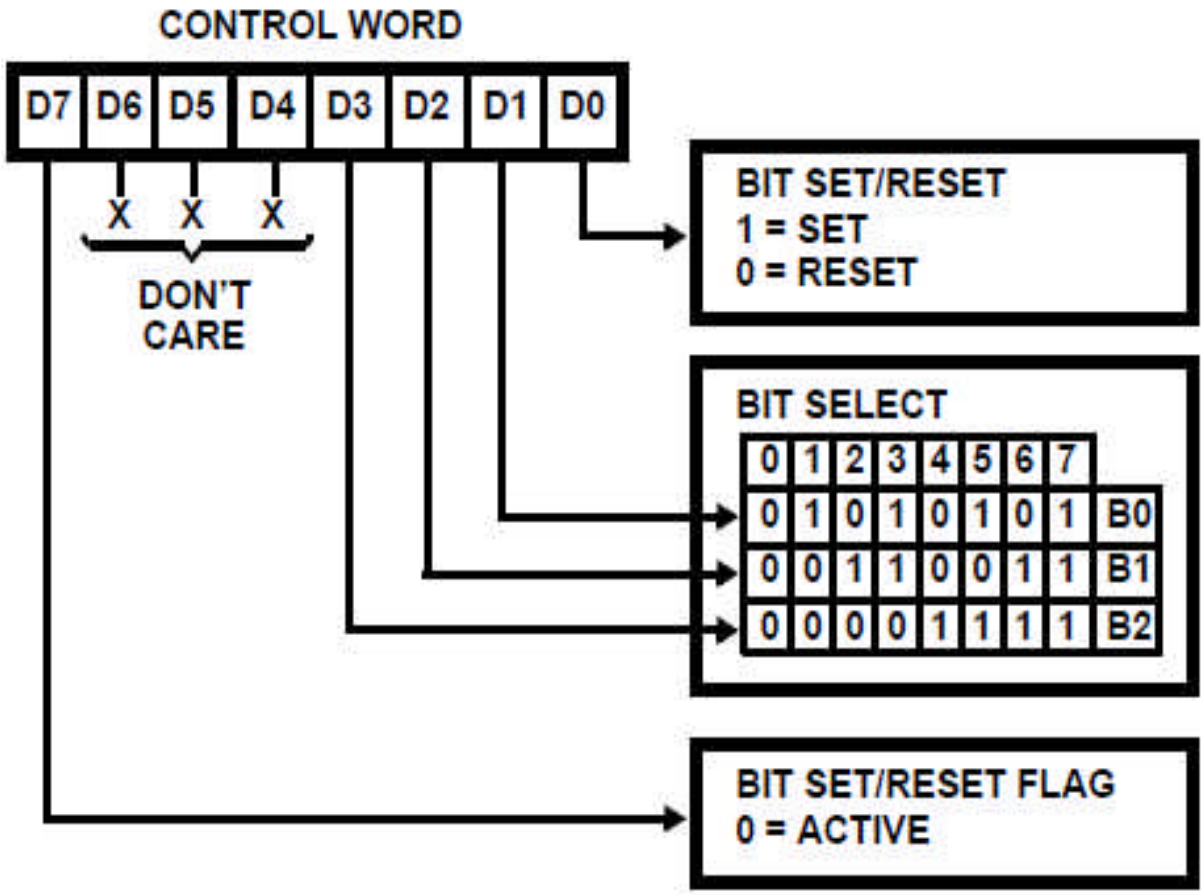
1)Control Word of 8255 - I/O Mode (I/O Command)

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)

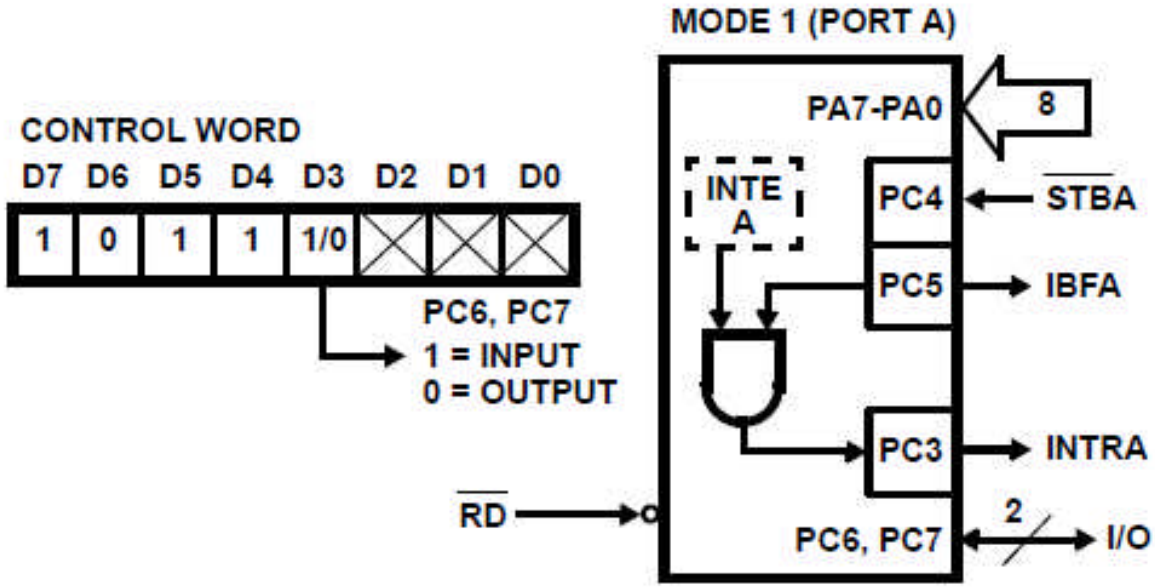
Port A and Port B used as 2 Simple 8-bit I/O Ports.
Port C is used as 2 simple 4-bit I/O Ports.
Each port can be programmed as input or output individually.
Ports do not have handshake or interrupting capability.
Hence, **slower** devices cannot be interfaced.

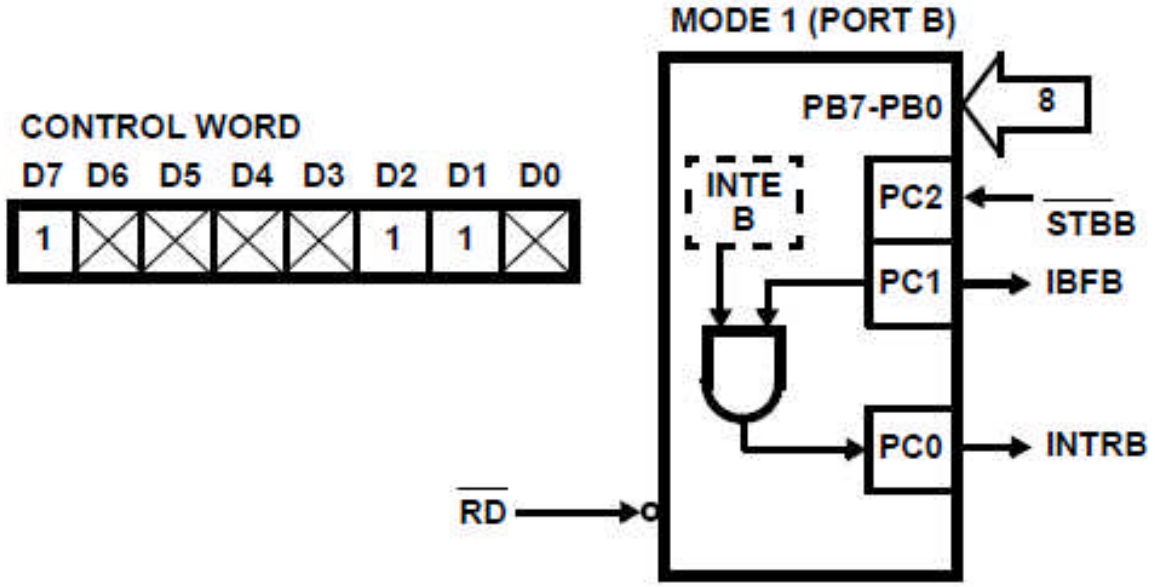
❖ Mode 1 (Handshake I/O)

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.

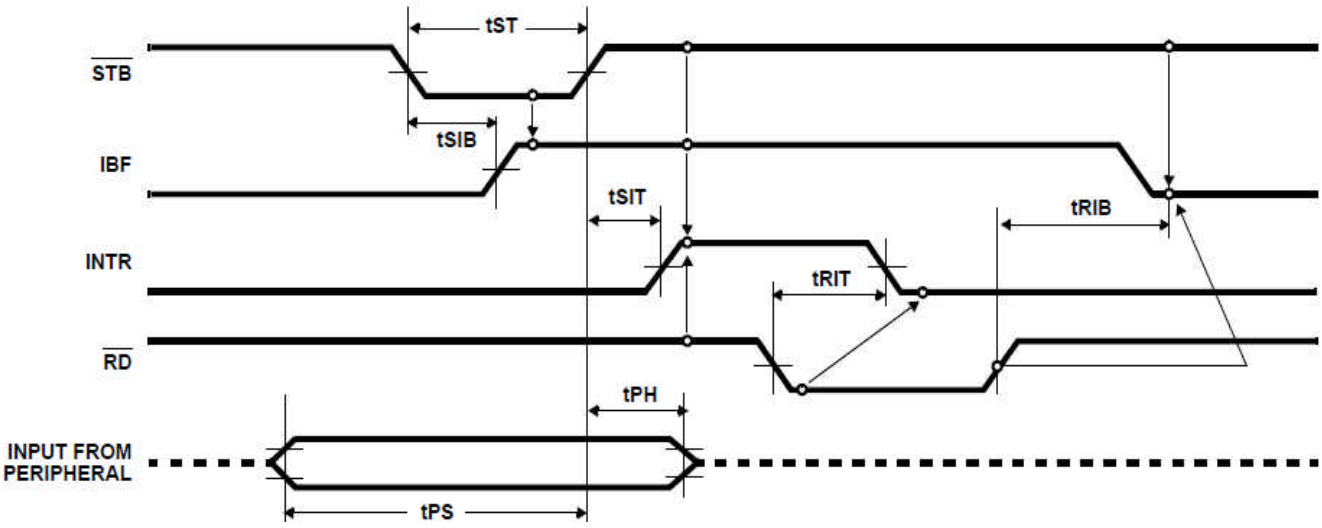
The handshake signals are different for input and output modes.
#Please refer Bharat Sir's Lecture Notes for this ...

♦ Mode 1 (Input Handshaking)





Timing Diagram for Mode 1 Input Transfer



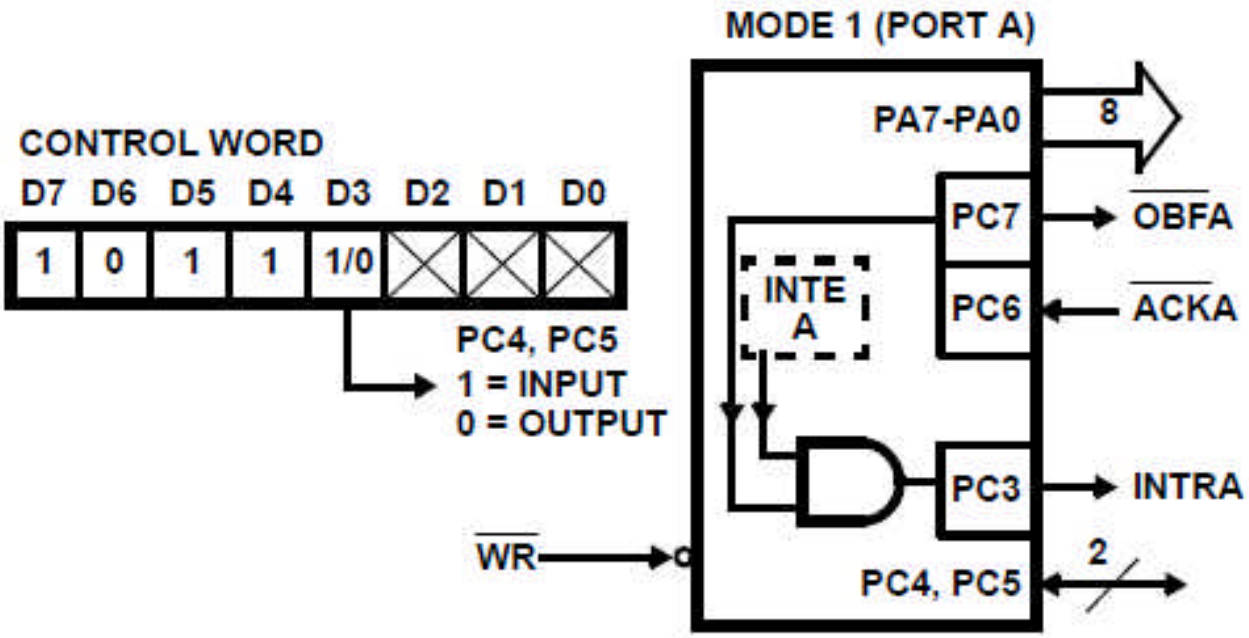
Working:

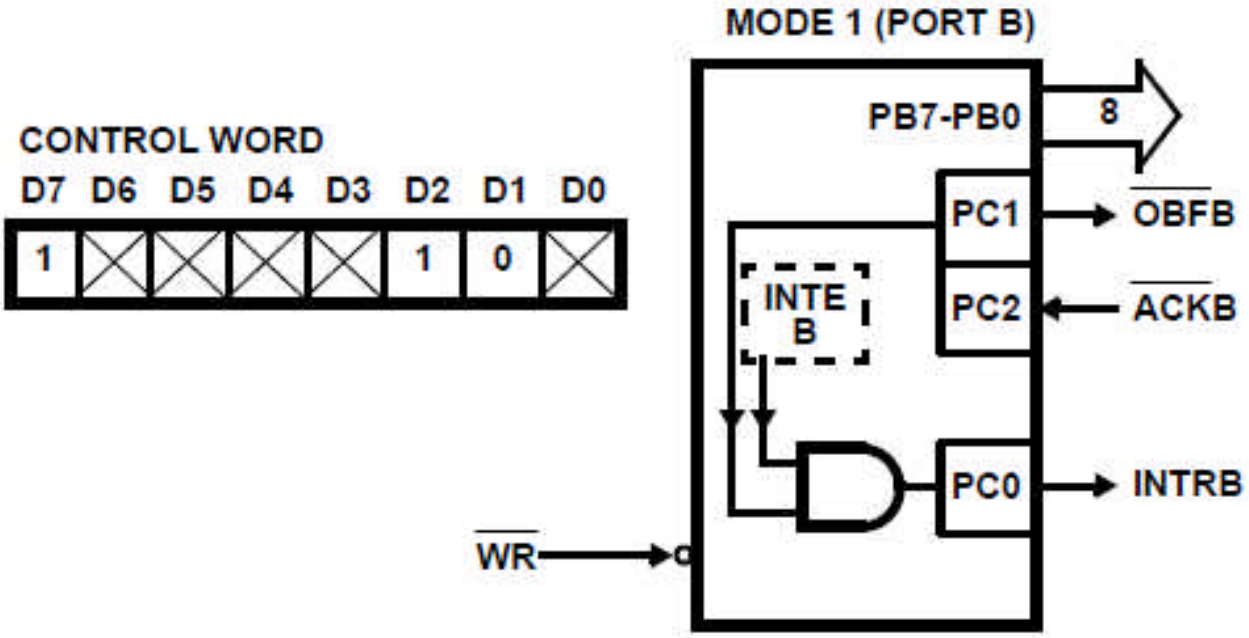
Each port uses **3 lines** of **Port C** for the following signals:
STB (Strobe), **IBF** (Input Buffer Full) ➔ Handhsake signals
INTR (interrupt) ➔ Interrupt signal
Additionally the $\overline{RD}$ signal of 8255 is also used.

Handshaking takes place in the following manner:

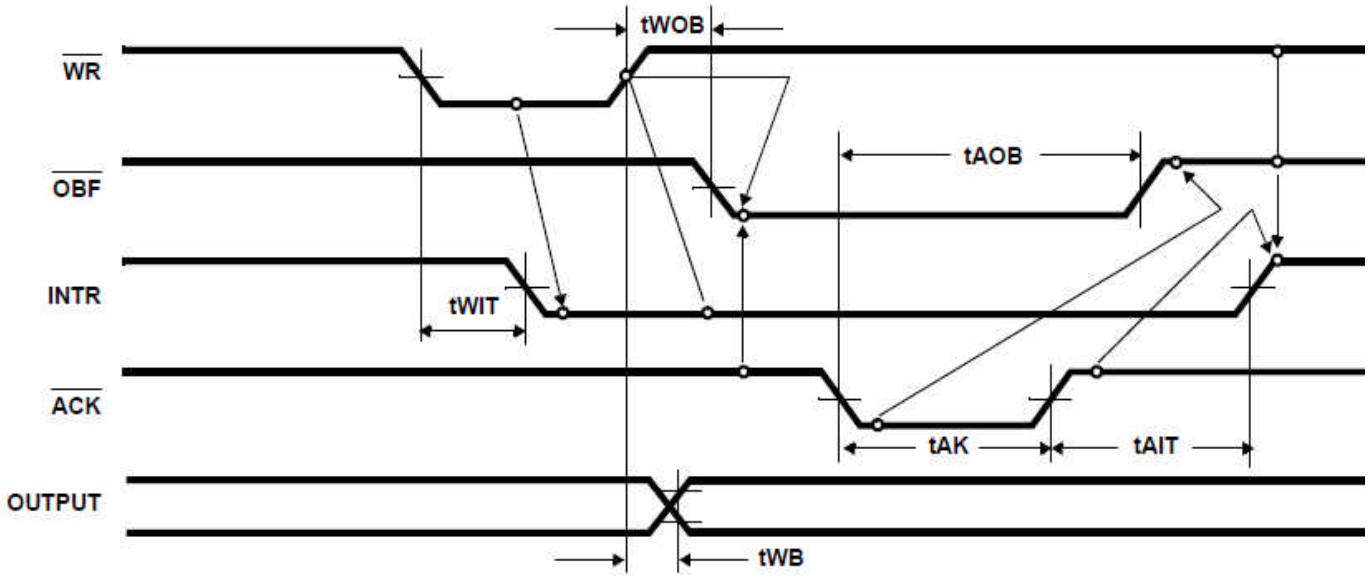
1) The **peripheral** device **places data** on the Port **bus** and informs the Port by **making STB low**.
2) The **input Port accepts** the **data** and informs the peripheral to wait by making **IBF high**. This **prevents** the peripheral from **sending more data** to the 8255 and **hence data loss** 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



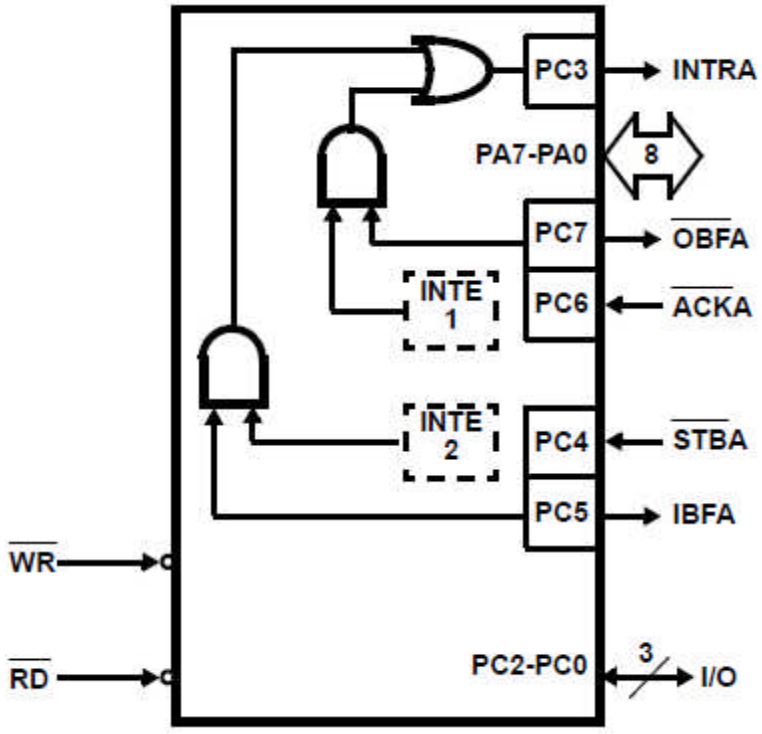
Working

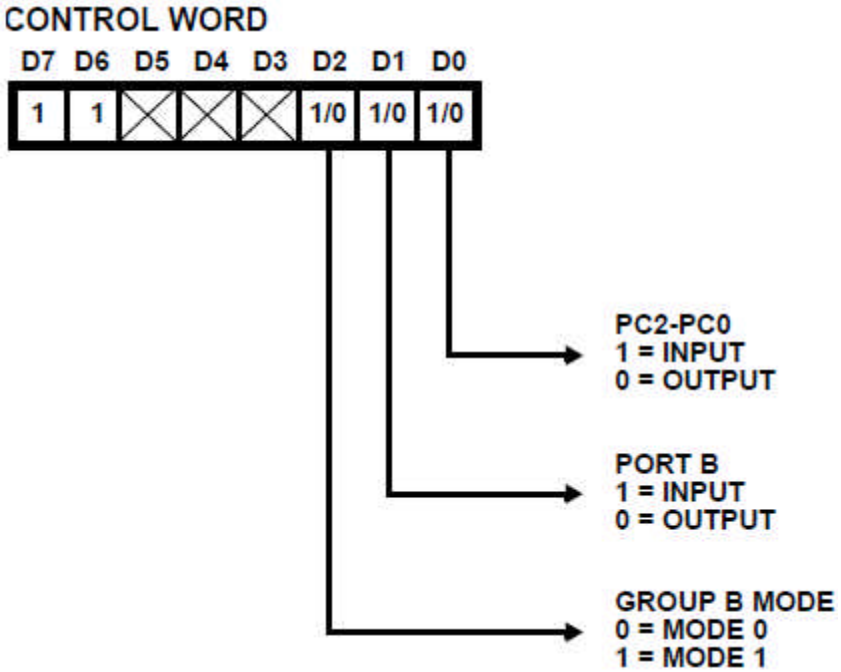
Each port uses **3 lines** of **Port C** for the following signals:
$\overline{\textbf{OBF}}$ (Output Buffer Full), $\overline{\textbf{ACK}}$ (Acknowledgement) ➔ Handhsake signals

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

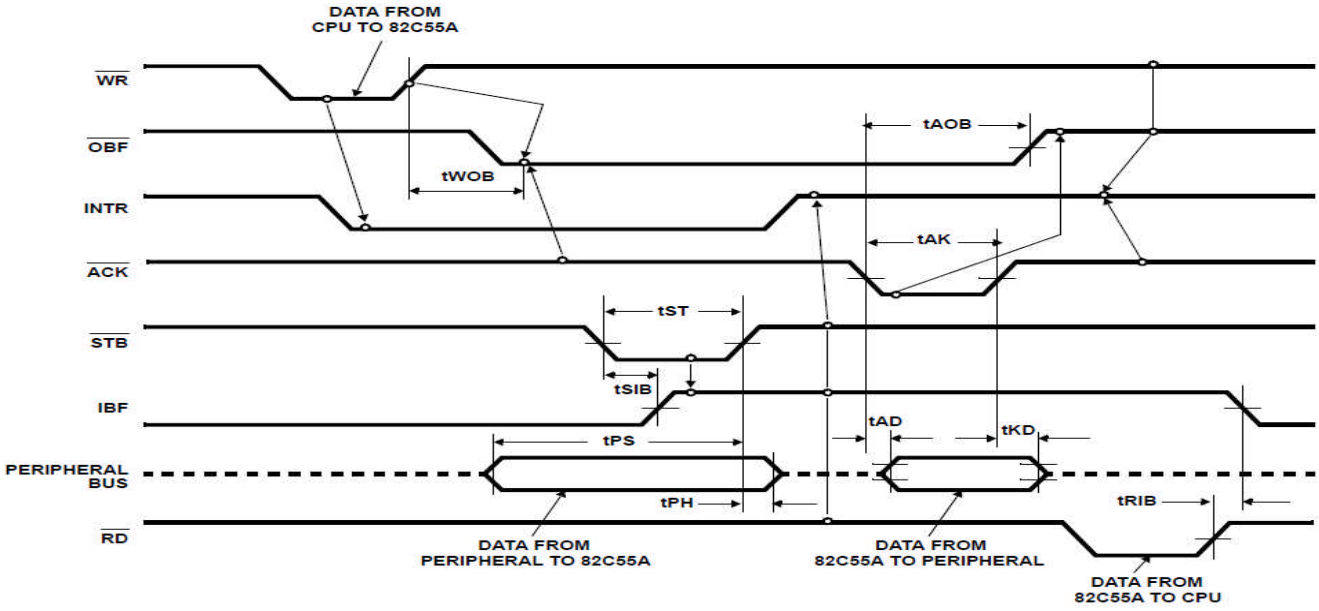
1) When the output port is empty (indicated by a high on the INTR line), the **µP writes data** on the output port by giving the $\overline{\textbf{WR}}$ signal.
2) As soon as the $\overline{\textbf{WR}}$ operation is complete, the **8255 makes the INTR low**, indicating that the µP should **wait**. 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**.

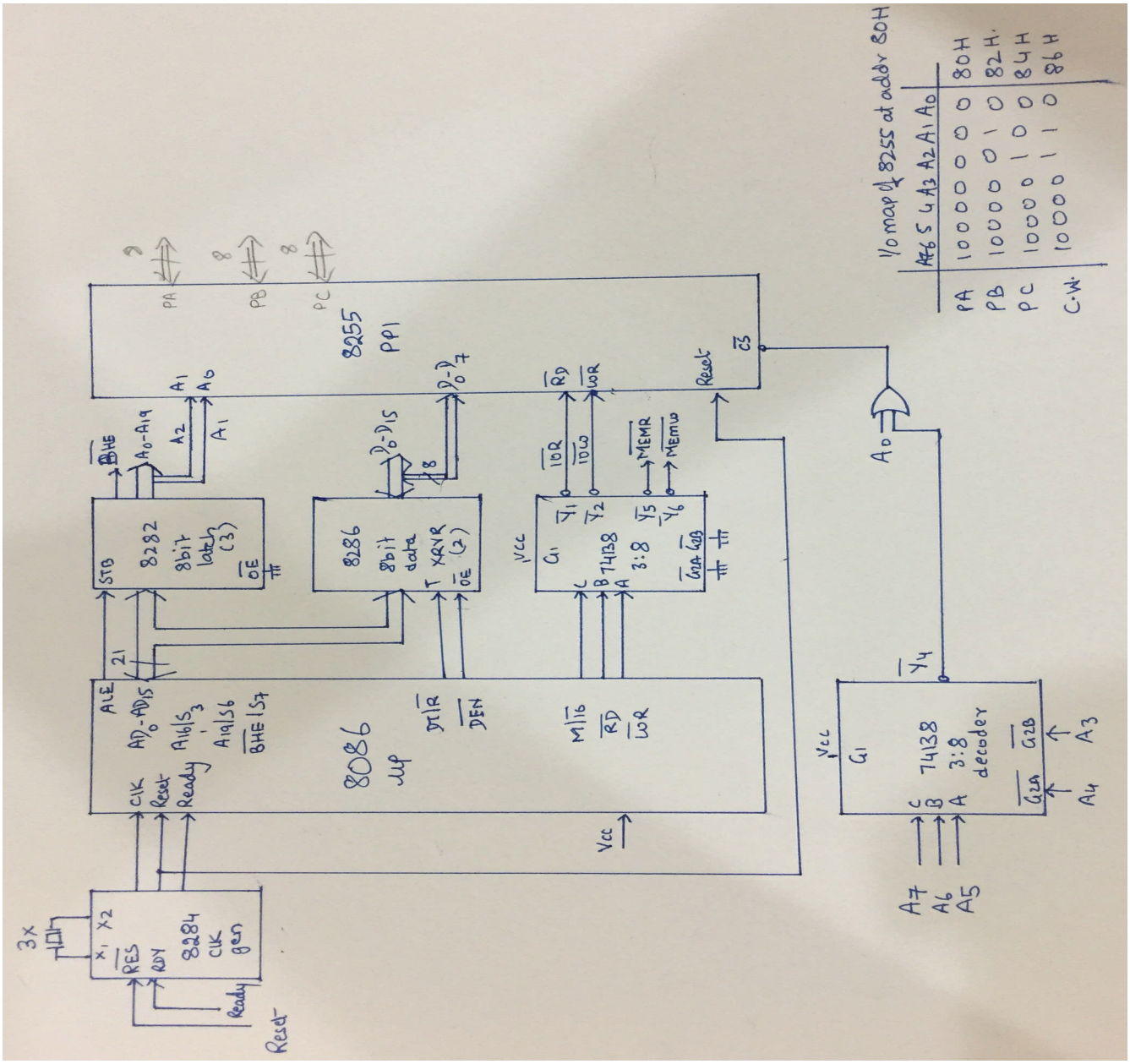
Port A can be used for data transfer between two computers as shown.
The high-speed computer is known as the master and the dedicated computer is known as the slave.
Handshaking process is similar to Mode 1.

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
A6 5 4 A3 A2 A1 A0
PA 1 0 0 0 0 0 0 80H
PB 1 0 0 0 0 0 1 82H
PC 1 0 0 0 0 1 0 84H
C.W. 1 0 0 0 0 1 1 86H
8255 PPI
PA
PB
PC
8086 μp
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

8255 PROGRAMMING

Q 1) **WAP to initialize 8255 as follows:**
Port A --- Mode1 ---- i/p
Port B --- Mode1 ---- o/p
Port C --- Handshaking
Assume 8255 is at 80H.

Soln:

```
Code  SEGMENT
      ASSUME CS: Code
      MOV   AL, B4H
      OUT   86H, AL          // Control Word = 1011 0100  = B4H
      INT 03H
Code  ENDS
      END
```

Q 2) **Generate a square wave on a display device connected to PortC_3 by BSR command.**

Soln: For doubts contact Bharat Sir on 98204 08217

```
Code  SEGMENT
      ASSUME CS: Code
Back: MOV   AL, 06H
      OUT   86H, AL          // BSR Command (Send 0) = 0000 0110
      CALL  Delay
      MOV   AL, 07H
      OUT   86H, AL          // BSR Command (Send 1) = 0000 0111
      CALL  Delay
      JMP   Back
      INT 03H
Code  ENDS
      END
```

Q 3) **WAP to generate "positive spikes" on a display device connected to PortC_3 using BSR command.**

Soln: For doubts contact Bharat Sir on 98204 08217

```
Code  SEGMENT
      ASSUME CS: Code
Back: MOV   AL, 06H
      OUT   86H, AL          // BSR Command (Send 0) = 0000 0110
      CALL  Delay
      MOV   AL, 07H
      OUT   86H, AL          // BSR Command (Send 1) = 0000 0111
      JMP   Back
      INT 03H
Code  ENDS
      END
```

Q 4) **WAP to generate a rectangular wave with a 25% duty cycle on a display device connected to PortC_3 using BSR command.**

Soln: For doubts contact Bharat Sir on 98204 08217

```
Code  SEGMENT
      ASSUME CS: Code
Back: MOV   AL, 06H
      OUT   86H, AL          // BSR Command (Send 0) = 0000 0110
      CALL  Delay
      CALL  Delay
      CALL  Delay
      MOV   AL, 07H
      OUT   86H, AL          // BSR Command (Send 1) = 0000 0111
      CALL  Delay
      JMP   Back
      INT 03H
Code  ENDS
      END
```

Q 5) **Generate a Staircase waveform on a display device connected to Port A.**

Soln: For doubts contact Bharat Sir on 98204 08217

```
Code  SEGMENT
      ASSUME CS: Code
      MOV   AL, 80H
      OUT   86H, AL          // Initialize Port A as Output port
Back: MOV   AL, 00H
      OUT   80H, AL
      CALL  Delay
      INC   AL
      JNZ   Back
      INT 03H
Code  ENDS
      END
```

Q 6) **Generate a Ramp waveform on a display device connected to Port A.**

Soln: For doubts contact Bharat Sir on 98204 08217

```
Code  SEGMENT
      ASSUME CS: Code
      MOV   AL, 80H
      OUT   86H, AL          // Initialize Port A as Output port
Back: MOV   AL, 00H
      OUT   80H, AL
      INC   AL
      JNZ   Back
      INT 03H
Code  ENDS
      END
```

Note: ***Additionally the circuit will require a DAC to produce a smooth waveform like a ramp. Refer later chapters.***

Q 7) Generate a Saw-tooth waveform on a display device connected to Port A.

Soln: For doubts contact Bharat Sir on 98204 08217

```
Code  SEGMENT
      ASSUME CS: Code
      MOV   AL, 80H
      OUT   86H, AL          // Initialize Port A as Output port
Back: MOV   AL, 00H
      OUT   80H, AL
      INC   AL
      JMP   Back
      INT 03H
Code  ENDS
      END
```

Note: Additionally the circuit will require a DAC to produce a smooth waveform like a saw-tooth.

Q 8) Generate a triangular waveform on a display device connected to Port A.

Soln: For doubts contact Bharat Sir on 98204 08217

```
Code  SEGMENT
      ASSUME CS: Code
      MOV   AL, 80H
      OUT   86H, AL          // Initialize Port A as Output port
Back: MOV   AL, 00H
      OUT   80H, AL
      INC   AL
      JNZ   Back             // 00H …… FFH
      MOV   AL, 0FEH
Back2:OUT   80H, AL
      DEC   AL
      JNZ   Back2            // FEH …… 01H
      JMP   Back
      INT 03H
Code  ENDS
      END
```

Note: Additionally the circuit will require a DAC to produce a smooth waveform like a triangular wave.

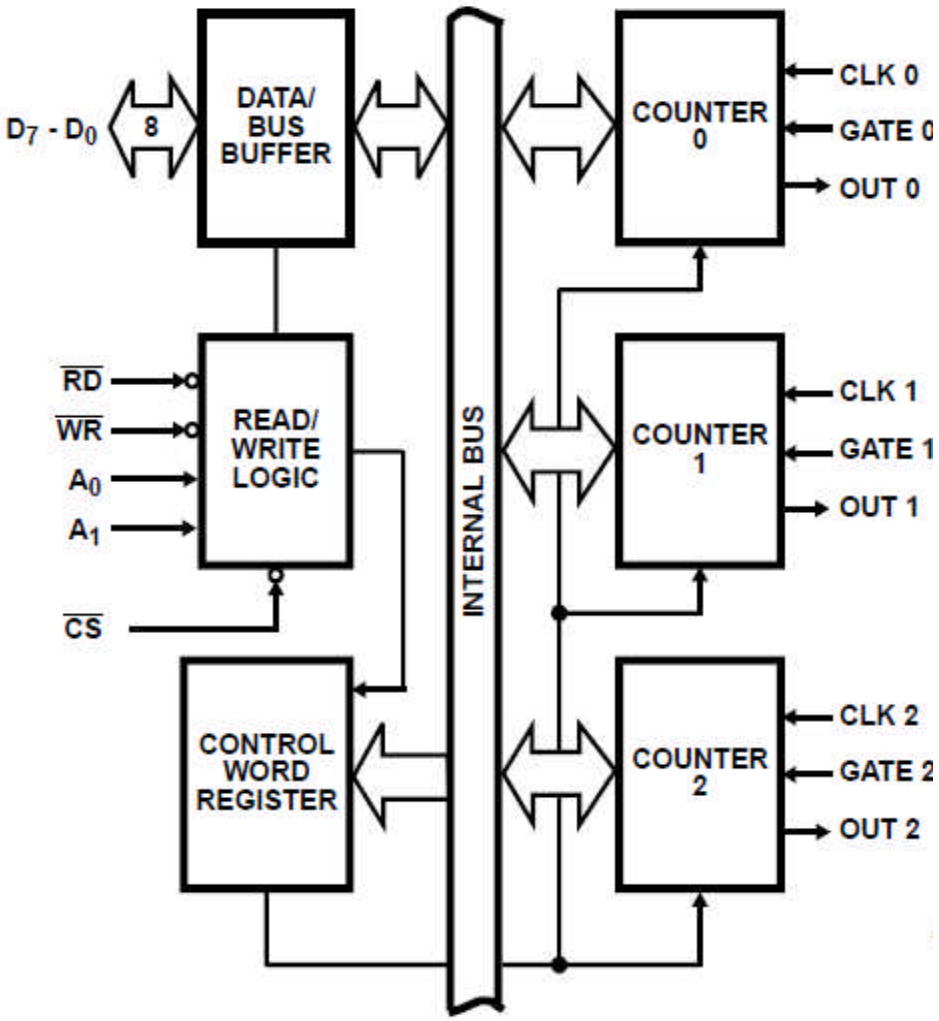
8254

PROGRAMMABLE INTERVAL TIMER

Salient Features

1) IC **8254** is **used** as a timer device to produce **Hardware delays**.
2) It can **also** be **used** to generate a **real-time clock**, or as a **square wave** generator **etc**.
3) 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.
4) The counting is done using **3 independent 16-bit down counters**.
5) These counters can take the count in **BCD or** in **Binary**.
6) 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 $D_7 - D_0$ form the µP.

2) Read Write Logic

It accepts the $\overline{RD}$ & $\overline{WR}$ signals, which are used to **control** the flow of **data** through data bus. The $\overline{CS}$ signal is used to **select** the **8254** chip.

It also accepts the $A_1 - A_0$ 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 --- Mono-stable Multi-vibrator
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

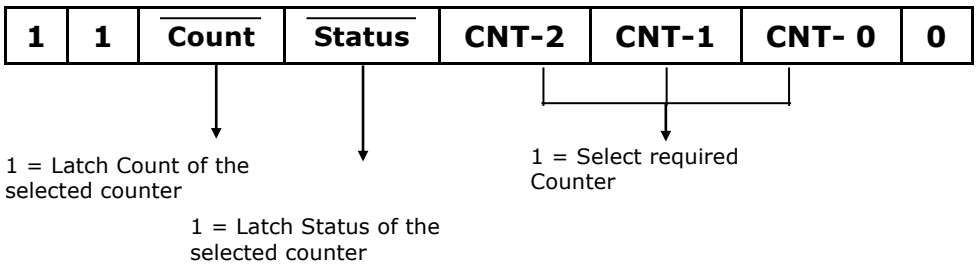
The Control Word Register is an **8-bit** register that holds the Control Word as shown above.
It is selected when $A_1 - A_0$ contain 11.
It has a different format when a Read Back command is given for 8254, as shown below

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**.

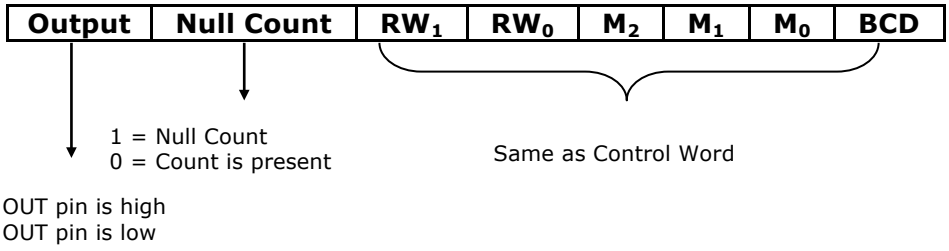
B) **Read on Fly**
In this method, the µP reads the **count** value **while** the **counting** is still in progress.
Thus, it is called as **Read On Fly**.
The appropriate value is written in the control word, and IO Read operation is performed.
The current value of the **Count** is **"latched" internally** 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**.

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 --- Mono-stable Multi-vibrator

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**. ☺ 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.

♦ 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.**

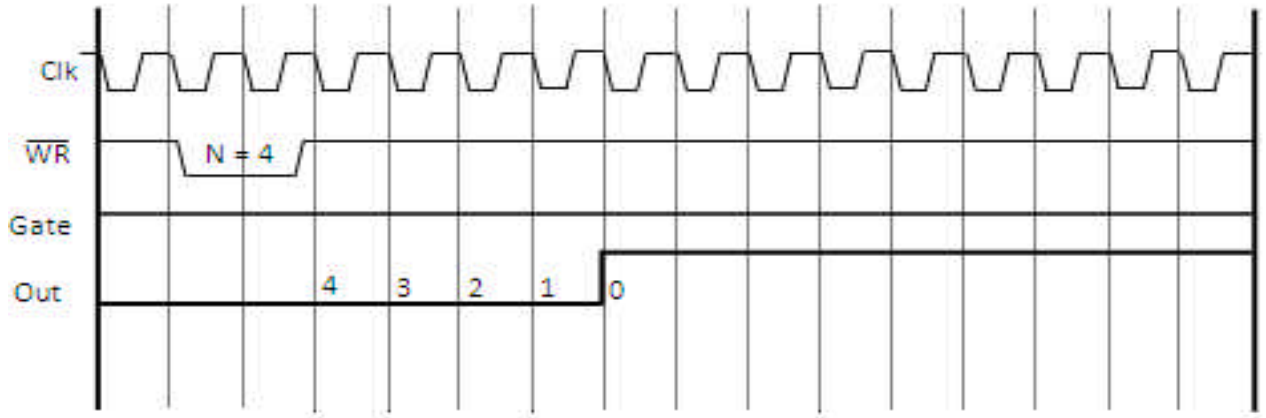
♦ Mode 4 --- Software Triggered Strobe

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

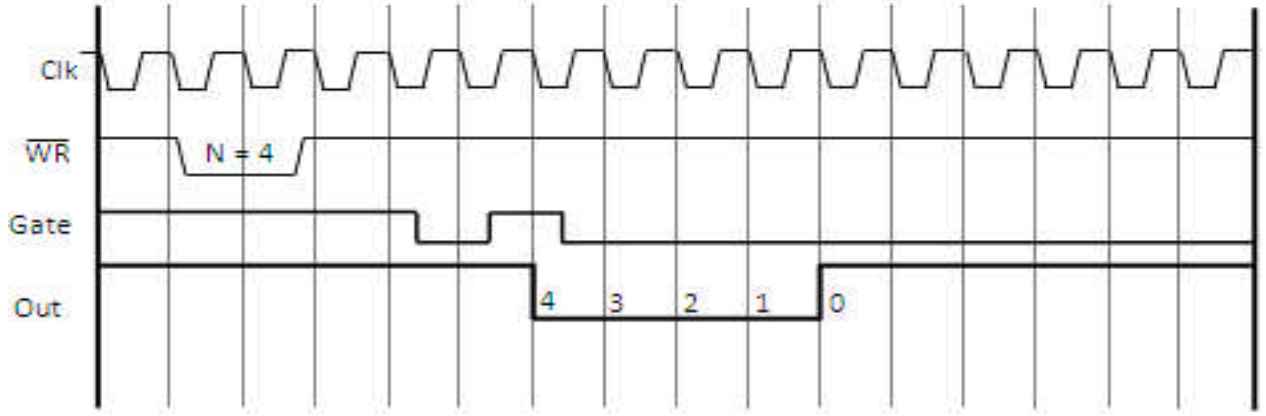
♦ Mode 5 --- Hardware Triggered Strobe

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**.
6) The **OUT** pin goes **low** for **one** clock **cycle, just after TC**.
7) **After that OUT** pin goes **high** and remains high.
8) **Thus GATE is used as a Trigger** i.e. it has to be triggered to start counting.
   Effect of Gate:
   Low ➔ No Effect; **High (Trigger) ➔** Starts Counting, can also re-rtigger it.

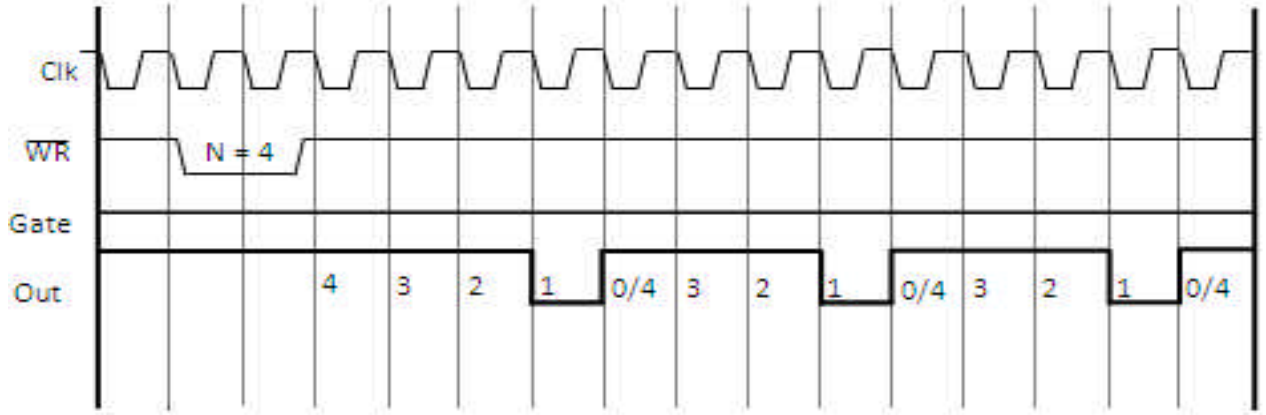
Mode 0: Interrupt On Terminal Count

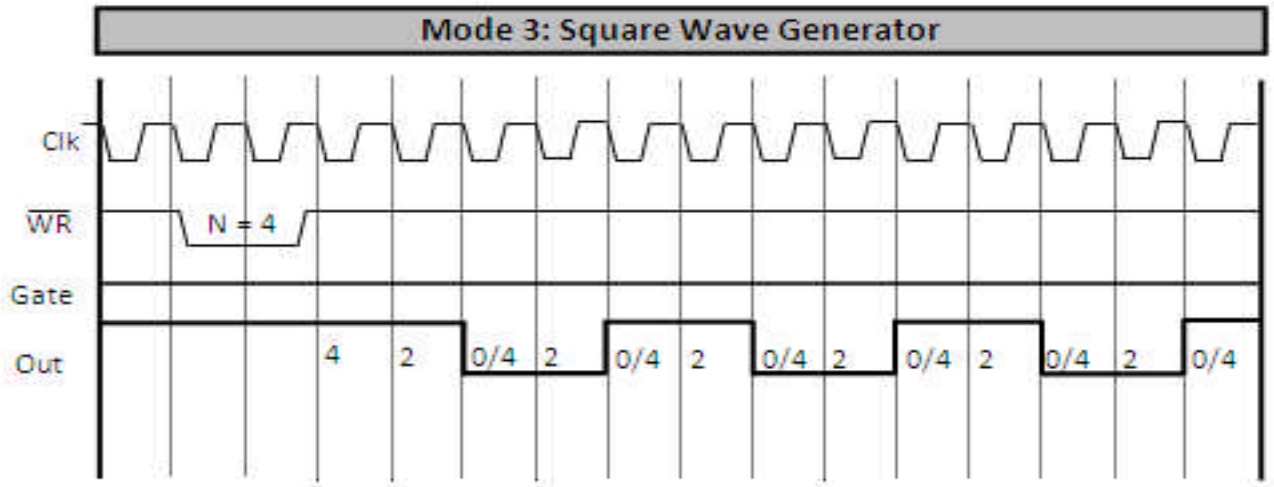


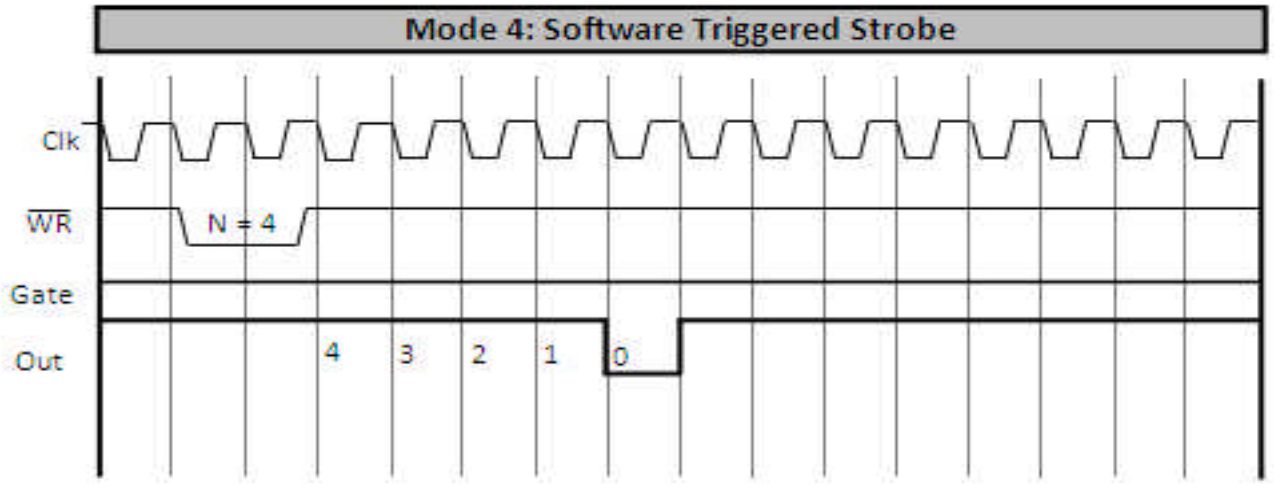
Mode 1: Monostable Multivibrator

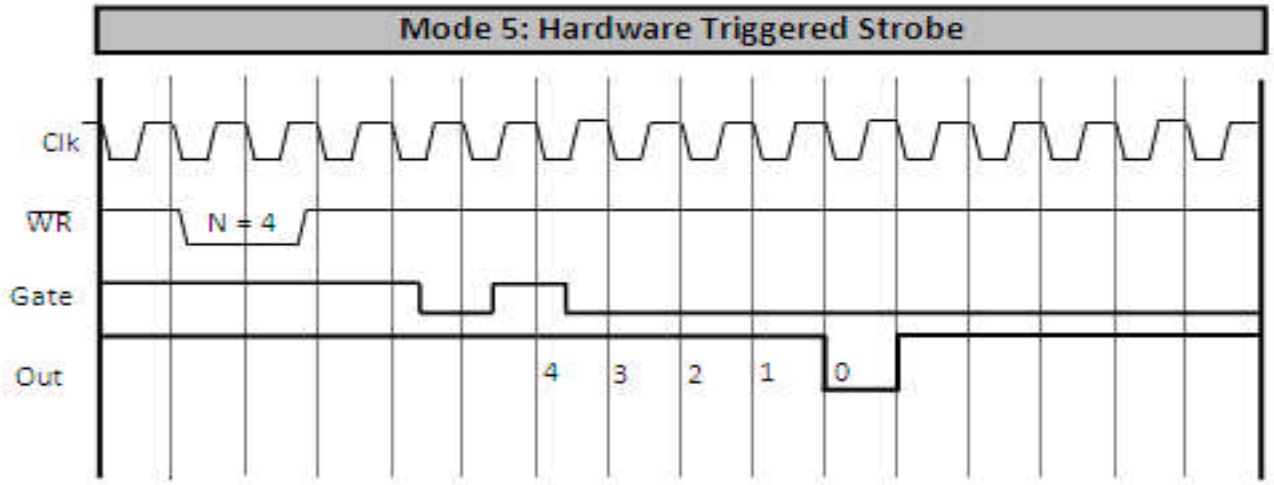


Mode 2: Rate Generator




Mode 3: Square Wave Generator
Clk
WR
N = 4
Gate
Out
4
2
0/4
2
0/4
2
0/4
2
0/4
2
0/4
2
0/4


Mode 4: Software Triggered Strobe
Clk
WR
N = 4
Gate
Out
4
3
2
1
0


Mode 5: Hardware Triggered Strobe
Clk
WR
N = 4
Gate
Out
4
3
2
1
0

8254 Programming

Q 1) **WAP to initialize 8254 for producing a square wave of 1 KHz from an input frequency of 1.625 MHz using Counter-0.**

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

```
Code  SEGMENT
      ASSUME CS: Code

      MOV   AL, 36H
      OUT   86H, AL          // Initialize Control Word = 0011 0111
Back: MOV   AL, 25H
      OUT   80H, AL          // Lower byte of BCD count
      MOV   AL, 16H
      OUT   80H, AL          // Upper byte of BCD count

      INT 03H
Code  ENDS
      END
```

Q 2) **WAP to initialize Counter-1 of 8254 for working like a mono-stable multi-vibrator for a count of 1299H.**

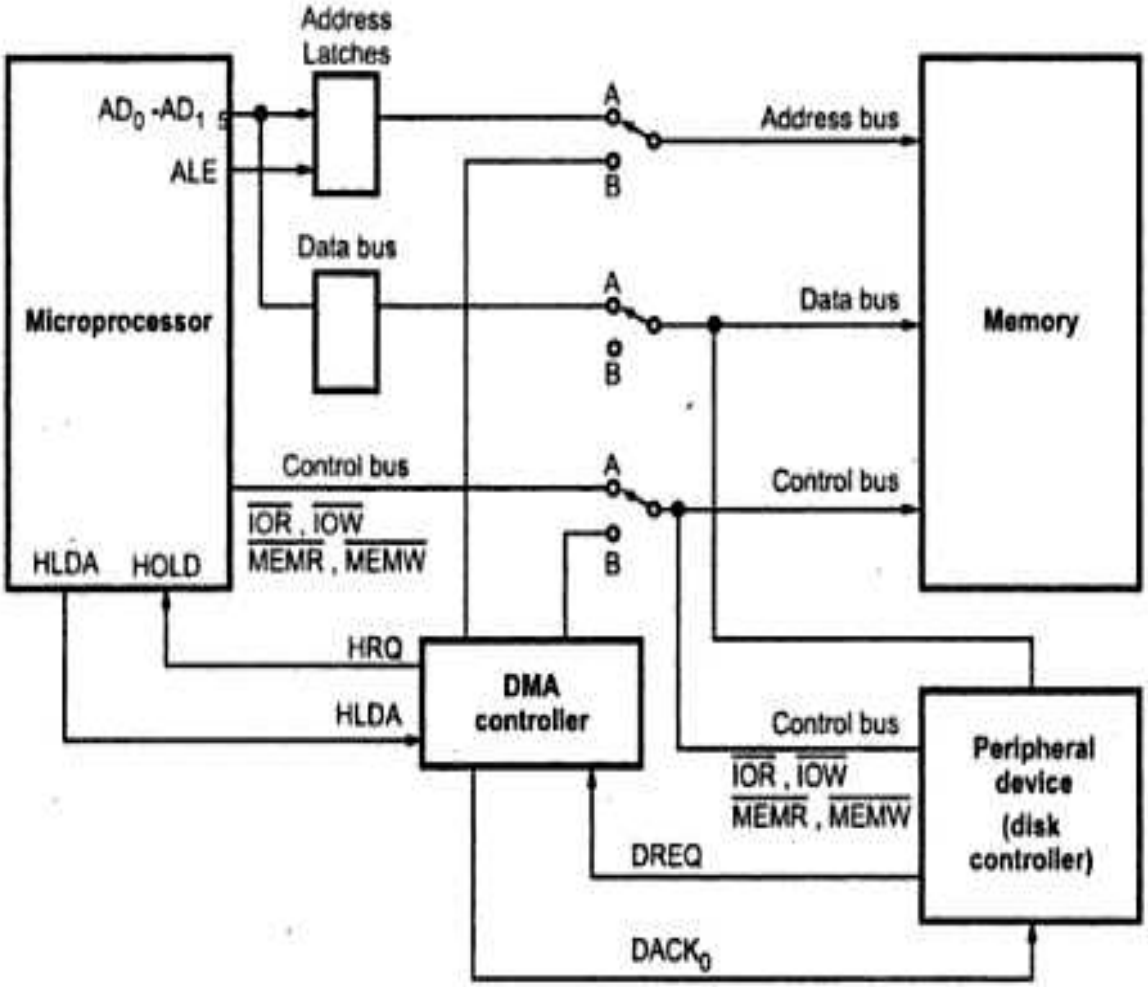
Soln: **For doubts contact Bharat Sir on 98204 08217**

```
Code  SEGMENT
      ASSUME CS: Code

      MOV   AL, 73H
      OUT   86H, AL          // Initialize Control Word = 0111 0011
Back: MOV   AL, 99H
      OUT   82H, AL          // Lower byte of Hex count
      MOV   AL, 12H
      OUT   82H, AL          // Upper byte of Hex count

      INT 03H
Code  ENDS
      END
```

8237 / 8257

DMA CONTROLLER

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.

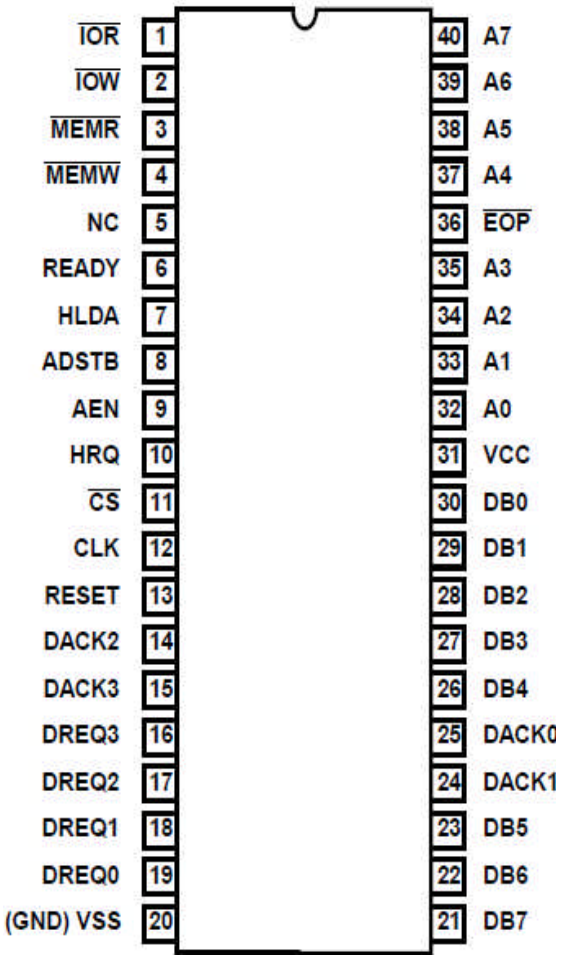
The **sequence of DMA transfer** is as follows:

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.

PIN DIAGRAM OF 8237 DMAC



The **PIN CONFIGURATION** of 8237 are explained as follows:

1) **$DREQ_3$ – $DREQ_0$** (Data Request)
   These **input pins** are used by the **I/O device to request the DMAC** for DMA Transfer.
   These pins can be made **active high / active low through program**.
   By **default**, they are **active high**.
   There are **4 DREQ** pins **as** there are **4 channels** (one per channel).
   At a time only one channel can perform DMA transfer.
   Thus, for **multiple requests**, **priorities** are fixed.
   $DREQ_0$ has the **highest priority** while **$DREQ_3$** has the **lowest priority**.

2) **$DACK_3$ – $DACK_0$** (Data Request)
   These **input pins** are used by the **DMAC to inform the I/O device** that it has acquired the system bus, and hence the **DMA transfer can begin**.
   These pins can be made **active high / active low through program**.
   By **default**, they are **active low**.
   There are **4 DACK pins** as there are **4 channels** (one per channel).

3) **HRQ** (Hold Request)
   This **output** pin is used by the DMAC to **request the µP** to release the system bus.
   It is **connected to** the **HOLD** pin of the µP.

4) **HLDA** (Hold Acknowledge)
   This **input pin** is used by the µP **to inform the DMAC** that it as released to system bus.
   It is **connected** to the **HLDA** pin of the µP.

5) **AEN** (Address Enable)
   This is an **output** pin from the DMAC.
   When this pin is high, **it disconnects the µP from the system bus**.
   It is **also** used to **enable** the **external latch**.

6) **ADSTB** (Address Strobe)
   It is an **output** signal.
   It is used to **strobe** the higher order **address** byte **into the latch**.

7) **DB_7 – DB_0** (Data Bus)
   These are **8 bi-directional data lines** used to connect the internal data bus of 8237 with the external (system) data bus.
   In **idle cycle**, **µP writes or reads** from 8237 **using this bus**.
   In **active cycle**, this bus **carries** the **8 higher order bits** of the 16 bit **address** (the other 8 bits being in the **A_7–A_0**).
   During **memory-to-memory transfer**, this bus **carries** the **data** byte to be transferred.

8) **A_7 – A_4** (Address bits)
   These are **4 output address lines**.
   In **active cycle**, these lines **carry the A_7–A_4 bits of the address** at which the transfer is to be done.
   As this address is generated by the 8237, these are output lines.

9) **A_3 – A_0** (Address bits)
   These are **4 bi-directional address lines**.

In **idle cycle**, **µP sends** the **address A_3–A_0**, to select one of its registers.
Since µP sends the address to 8237, these are input lines.
In **active cycle**, these lines **carry the A_3–A_0 bits of the address** at which the transfer is to be done.
As this address is generated by the 8237, these are output lines.

10) IOR, IOW

These are **bi-directional control lines**.
During **idle state**, the **µP issues** these signals **to read from or write into the 8237** (as the DMAC itself is an I/O device w.r.t. the µP).
During **active state**, the **DMAC issues** these signals **to read from or write into an I/O Device**.

11) MEMR, MEMW

These are **output control lines**.
During **active state**, the **DMAC issues** these signals **to read from or write into the Memory**.

12) EOP

This is a **bi-directional signal** indicating the **end of DMA process**.
During **active cycle**, **after each byte** is transferred through DMA, the **Count Register decrements** by 1.
When the **Terminal Count** is reached, it means that **all** the required **bytes** are **transferred**.
At this point, the **DMAC issues** this signal and the **DMA operation is terminated**.
Hence, it is an output signal.
During the course of the transfer, the **DMA operation can be explicitly terminated by giving this signal** externally to the DMAC.
Hence, it is also an input signal.

13) CLK

This is a **clock-input** signal for the DMAC.
It is usually connected to the system clock.

14) RESET

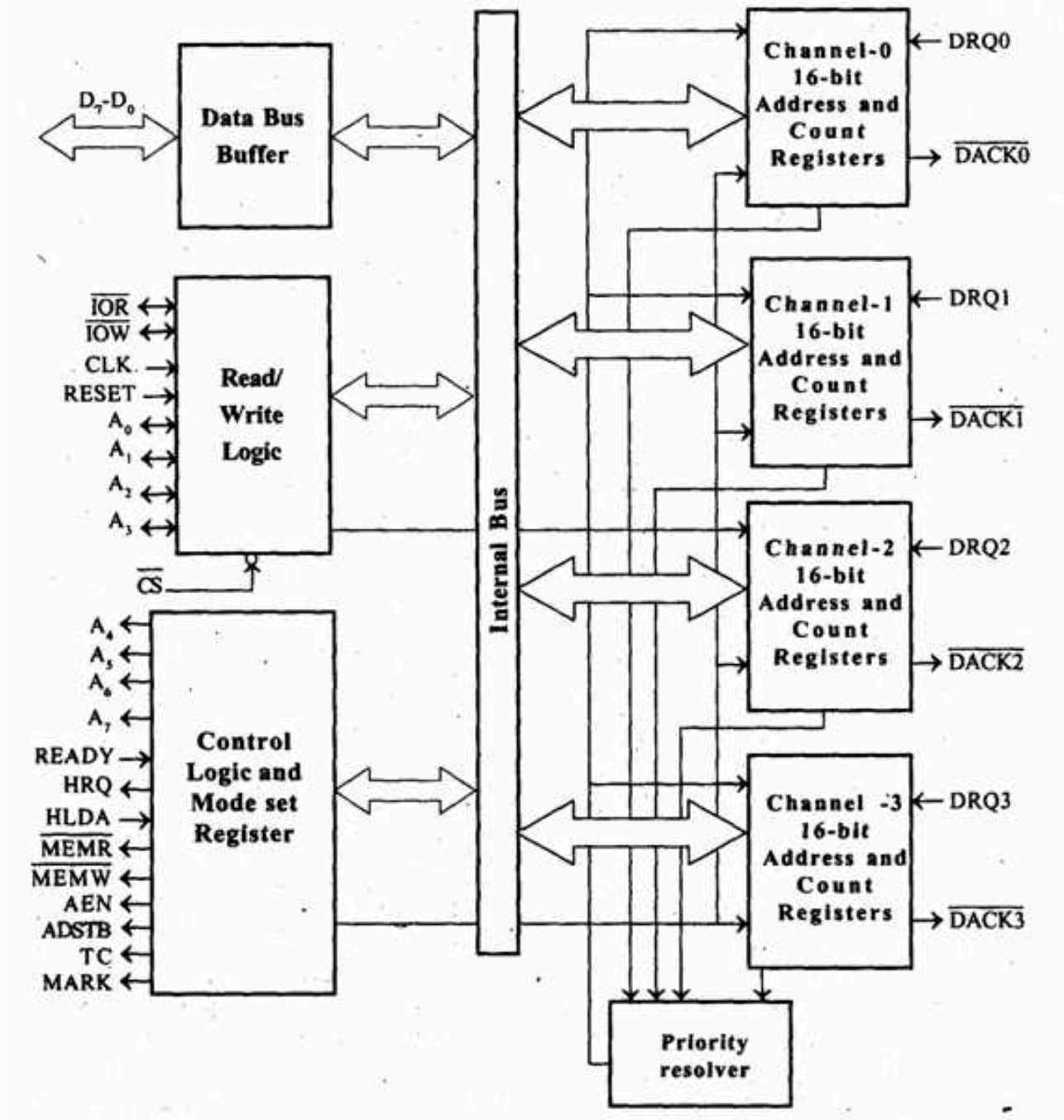
This is a **reset-input** signal for the DMAC.
This signal **clears the internal registers** of the DMAC and causes it to enter **Idle State**.

15) READY

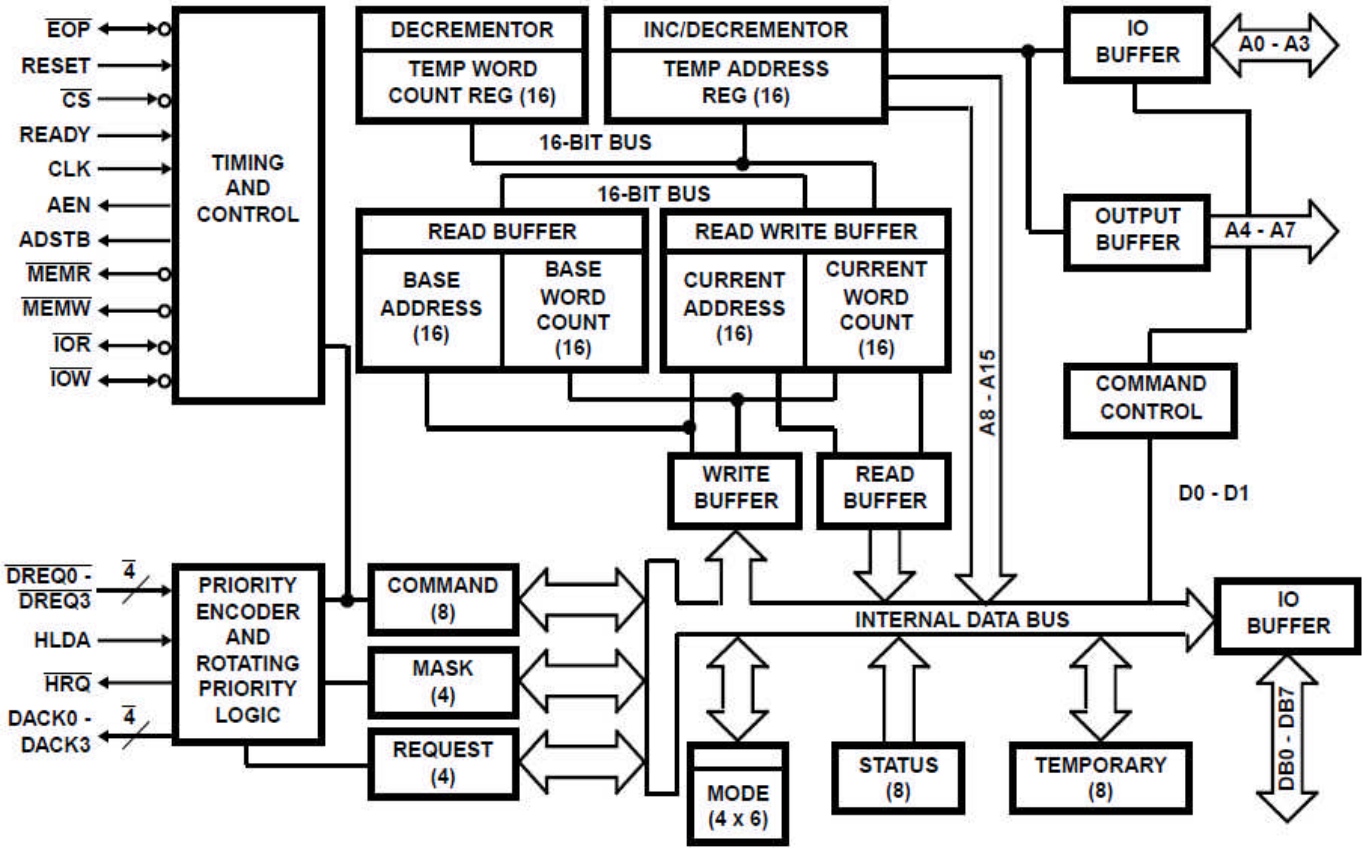
This is an **input** signal to the DMAC.
It is used to **synchronize** the DMAC with "**Slower**" peripherals.
If a peripheral is slow (**not ready**), it put a **low** on this line causing the **DMAC** to **wait**.

Only after the DMAC finds a high on this pin, it executes the DMA operation.

Architecture of 8257



Architecture of 8237



The **INTERNAL REGISTERS (ARCHITECTURE)** of 8237 are explained as follows:

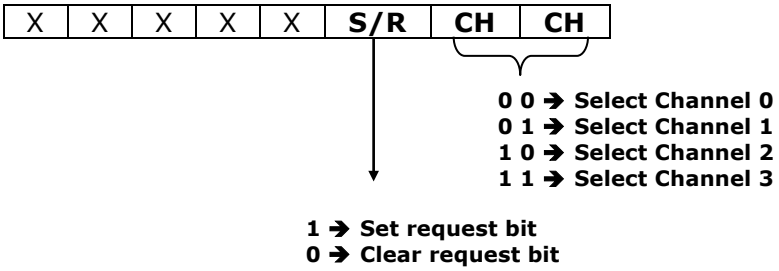
1) **BAR** (Base Address Register)
It is a **16-bit** register.
Each channel has one BAR.
During Idle cycle, **µP writes** the **starting address** of the data transfer into the BAR.
Thereafter, the **contents of BAR do not change** throughout the transfer.

2) **BWCR** (Base Word Count Register)
It is a **16-bit** register.
Each channel has one BWCR.
During Idle cycle, **µP writes** the **number of bytes to be transferred** into the BWCR.
Eg: if **5 bytes** are to be transferred the **µP writes 0004H** into the BWCR.
Thereafter, the **contents of BWCR do not change** throughout the transfer.

3) **CAR** (Current Address Register)
It is a **16-bit** register.
Each channel has one CAR.
It is used to hold the current address for the DMA Transfer.
After each byte is transferred, the CAR is **automatically incremented/decremented**.
The **µP can read** the contents of CAR.

4) CWCR (Current Word Count Register)
It is a **16-bit** register.
Each channel has one CAR.
It is used to hold the current address for the DMA Transfer.
After each byte is transferred, the CAR is **automatically incremented/decremented**.
The **µP can read** the contents of CAR.

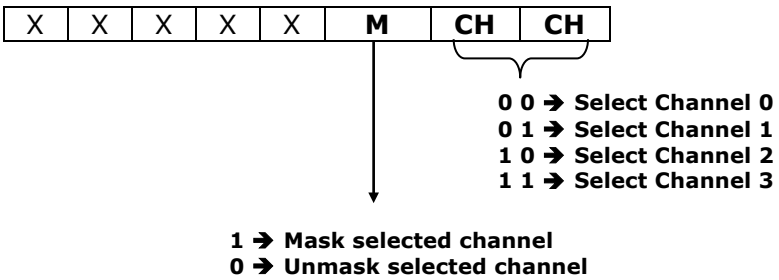
5) Request Register
It is a **4-bit** register having individual bits per channel.
It is used to **store** the **DMA Request** of the peripheral device.
The request bits can be set/reset (through program) as follows:



6) Temporary Register
It is an **8-bit** register.
It is used to **store data temporarily** during **memory-to-memory transfer only**.

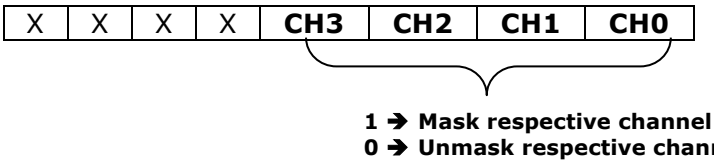
7) Mask Register
It is a **4-bit register** having individual bits per channel.
This register holds the **masking information** about the channels.
By changing these bits the channels can be masked/unmasked in the following two ways:

i. Single Mask Command



Using this command, an **individual channel** can be masked/unmasked.
This action would **not effect the other channels**.

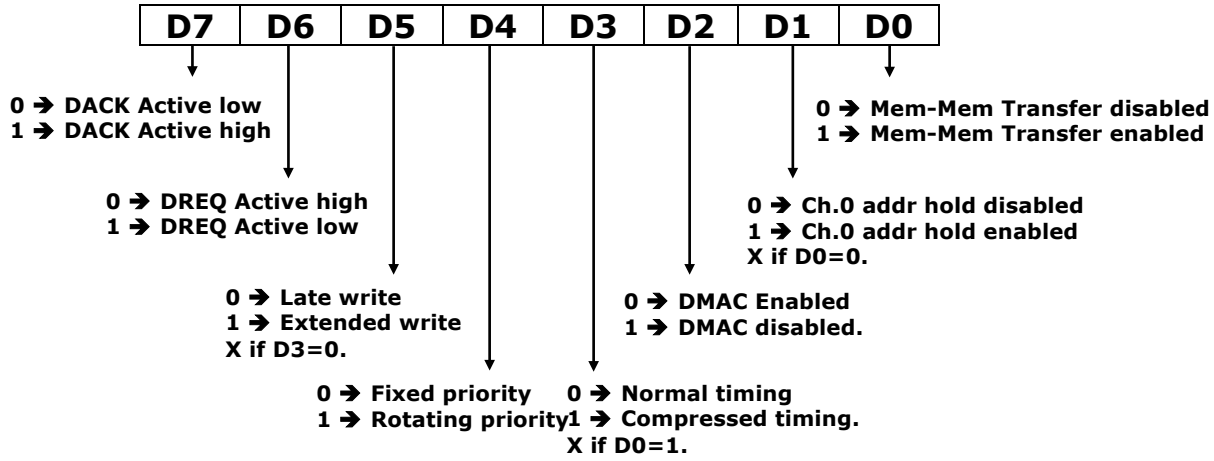
ii. All Mask Command



Using this command, **all channels** can be **simultaneously** masked/unmasked.
This action would **effect all channels** together.

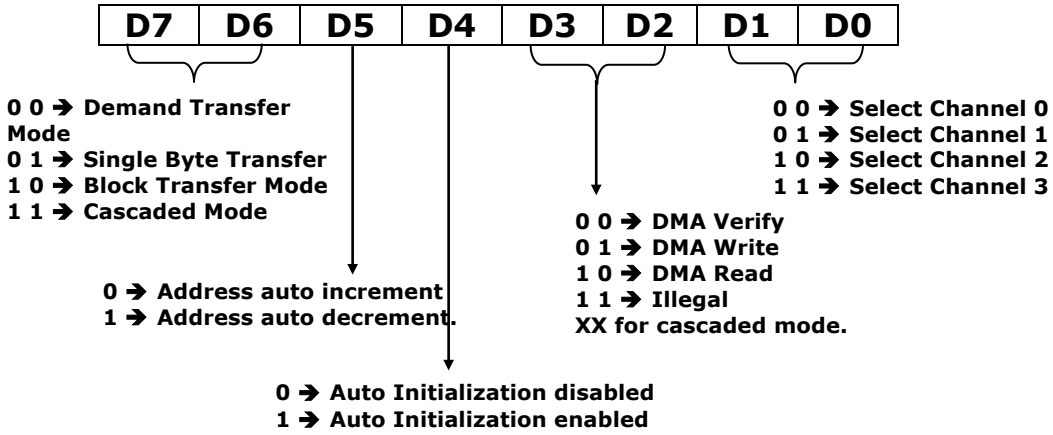
8) Command Register

It is an **8-bit** register.
It is used to store the **command word**.
The **µP writes** the command word.



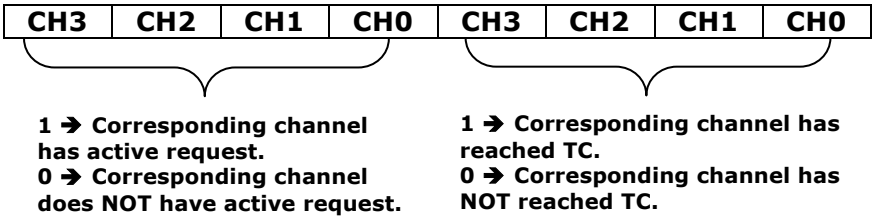
9) Mode Register

It is an **8-bit** register.
Each channel has one Mode Register.
It is used to program the **operational mode** of that channel.



10) Status Register

It is an **8-bit** register.
It stores the status word of the 8237.
The **µP can read** the Status Register**.**



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.

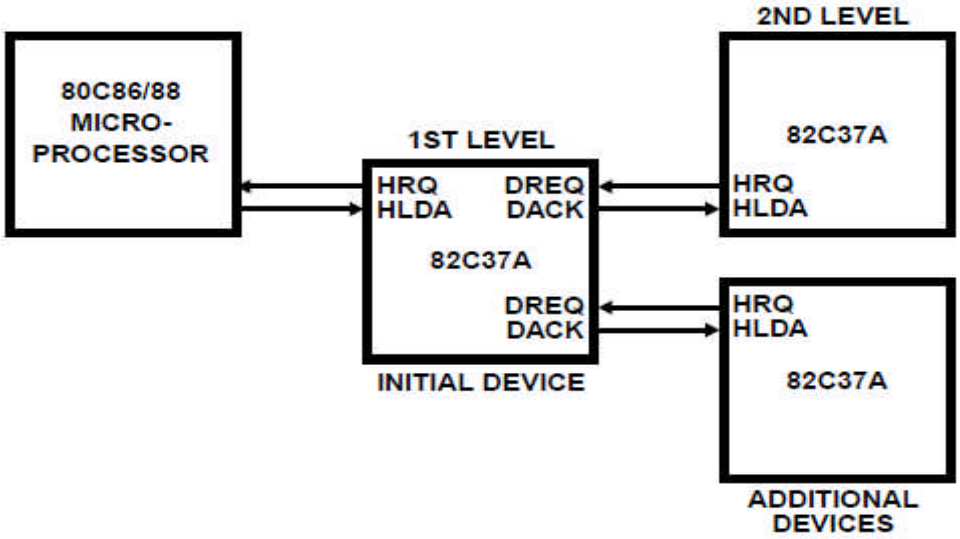
In this mode, the DMAC is programmed to **transfer ALL THE BYTES** in one complete DMA operation.
After a byte is transferred, the **CAR and CWCR are adjusted** accordingly.
The **system bus** is **returned** to the **µP, ONLY after all the bytes are transferred.** I.e. **TC** is **reached or EOP** signal 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.

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.**



Priority Methods of 8237

8237 has **two priority methods**:

1) Fixed Priority.

This is the default mode of 8237.
Here the priorities of the channels are fixed.
Channel 0 has the highest priority, followed by Channel 1, Channel 2 and finally **Channel 3** having the lowest **priority**.

2) Rotating Priority.

It is a **flexible** priority mode.
In this mode, the Channel, which was **most recently serviced, receives the lowest priority**.
This **prevents** any **one Channel from dominating** the system.
Eg: If **Channel 0 was just serviced** then Channel 0 **gets lowest priority**.
Channel 1 gets highest priority, followed by Channel 2, Channel 3 and finally Channel 0.

Before CH.0 is serviced

Highest	**CH.0**
	CH.1
	CH.2
Lowest	CH.3

← Active DMA request

After CH.0 is serviced

CH.1	Highest
CH.2	
CH.3	
CH.0	Lowest

INITIALIZATION OF 8237

1) **Write** a control word in the **Mode Register** that would specify the following:
2) **Write** a control word in the **Command Register** that would specify:
3) **Write** the **starting address** of the transfer in the channel's Memory Address Register.
4) **Write** the **count** (no. of bytes to be transferred) in the channel's Count Register

INTERFACING 8237/ 8257 DMAC WITH 8086

DMA means **transferring data directly between memory and I/O.**
DMA transfers are **very fast** as compared to microprocessor based transfers due to two reasons.
1. They are **hardware based** so no time is wasted in fetching and decoding instructions.
2. Transfers are **directly between memory and I/O** without data going via the microprocessor.

To Perform a DMA transfer we need a **DMA Controller like 8237/ 8257.**
It is capable of taking control of the buses from the microprocessor.
The process is performed as follows.

1. By Default **Microprocessor is the bus master**.
2. To start a DMA based transfer, **microprocessor programs two registers inside the DMAC called CAR and CWCR** giving the starting address and the number of bytes to be transferred.
3. DMAC now ensures that the I/O device is ready for the transfer by checking the DREQ signal.
4. **If DREQ=1, then DMAC gives HOLD signal** to the Microprocessor requesting control of the system buses.
5. **Microprocessor releases control of the bus** after finishing the current machine (bus) cycle.
6. Microprocessor **gives HLDA** informing DMAC that it is now the bus master.
7. **DMAC issues DACK#** (by default active low, but can be changed) to I/O device indicating that the transfer is about to begin.
8. Now DMAC **transfers one byte in one cycle**.
9. After every byte is transferred the **Address register and Count register are decremented by 1**.
10. This repeats till Count reaches **"0"** also called **Terminal Count**.
11. Now the **transfer is complete**.
12. DMAC **returns the system bus to Microprocessor by making HOLD = 0**.
13. Microprocessor once again **becomes bus master**.

(Additional detail, include only if you have the time, else simply useful for Knowledge/ Viva)

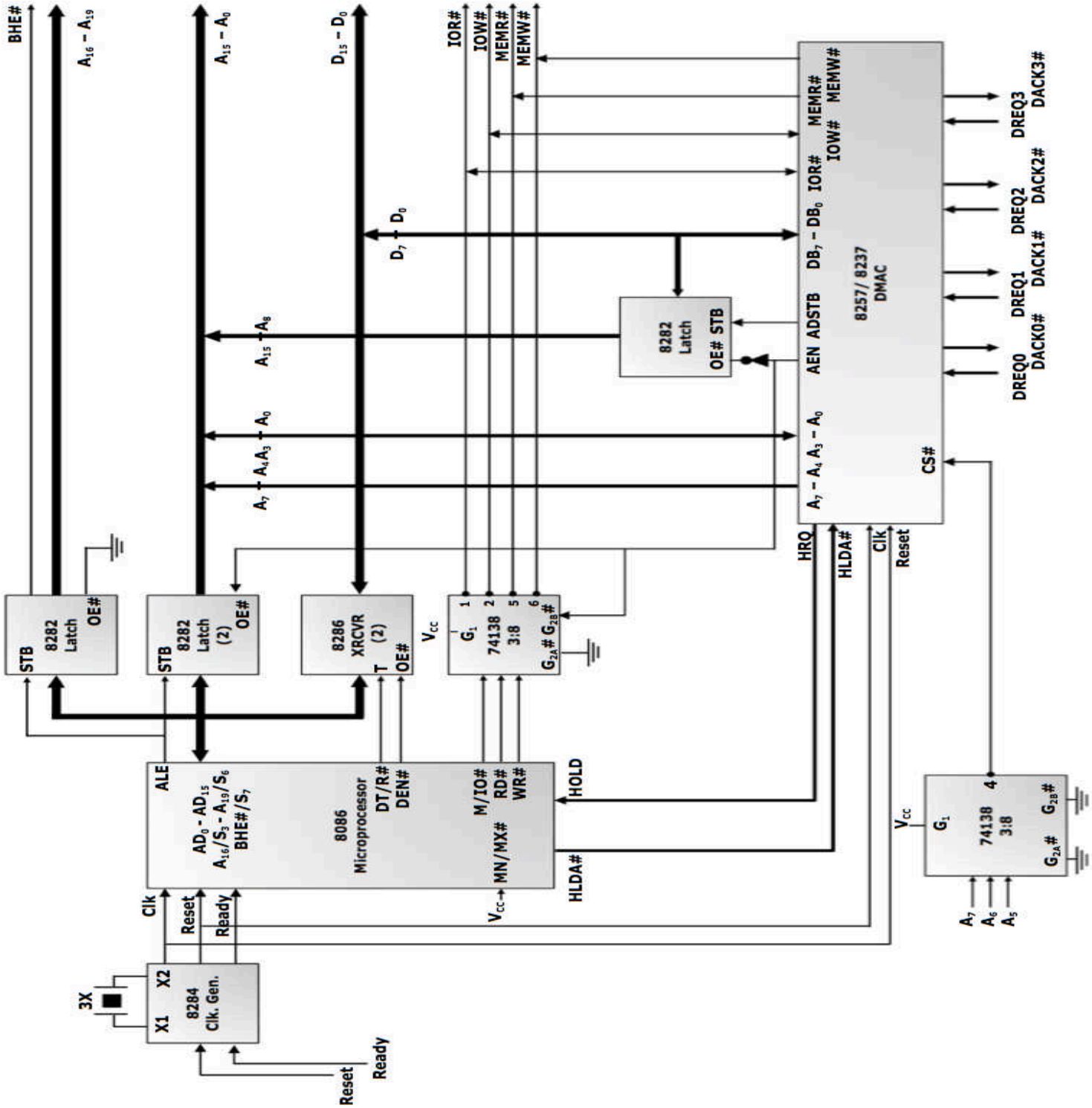
14. *When microprocessor is the bus master, mic**roprocessor controls the buses**.*
    *Hence the **upper latches and decoder are enabled**.*
    *That's why **by default, in DMAC, AEN is 0 indicating that microprocessor is the bus master**.*
15. *When **DMAC becomes bus master, AEN becomes 1**.*
    *Now microprocessor's **latch and decoder are disabled**.*
    *Instead **DMAC will issue address and control signals**.*
16. *DMAC gives **16-bit address**.*
17. ***A0-A7** are not multiplexed so they are **directly connected to address bus**.*
18. ***A8-A15 and D0-D7 are multiplexed as DB0-DB7.***
    They are given to the lower latch.
19. *The **ADSTB** (Address Strobe, same as ALE), helps the latch to **capture address that's A8-A15**.*
    ***This connects to the address bus**.*
20. *The part **not captured** by the latch, **carries data D0-D7**.*
    ***This is connected to the data bus**.*

21. ***A0-A3 are bidirectional*** *as they are used by the microprocessor for* ***internal selection while giving commands****.*
22. *Similarly,* ***IOR# (IOR bar) and IOW# are bidirectional*** *as DMAC is an I/O device so it also* ***receives IOR# and IOW# from the microprocessor****, apart from* ***itself generating the four control signals MEMR#, MEMW#, IOR# and IOW#****.*
23. *It is important to note that* ***DMAC can only produce a 16 bit address*** *whereas the actual address is 20 bit. Hence the* ***upper 4 bits of the address, A16-A19, have to be still produced by the microprocessor****.* ***They are stored in the topmost latch, which is not disabled when AEN becomes 1****.*

I/O map DMAC 8237/ 8257	A7	A6	A5	A4	A3	A2	A1	A0	I/O Address
16 Internal addresses of DMAC	0	1	0	**0**	**0**	**0**	**0**	0	**80**
	0	1	0	**1**	**1**	**1**	**1**	0	**9E**

List of 16 address (Not important for exam)

OPERATION	A3	A2	A1	A0	$\overline{IOR}$	$\overline{IOW}$
Read Status Register	1	0	0	0	0	1
Write Command Register	1	0	0	0	1	0
Read Request Register	1	0	0	1	0	1
Write Request Register	1	0	0	1	1	0
Read Command Register	1	0	1	0	0	1
Write Single Mask Bit	1	0	1	0	1	0
Read Mode Register	1	0	1	1	0	1
Write Mode Register	1	0	1	1	1	0
Set First/Last F/F	1	1	0	0	0	1
Clear First/Last F/F	1	1	0	0	1	0
Read Temporary Register	1	1	0	1	0	1
Master Clear	1	1	0	1	1	0
Clear Mode Reg. Counter	1	1	1	0	0	1
Clear Mask Register	1	1	1	0	1	0
Read All Mask Bits	1	1	1	1	0	1
Write All Mask Bits	1	1	1	1	1	0


BHE#
A16 – A19
A15 – A0
D15 – D0
IOR#
IOW#
MEMR#
MEMW#
D7 – D0
A15 – A8
A7 – A4 A3 – A0
8282 Latch
STB
OE#
8282 Latch (2)
8286 XRCVR (2)
T
VCC
74138 3:8
G1
G2A# G2B#
8257/ 8237 DMAC
DB7 – DB0
AEN ADSTB
A7 – A4 A3 – A0
CS#
DREQ0 DREQ1 DREQ2 DREQ3
DACK0# DACK1# DACK2# DACK3#
HRQ
HLDA#
Clk
Reset
8086 Microprocessor
ALE
AD0 - AD15
A16/S3 - A19/S6
BHE#/S7
DT/R#
DEN#
M/IO#
RD#
WR#
MN/MX#
HOLD
Ready
8284 Clk. Gen.
X1 X2
3X
A7 A6 A5

MULTIPROCESSOR SYSTEMS

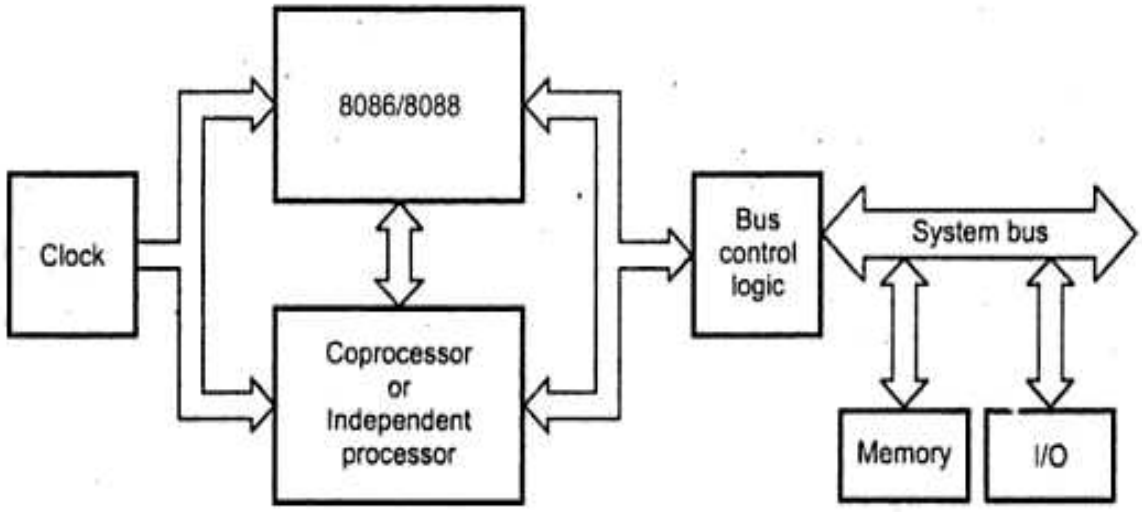
A Multiprocessor system has **more than one Processor** in it.
The **Maximum Mode** of 8086 is designed to implement Multiprocessor systems.

Advantages of Multiprocessor systems

1) It **improves** the **performance** and **efficiency** of the system.
2) **Increases speed** of the system due to multiprocessing.
3) The system becomes more **cost effective**
   On failure, it is cheaper and easier to **replace only** the **faulty processor**.

Types of Multiprocessor Configurations
There are two main types: **Closely Coupled** and **Loosely Coupled** Configuration.

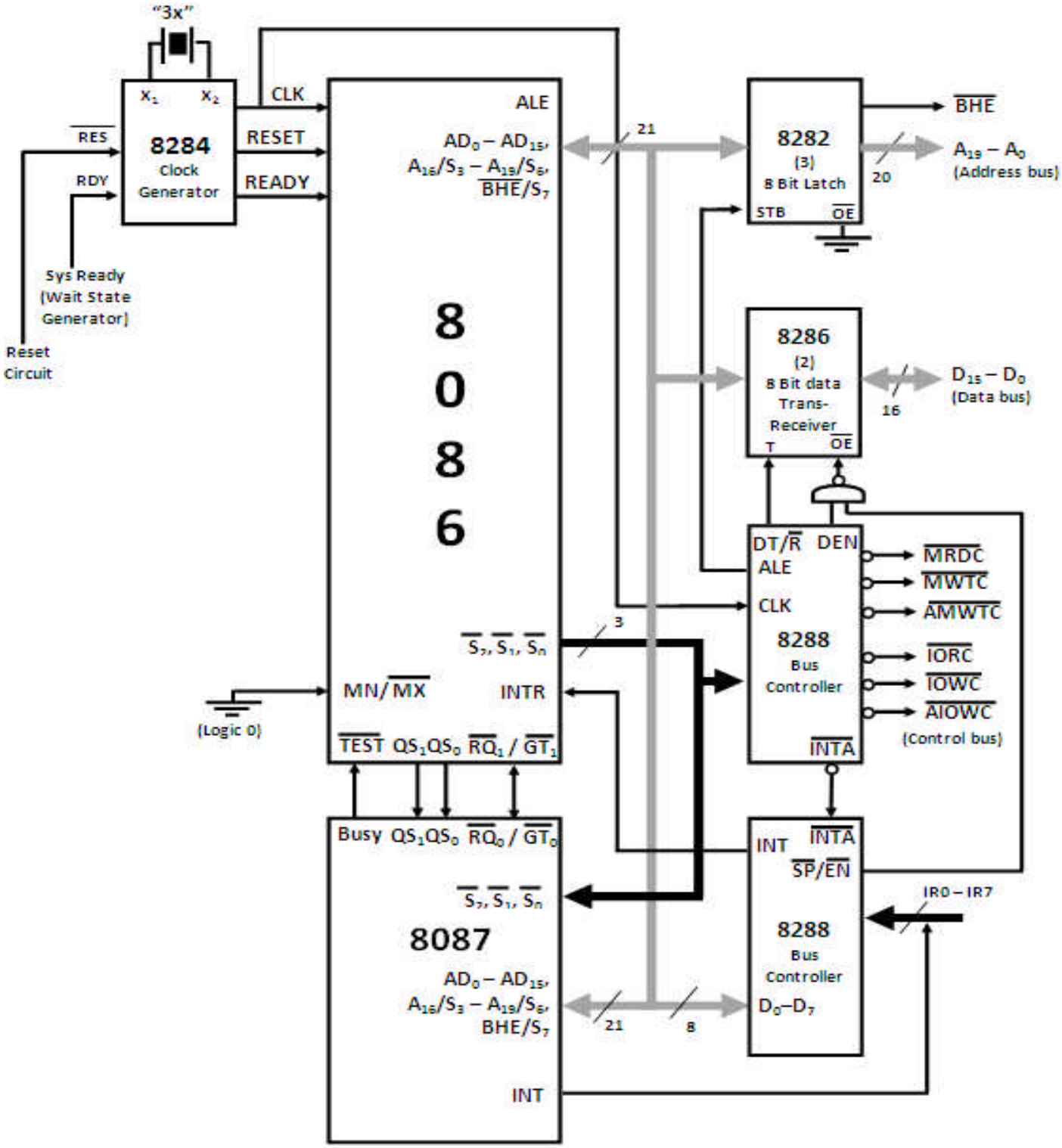
1) CLOSELY COUPLED CONFIGURATION / TIGHTLY COUPLED



- Here, all processor s **share** a **common Memory**, **I/O, Bus Control Logic** and **Clk Generator**. Hence the system is said to be "Closely Coupled".
- **8086** is called the **Master** (Host) and the **other processors** are called **Slaves**.
- By default, the Master controls the common shared bus, and hence the shared components.
- The other processors have to send a request signal to the master to access the shared components.
  #Please refer Bharat Sir's Lecture Notes for this ...
- The **advantage** of such systems is that they are quite **simple** to **design** and are **cheaper**.
- Main **disadvantages** are:
  - As all processors try to access the same Memory, I/O etc through the same Single Shared Bus, **traffic on the shared bus is very high**.
  - If **one component fails** (eg: shared memory, or shared bus), the **entire system may fail**.

Example: 8086 – 8087 Interface

a) 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.

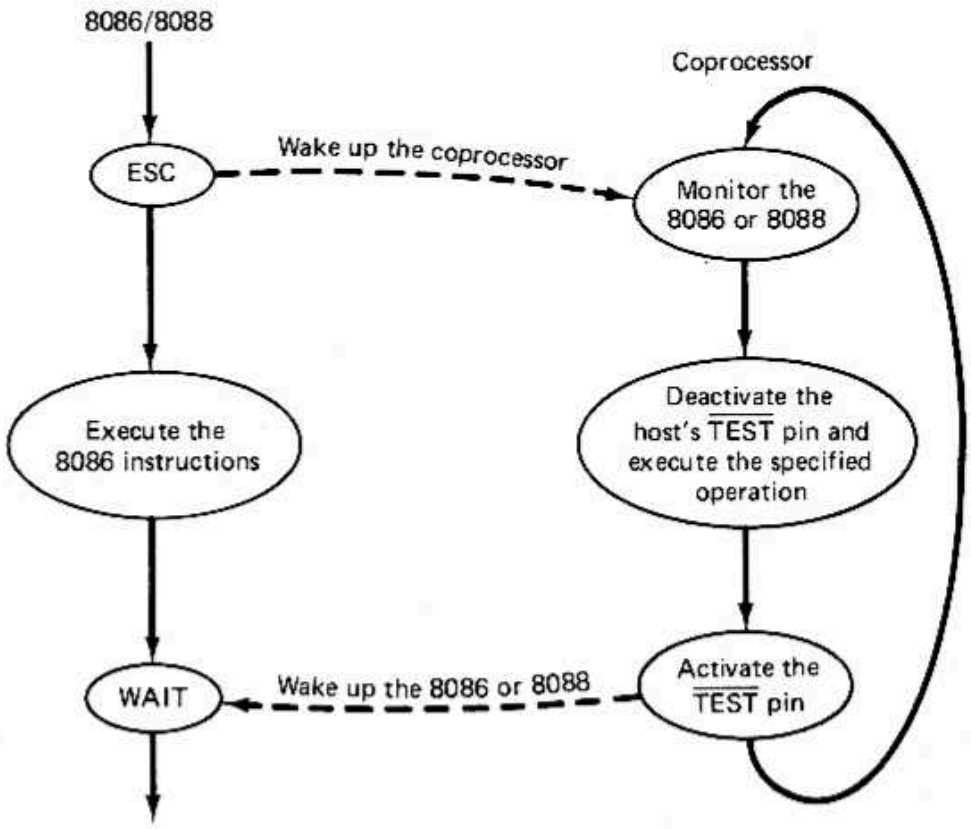
12) During the execution **if** an **exception occurs**, which is **unmasked**, **8087 interrupts μP** using the **INT** o/p **pin through** the **PIC 8259**.

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)



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)

Numericals on Floating Point Conversions *(10m)*

Steps for conversion:

1) Convert the given number into binary.
2) Normalize the number.
3) Calculate the biased exponent.
4) Convert the biased exponent into binary.
5) Represent in the required format.
6) (Optional) Convert the number into hexadecimal form.

1) Convert 2A3BH into Short Real and Temp Real formats {Exam question}

Short real:

Converting the number into binary we get:
0010 1010 0011 1011

Normalizing the number we get:

$$(-1)^{0} \times 1.0101000111011 \times 2^{13}$$

Here S = 0; M = 0101000111011; True Exponent = 13.

Bias value for Short Real format is 127:
Biased Exponent (BE) = True Exponent + Bias
= 13 + 127
= 140.

Converting the Biased exponent into binary we get:
Biased Exponent (BE) = (1000 1100)

Representing in the required format we get:

0	**10001100**	**010100011101100…**
S	Biased Exp	Mantissa
(1)	(8)	(23)

Converting the number into hexadecimal form we get:
4628EC00H … 32 bits.

Temp real:

Bias value for Temp Real format is 16383:
Biased Exponent (BE) = True Exponent + Bias
= 13 + 16383
= 16396.

Converting the Biased exponent into binary we get:
Biased Exponent (BE) = (100 0000 0000 1100)

Representing in the required format we get:

0	100000000001100	**1**010100011101100…
S	Biased Exp	Mantissa
(1)	(15)	(64)

Converting the number into hexadecimal form we get:
400C A8EC 0000 0000 0000H … 80 bits.

2) Convert $(12.125)_d$ into Temp Real format {Exam question}

Temp real:

Converting the number into binary we get:
1100.001

Normalizing the number we get:
$(-1)^{\mathbf{0}}$ **x** $1.\mathbf{100001} \times 2^{\mathbf{3}}$
Here S = 0; M = 100001; True Exponent = 3.

Bias value for Temp Real format is 16383:
Biased Exponent (BE) = True Exponent + Bias
= 3 + 16383
= 16386.

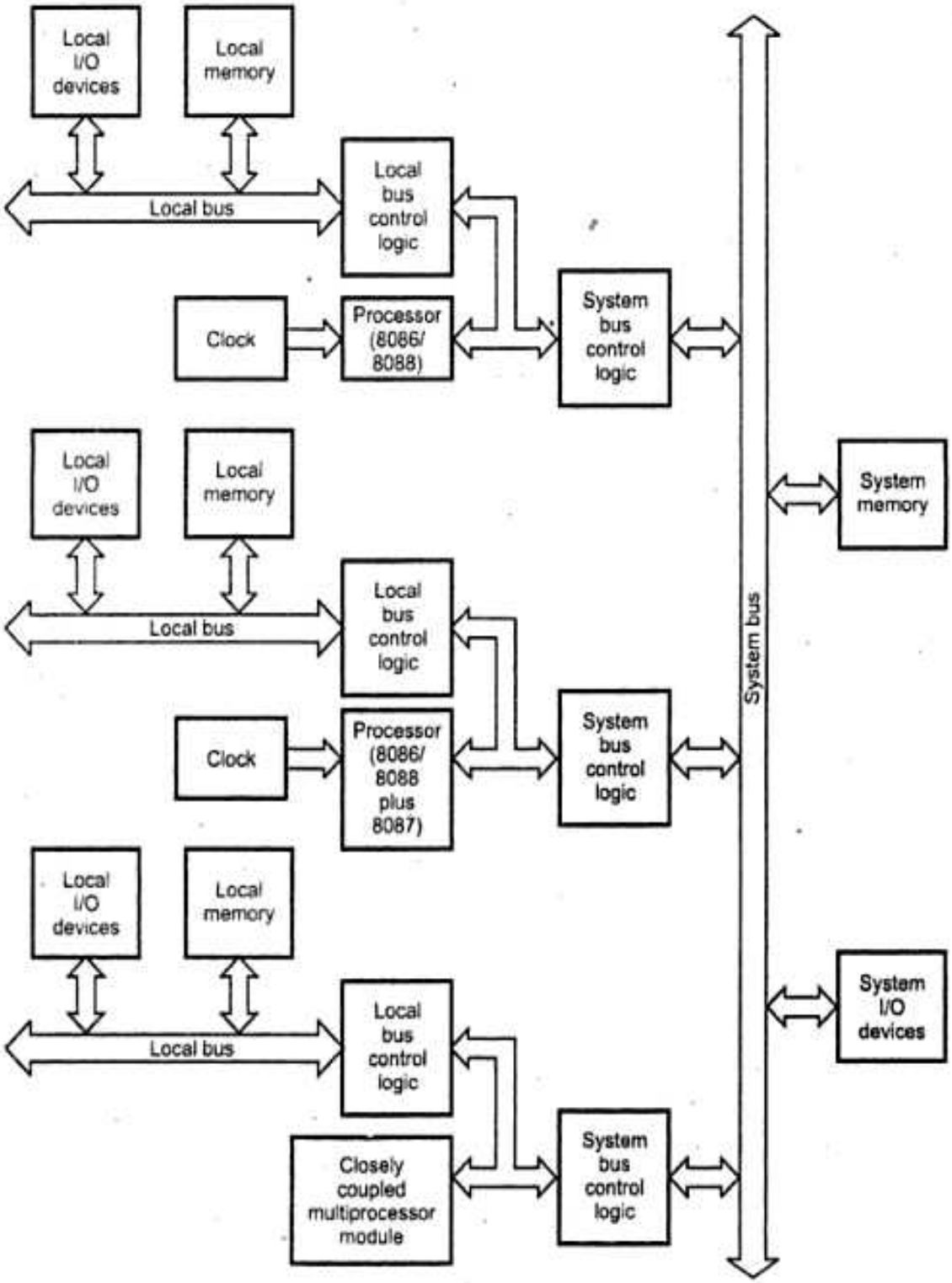
Converting the Biased exponent into binary we get:
Biased Exponent (BE) = (100 0000 0000 0010)

Representing in the required format we get:

0	100000000000010	**1**10000100000…
S	Biased Exp	Mantissa
(1)	(8)	(23)

Converting the number into hexadecimal form we get:
4002 C200 0000 0000 0000H … 80 bits.

LOOSELY COUPLED CONFIGURATION (In the exam draw only Two Modules)



- A Loosely Coupled system can be **divided into modules**.
- Each module is like a Closely Coupled system in itself.
  i.e. **each module has** a ***local resources*** like local **Memory** and local **I/O devices.**
  Hence the system is said to be "Loosely Coupled".
- Thus a **module** may consist of a **single 8086**, or **8086 with 8087** or **8089** or both.
- **Communication within** a **module** happens through the **local bus.**
  Access to this **local bus** is controlled **by** the **Local Bus Control Logic**.
- **All Modules** are **connected to System memory** and **System I/O** through the **System Bus**.
  Access to this **system bus** is controlled by the **System Bus Control Logic**.
- As each module has local resources (Memory and I/O), most of its work happens with the module itself. For this, the modules do not require the system bus.
- Thus traffic on the **system bus** is **only for Inter-Module** communication

Advantage of Loosely Coupled Configuration:
- Very **powerful systems** can be made using this configuration.
- The system becomes **more reliable**.
  Even if one module fails the others can continue their work as they have local resources.
- **High** degree of **multiprocessing** can be achieved.
- **Upgrading** the system becomes **easy** as more processors can be easily added.

Disadvantage of Loosely Coupled Configuration:
- Circuit is **large** and quiet **complex**.
- **Cost is higher** as compared to Closely Coupled systems.

PROBLEMS IN MULTIPROCESSOR SYSTEMS

1) BUS CONTENTION / BUS ARBITRATION / PRIORITY RESOLVING SCHEMES

- Bus contention is the problem of **simultaneous bus requests by more than one processor**.
- In multiprocessor systems, **more than one processors** try to **access** the shared resources through the **system bus**. Hence, it is **necessary** to have **extra bus logic** to ensure that only **one processor** has **access** to the system bus **at one time**.
- The **8289 Bus Arbiter** is used to resolve this bus contention.
- The following **three methods** are used **to resolve bus contention**:

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 ($\mathbf{2^3 = 8}$).
- 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.

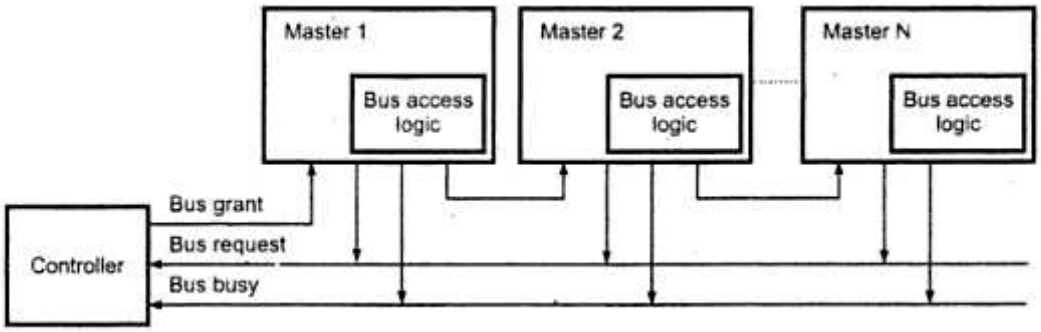
Advantage:

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

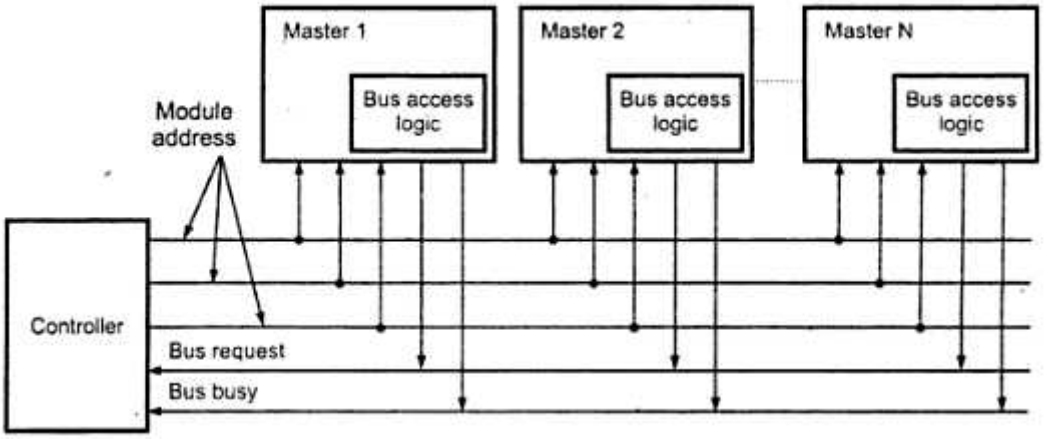
Disadvantage:

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

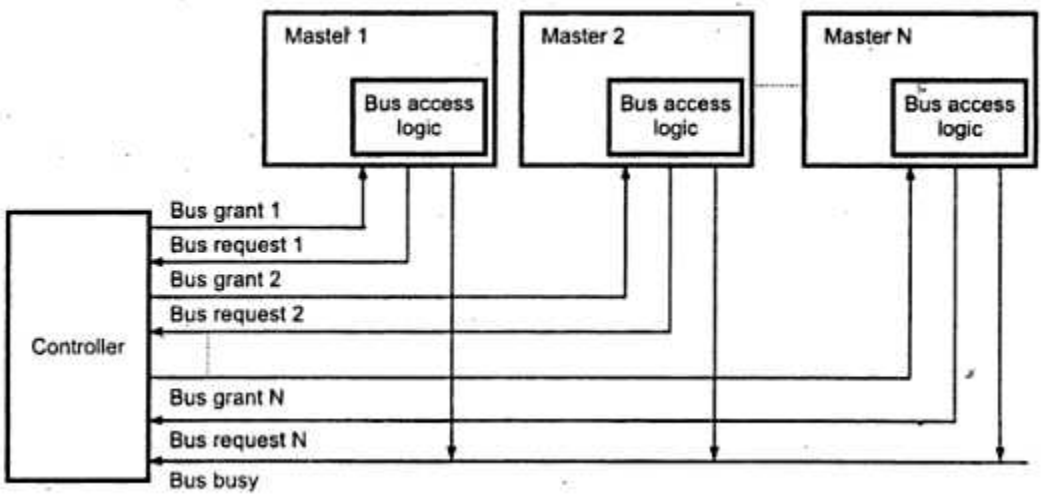
Daisy Chaining



Polling



Independent Requests



Inter-Processor Communication

- When **one processor uses** a **shared** system **resource**, the **other processors should not** use the same resource. If this is not done, error may occur.
- Thus a technique known as **Mutual Exclusion** is required which, at a time, permits only one processor to access the common resources. **For doubts contact Bharat Sir on 98204 08217**
- Here a flag is stored at a memory location known as **Semaphore**, in the system memory.
- If this flag is **set** (1) ➔ the common **resource** is **free**.
- If this flag is **reset** (0) ➔ the common **resource** is **busy** i.e. used by another processor.
- Hence, before accessing the common resources a processor checks this flag as follows:

```
        MOV AL, 00
Retry:  XCHG Semaphore, AL
        TEST AL, AL
        JZ Retry
                } Program that uses the common resources.
        MOV Semaphore, 01
```

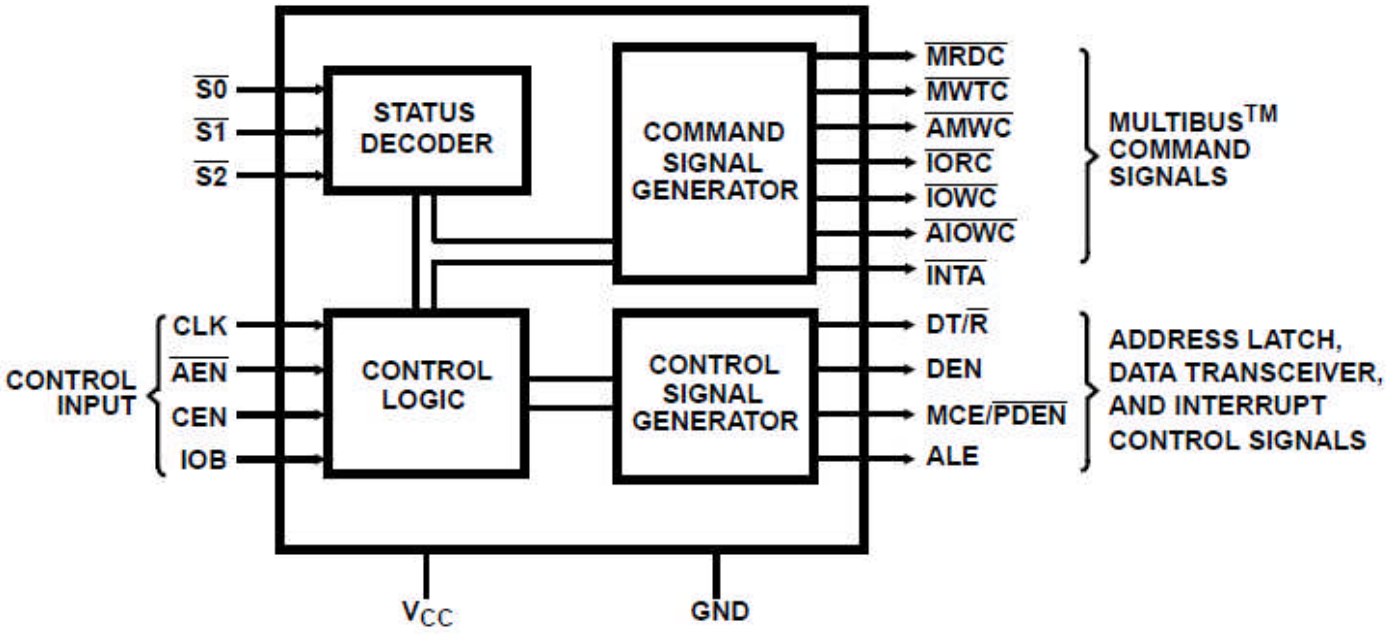
- Due to this program, only one processor can use the common resources at one time.
- However, there is **one drawback**.
  If **two processors** perform the above exchange instruction **one by one**, chances are that both processors will enter the common resources together.
  This is because the **XCHG** instruction involves **two bus cycles**.
  This problem is shown below:
  Assume **Semaphore** is **set** i.e. resource is **FREE**.
  Both processors execute **XCHG Semaphore, AL** simultaneously.
  - During first bus cycle, processor **A gets semaphore = 1** in it.
  - During next bus cycle, processor **B gets semaphore = 1** in it.
  - During next bus cycle, processor **A puts Semaphore = 0**.
  - During next bus cycle, processor **B puts Semaphore = 0**.

  Now for both processors the **TEST AL, AL** will produce non-zero result.
  Hence, **both** processors **enter** the **common resource simultaneously.**
- To avoid this problem the **LOCK** prefix is used with XCGH instruction as shown below
  Retry: LOCK XCHG Semaphore, AL #Please refer Bharat Sir's Lecture Notes for this ...
- ∴ Processor B cannot take the system bus from processor A till its XCHG instruction is complete, and by then the Semaphore has 00.
- Thus, **only one processor** enters the common resources **at one time**.

2) **Bus Congestion** (Bus Traffic)

- As the number of modules in the system increase, there is a problem of Bus Congestion i.e. the **modules have to wait for a long time** for getting control over the System Bus.
- This delay **reduces** the **efficiency** of the system.
- As a **solution, local memory** and **local I/O** is provided to each module, which are accessed internally by the module using a local resident Bus. **For doubts contact Bharat Sir on 98204 08217**
- Now the **modules need** the **System Bus only for Inter-Modular** communication.
- Therefore, each module needs the system bus for a less time so, the problem of Bus Congestion is reduced.

8288 BUS CONTROLLER



- In **maximum mode** of 8086, its **$\overline{S_2}$, $\overline{S_1}$, $\overline{S_0}$** lines are **connected to** the **8288** Bus Controller.
- 8288 decodes these lines to produce the control signals as follows

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	**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

Status Input Signals

1) $\overline{S_2}$, $\overline{S_1}$, $\overline{S_0}$

These are used to provide the input for generating the Multibus Command Signals.

Multibus Command Signals

1) M $\overline{RD}$ C, MWTC

It is used to read data from / Write data into memory respectively.

2) IORC, IOWC

It is used to read data from / Write data into an I/O Port respectively.

3) AMWTC, AIOWC

This is similar to MWTC / IOWC except that it is activated one clock cycle earlier.

4) $\overline{INTA}$

This is an output signal, used to get the vector number form the interrupting device.

Output Control Signals

1) DEN

This signal enables the bi-directional data trans-receivers in the system.

2) DT/ $\overline{R}$

This signal controls the direction of the data trans-receivers in the system.

3) ALE

Used to latch the address from the multiplexed address-data bus.

4) MCE/PDEN

In **System Bus Mode** (IOB=0) it acts as *Master Cascade Enable*. It **controls cascaded 8259**.
In **IO Bus Mode** (IOB=1) it is *Peripheral Data Enable*. It **enables IO Bus trans-receivers**.

Input Control Signals

1) CLK

It is the System Clock from the 8284 Clock Generator.

2) CEN, IOB, AEN

These are input signals used to determine the mode of operation of 8288 as follows

CEN	IOB	AEN	Description
1	**1**	X	– **I/O Bus Mode** – All control signals enabled
1	**0**	**1**	– **System Bus Mode** – All **control signals disabled.** Bus controlled by **another master**
1	**0**	**0**	– **System Bus Mode** – All **control signals Enabled.**Bus **free** for use
0	X	X	– 8288 **disabled**

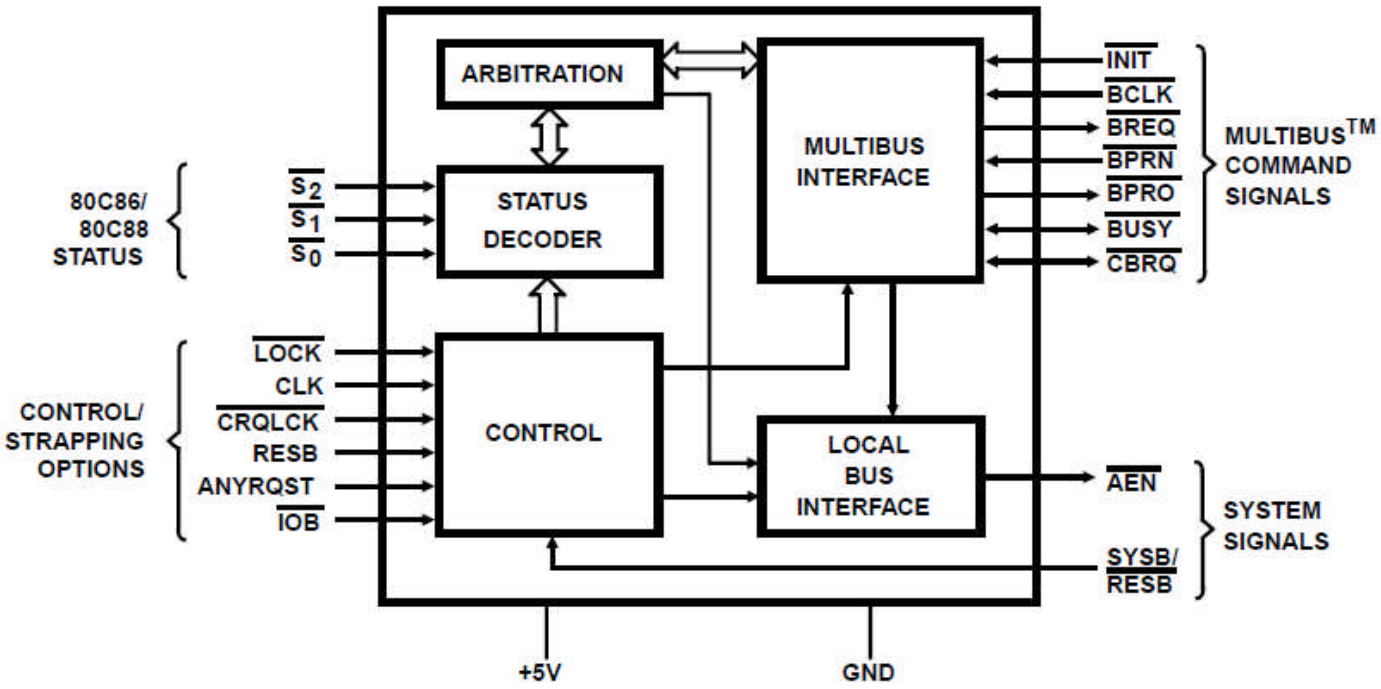
Thus 8288 works in 2 modes:

IOB = 0 --- **System Bus Mode** (Resident Bus Mode).
Here DEN = 1 for all instructions.

IOB = 1 --- **IO Bus Mode** #Please refer Bharat Sir's Lecture Notes for this ...
Here for memory instructions
DEN = 1 (enable memory data transreceiver)
PDEN = 1 (disable IO data transreceiver)
Here for I/O instructions
DEN = 0 (disable memory data transreceiver)
PDEN = 0 (enable IO data transreceiver)

8289 BUS ARBITER

The 8289 Bus Arbiter is used to provide access to the shared system bus.
It provides the bus request signal and accepts the bus grant signal.
It resolves priority during simultaneous bus requests.



- **CLK**
  It is the system clock input from 8284. It is used to synchronize 8289 with 8086.

- **BCLK** (Bus Clock)
  It is the Multi Master system bus clock signal used to synchronize 8289 with system bus.

- **$\overline{\text{LOCK}}$**
  This is an input signal generated by the processor, which controls the system bus.
  It prevents 8289 from surrendering the System bus to another master regardless of priority.

- **CRQLCK** (Common bus request lock)
  This is an input signal. When 0, it prevents 8289 from surrendering the system bus for a request made through CBRQ input.

- **$\overline{S_2}$, $\overline{S_1}$, $\overline{S_0}$**
  These are i/p signals from 8086.
  8289 monitors these lines to determine when would 8086 require the system bus and accordingly gives the request signal or releases the system bus.
- **RESB** (Resident Bus Mode)

This signal is used to select the Multi Master system bus or the Resident bus.
If this signal is high, the selection is based on the **SYSB/RESB** pin.
If low, the SYSB/RESB pin is ignored.

- **SYSB/RESB** (System Bus/ Resident Bus)
  This is an input signal from the address decoding circuit, which determines when the Multi Master system bus is requested or surrendered.
  1 ➔ 8289 requests Multi Master system bus.
  0 ➔ 8289 requests Resident Bus.

- **IOB** (I/O Bus Mode)
  It is used to configure 8289 to operate in systems having both IO bus and Multi Master system bus.

- **ANY RQST** (Any Request)
  If high 8289 has to recognize any request that occurs on the CBRQ.

- **AEN** (Address Enable)
  It is a very important output signal.
  When 8289 uses the system bus ---- it is low (Makes 8288 enable its control signals).
  When 8289 is not using the system bus ---- it is high (8288 disables its control signals).

- **INIT** (Initialize)
  It is an input signal used to reset (initialize) 8289.

- **BREQ** (Bus Request)
  It is an output signal used by 8289 to request for the Multi Master system bus.

- **BPRN** (Bus Priority In)
  It is an active low input signal sent to the arbiter, indicating that it may occupy the Multi Master system bus from the next clock cycle.

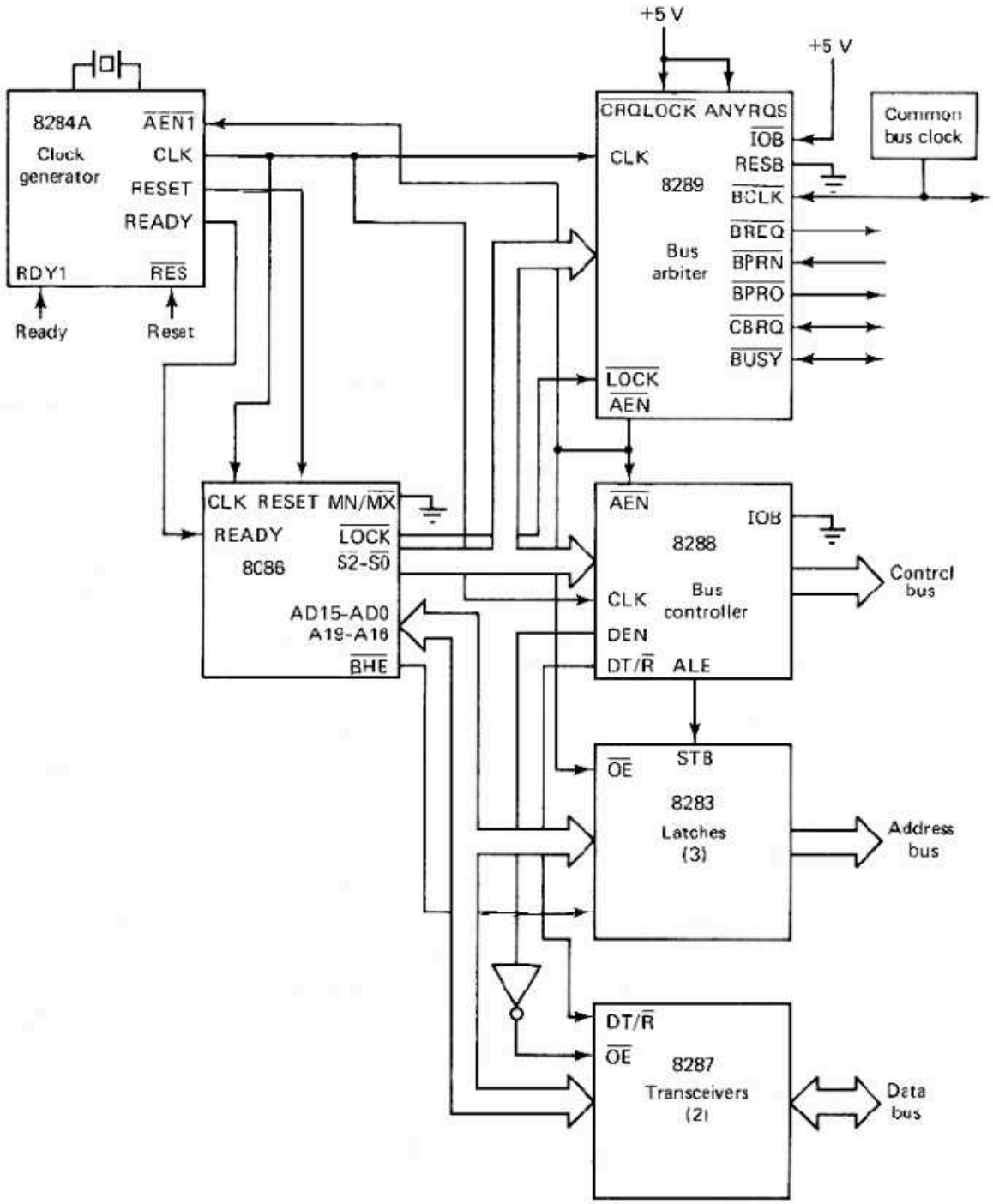
- **BPRO** (Bus Priority Out)
  It is an active low out output signal used specially in Serial Priority Resolving Technique.
  It is used to pass on the Bus Control to the BPRN of the next device.

- **BUSY** (Bus Busy)
  This is an active low, bi-directional signal used to indicate to all Bus Arbiters that the system bus is currently being used by some other device (i.e. it is busy).

- **CBRQ** (Common Bus Request)
  It is an input signal used to inform the arbiter that another arbiter is requesting for the Multi Master system bus. The bus arbiter currently running cannot put the CBRO line low.

ROLE OF 8289 IN A LOOSELY COUPLED SYSTEM

Note: 8283 and 8282 are similar latches, 8286 and 8287 are similar transreceivers



8289 monitors $S_2S_1S_0$ from the **μP** to find out when 8086 would require the system bus.
Once 8086 initiates a bus cycle, 8289 gives a bus request by making BREQ low.
Now 8289 may or may not get the system bus.

Case 1 : 8289 does not get the system bus.

- Now **8289 makes its AEN o/p = 1**.
- This makes **AEN** i/p **of 8288 = 1**.
  Hence all control signals of **8288** are **disabled**.
  i.e. the control bus is disabled.
  The $\overline{\textbf{OE}}$ of the latch = 1 ∴ the Latch is disabled.
  Since DEN o/p of 8288 is inactive, the $\overline{\textbf{OE}}$ of Transreceiver = 1 ∴ transreceiver is disabled.
- Hence **Address, Data and Control buses are disabled.**
- Moreover, **AEN1 i/p of 8284 = 1** hence its Ready o/p becomes = 0.
  ∴ Ready i/p of 8086 = 0.
  This makes 8086 enter WAIT state.

Case 2 : Now 8289 gets the system bus.

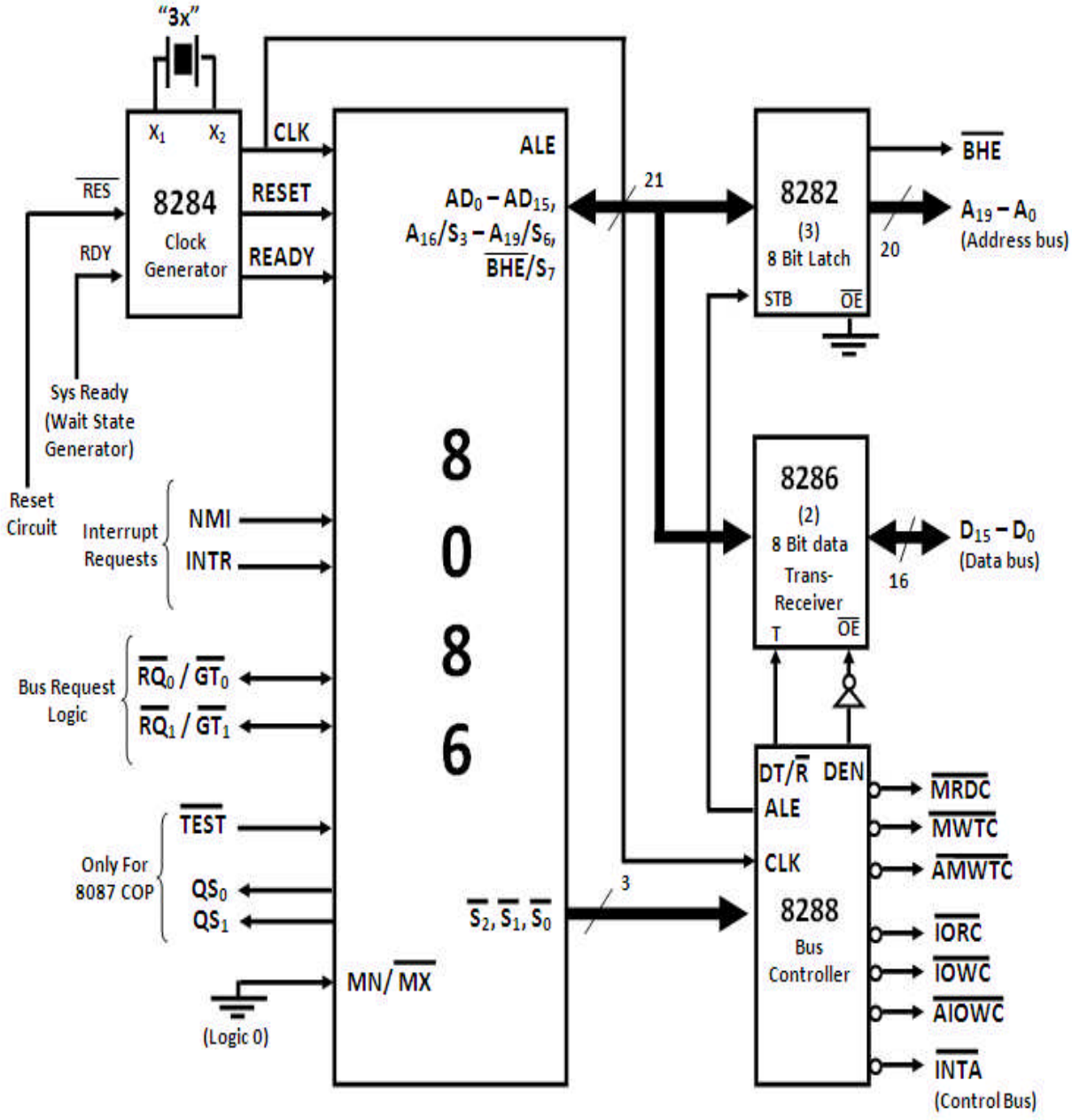
- Now **8289 makes its AEN o/p = 0.**
- This makes **AEN** i/p **of 8288 = 0.**
  Hence all control signals of **8288** are **enabled**.
  i.e. the control bus is enabled.
  The $\overline{\textbf{OE}}$ of the latch = 0 ∴ the Latch is enabled.
  The $\overline{\textbf{OE}}$ of Transreceiver = 0 ∴ transreceiver is enabled.
- **Hence Address, Data and Control buses are all enabled.**
- Moreover, **AEN1 i/p of 8284 = 0** hence its Ready o/p becomes = 1.
  ∴ Ready i/p of 8086 = 1. **For doubts contact Bharat Sir on 98204 08217**
  Thus 8086 comes out of WAIT state.
- The $\overline{\textbf{LOCK}}$ **from 8086** is given **to 8289**.
- This signal is **active** when 8086 is **executing** a $\overline{\textbf{LOCK}}$ **instruction**.
- It **prevents 8289** from **granting** the **system bus** to another bus master **during** the $\overline{\textbf{LOCK}}$ **instruction**. #Please refer Bharat Sir's Lecture Notes for this ...

This is how 8086 uses 8289 to access the system bus in a Loosely Coupled system.

8086 DESIGNING

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

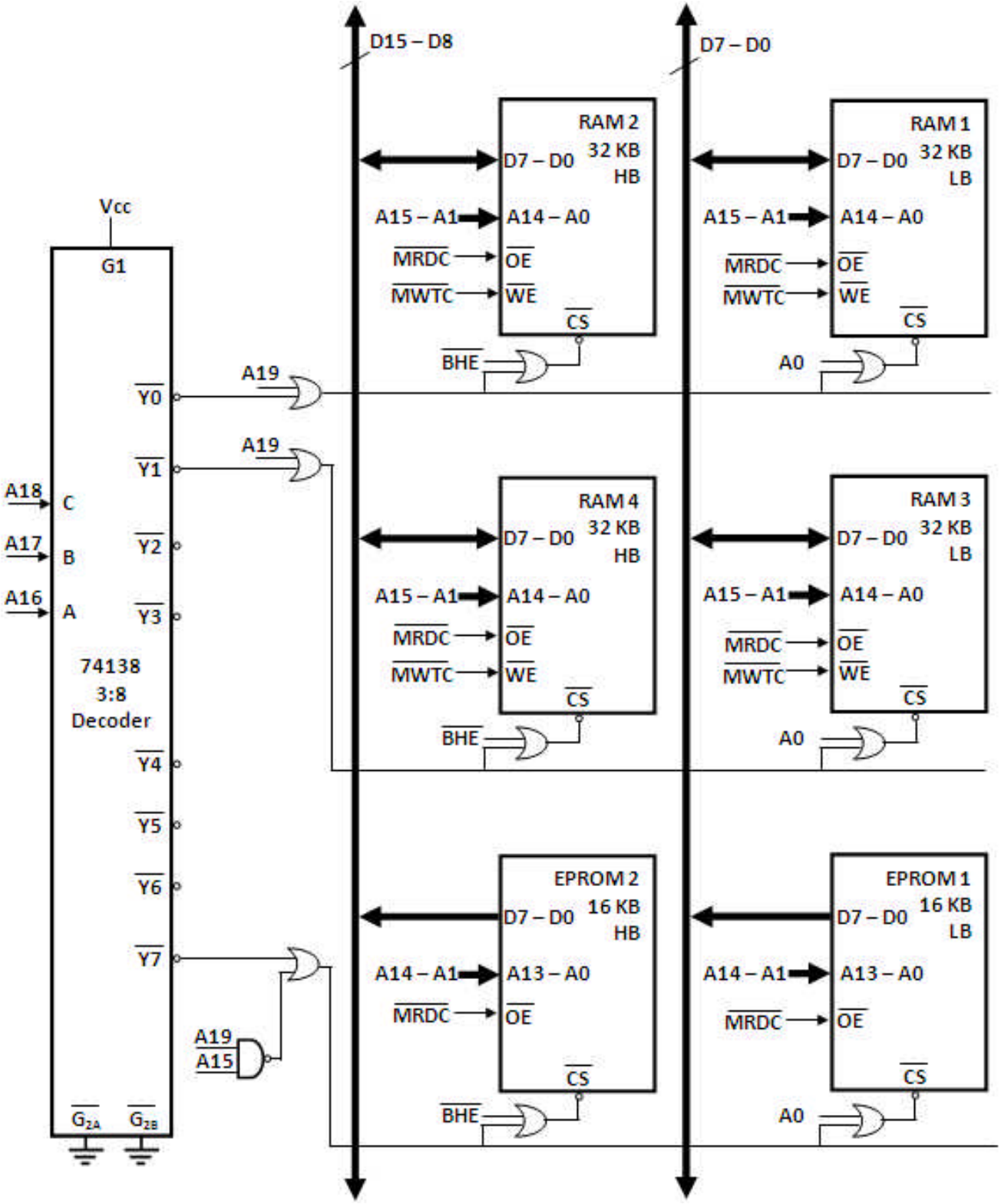
Size of a single RAM chip = 32 KB
$= 32 \times 1 KB = 2^5 \times 2^{10}$
$= 2^{15}$
= 15 address lines

= **(A15 … A1)**

For doubts contact Bharat Sir at 98204 08217

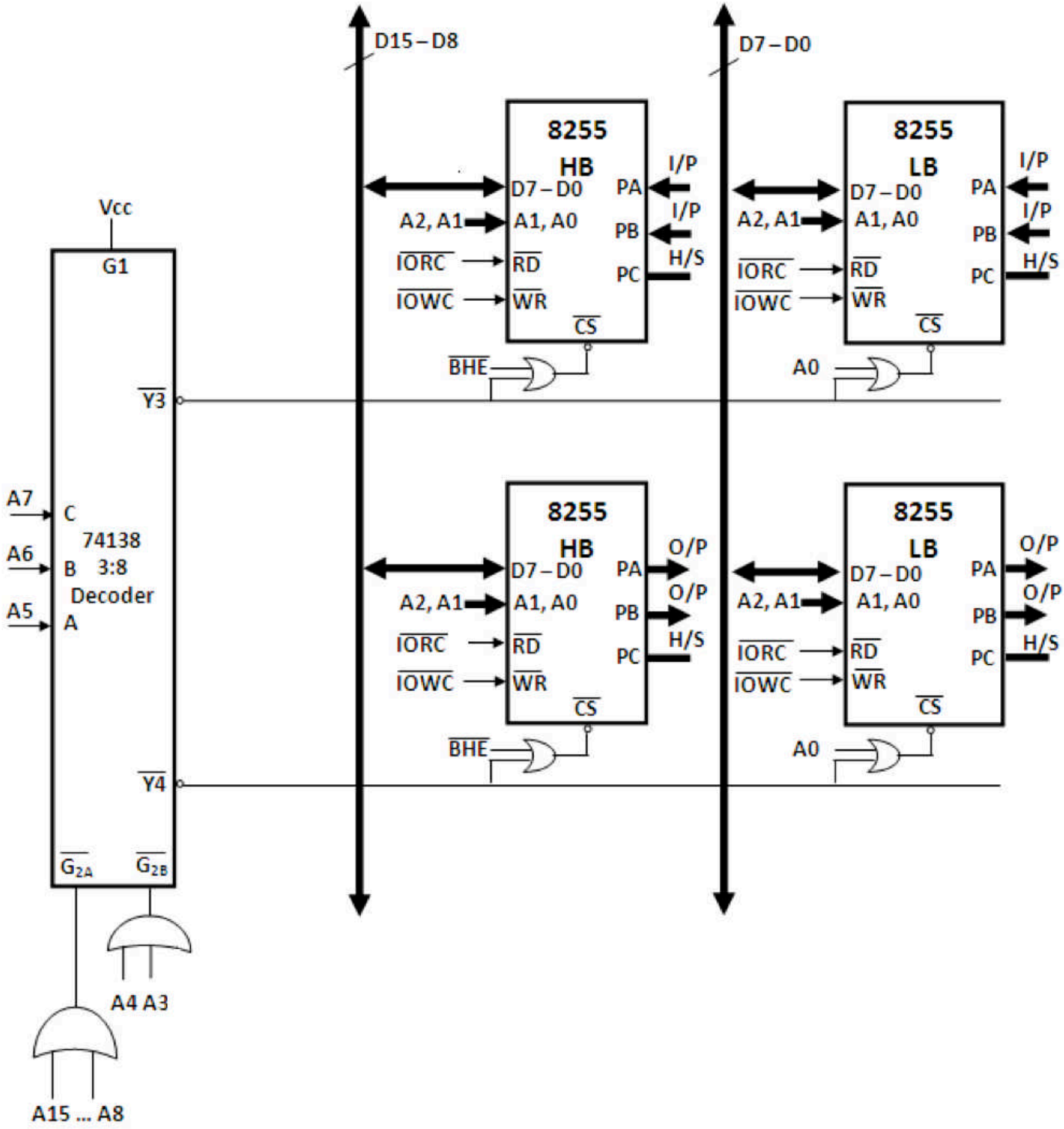
MEMORY MAP

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

Q2) Design an 8086 based Minimum Mode system working at 6 MHz having the following:
128KB EPROM using 32KB chips,
128KB RAM using 64KB chips,

Soln: EPORM:

Required = _____ KB

Available = _____ KB

No. of chips = _____ chips.

Starting address of EPROM is calculated as:

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

F F F F F H
- _____ H
_____ H

Size of a single EPROM chip = _____KB; = ___ x 1KB;

= 2—— x 2^{10} ; = 2—— ;

= _____ address lines

= **(A_____ … A1)**

RAM:

Required = _____ KB

Available = _____ KB

No. of chips = _____ chips.

Size of a single RAM chip = _____KB; = ___ x 1KB;

= 2—— x 2^{10} ; = 2—— ;

= _____ address lines

= **(A_____ … A1)**

For doubts contact Bharat Sir at 98204 08217

MEMORY MAP

MEMORY CHIP	Address Bus																			MEM ADDRESS
	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	A0	
RAM 1 Begin	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	00000H
RAM 1 End																					
RAM 2 Begin																					
RAM 2 End																					
EPROM 1 Begin																					
EPROM 1 End																					
EPROM 2 Begin																					
EPROM 2 End																					
EPROM 3 Begin																					
EPROM 3 End																					
EPROM 4 Begin																					
EPROM 4 End	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	FFFFFH

MEMORY DIAGRAM

Q3) Design an 8086 – 8087 based Maximum Mode system working at 6 MHz having the following:
64 KB EPROM using 16 KB chips,
256 KB RAM using 64 KB chips,

Soln: EPORM:

Required = _____ KB

Available = _____ KB

No. of chips = _____ chips.

Starting address of EPROM is calculated as:

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

F F F F F H
- _____ H
_____ H

Size of a single EPROM chip = _____KB; = ___ x 1KB;

= 2—— x 2^{10} ; = 2—— ;

= _____ address lines

= **(A_____ … A1)**

RAM:

Required = _____ KB

Available = _____ KB

No. of chips = _____ chips.

Size of a single RAM chip = _____KB; = ___ x 1KB;

= 2—— x 2^{10} ; = 2—— ;

= _____ address lines

= **(A_____ … A1)**

For doubts contact Bharat Sir at 98204 08217

MEMORY MAP

MEMORY	Address Bus																				MEM
CHIP	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	A0	ADDRESS
RAM 1 Begin	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	00000H
RAM 1 End																					
RAM 2 Begin																					
RAM 2 End																					
RAM 3 Begin																					
RAM 3 End																					
RAM 4 Begin																					
RAM 4 End																					
EPROM 1 Begin																					
EPROM 1 End																					
EPROM 2 Begin																					
EPROM 2 End																					
EPROM 3 Begin																					
EPROM 3 End																					
EPROM 4 Begin																					
EPROM 4 End	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	FFFFFH

MEMORY DIAGRAM

Q4) Design an 8086 based Minimum Mode system working at 10 MHz having the following:
16 KB EPROM using 4 KB chips,
64 KB RAM using 8 KB chips,

Soln: Soln: EPORM:

Required = _____ KB

Available = _____ KB

No. of chips = _____ chips.

Starting address of EPROM is calculated as:

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

F F F F F H
- _____ H
_____ H

Size of a single EPROM chip = _____KB; = ___ x 1KB;

= 2—— x 2^{10} ; = 2—— ;

= _____ address lines

= **(A_____ … A1)**

RAM:

Required = _____ KB

Available = _____ KB

No. of chips = _____ chips.

Size of a single RAM chip = _____KB; = ___ x 1KB;

= 2—— x 2^{10} ; = 2—— ;

= _____ address lines

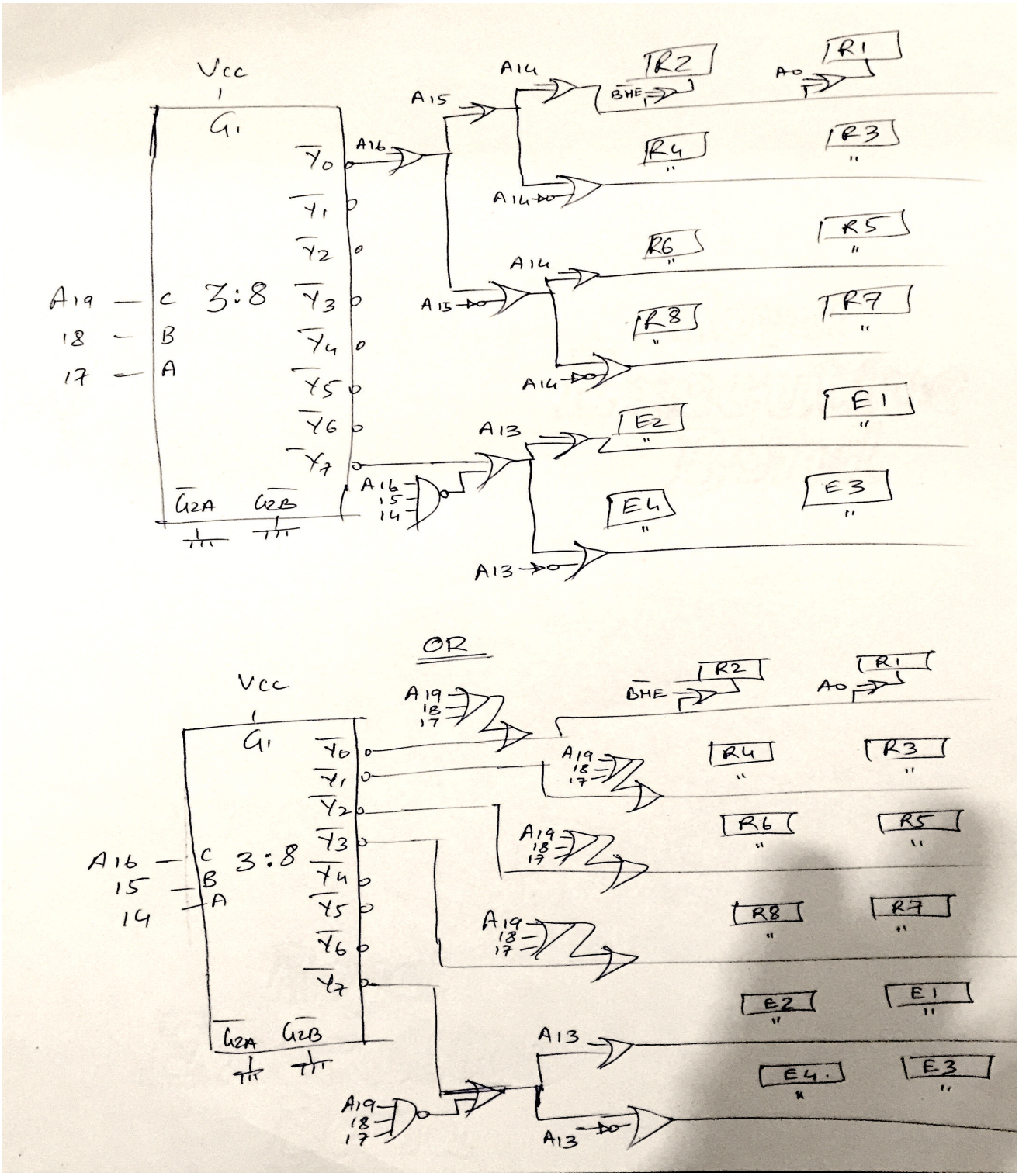
= **(A_____ … A1)**

For doubts contact Bharat Sir at 98204 08217

MEMORY MAP

MEMORY CHIP	Address Bus																				MEM ADDRESS
	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	A0	
RAM 1 Begin	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	00000H
RAM 1 End	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	03FFEH
RAM 2 Begin	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	00001H
RAM 2 End	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	03FFFH
RAM 3 Begin	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	04000H
RAM 3 End	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	07FFEH
RAM 4 Begin	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	04001H
RAM 4 End	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	07FFFH
RAM 5 Begin	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	08000H
RAM 5 End	0	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0BFFEH
RAM 6 Begin	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	08001H
RAM 6 End	0	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0BFFFH
RAM 7 Begin	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0C000H
RAM 7 End	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0FFFEH
RAM 8 Begin	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0C001H
RAM 8 End	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0FFFFH
EPROM 1 Begin	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	FC000H
EPROM 1 End	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	FDFFEH
EPROM 2 Begin	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	FC001H
EPROM 2 End	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	FDFFFH
EPROM 3 Begin	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	FE000H
EPROM 3 End	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	FFFFEH
EPROM 4 Begin	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	FE001H
EPROM 4 End	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	FFFFFH

2 EQUALLY CORRECT DECODING SOLUTIONS



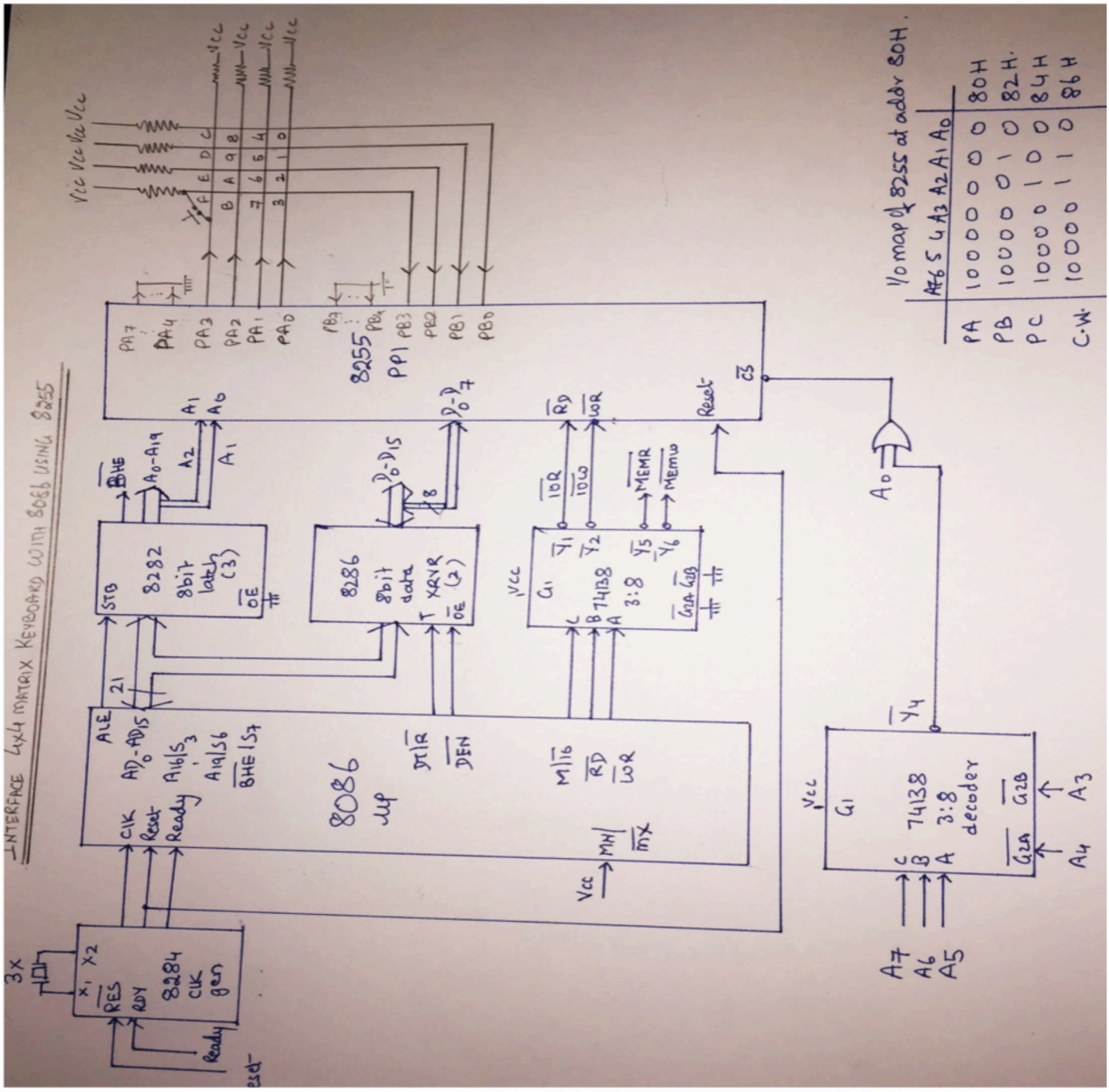
KEYBOARD & DISPLAY INTERFACING

ETRX ONLY

4 x 4 MATRIX KEYBOARD

Interface a 4 x 4 Matrix Keyboard to 8086 using 8255

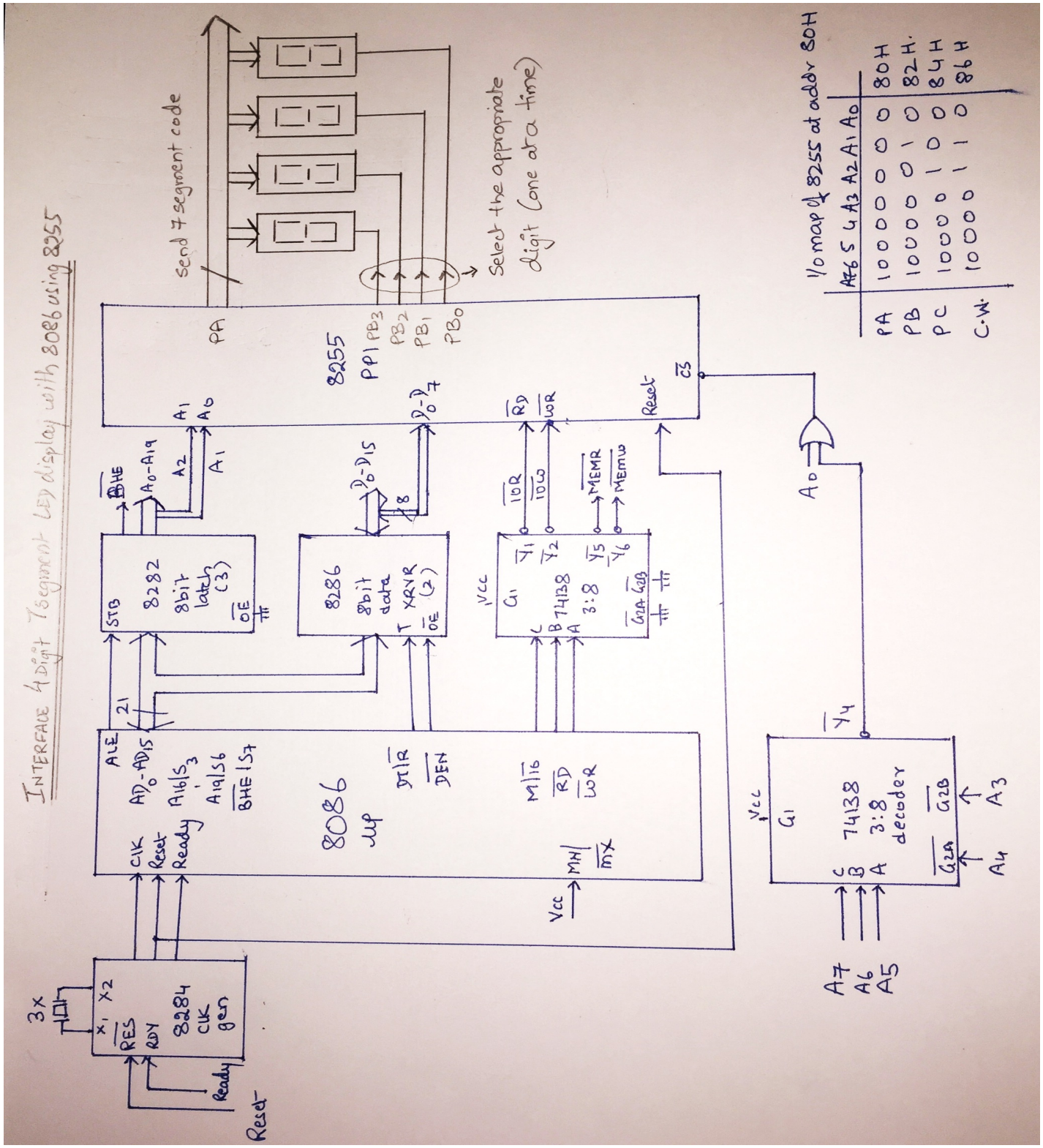
1) A Matrix Keyboard is formed by a combination of **rows and columns**.
2) The advantage of a matrix keyboard is that we can interface **more keys** using **less lines**.
3) A 4x4 Matrix Keyboard uses **4 lines as rows and 4 as columns**.
4) This provides 16 intersecting points to connect **16 keys**.
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
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
8282 8bit latch (3)
8286 8bit data XCVR (2)
74138 3:8 decoder
8284 CLK gen
8086 up
A7 A6 A5
A4 A3

7 SEGMENT LED DISPLAY

Interface a 4 digit 7 segment LED display to 8086 using 8255

1) 7 segment LEDs are ideal for displaying **Numeric information**.
2) Numbers can be displayed by sending **appropriate 7 segment codes**.
3) These 7 segment codes are **obtained from Look up tables** stored in the memory.
4) Typical instruction used to obtain 7 segment code is **XLAT**.
5) This is a 4 digit **multiplexed** seven segment display.
6) To display a 4 digit number, we use the **principle of persistence of vision**.
7) We send the 7 segment code of the **first digit and activate the first digit only**.
8) We do the same for the **remaining three digits** and repeat the entire process in a **loop**.
9) The idea is, any image persists on the human eye for **about 100 milliseconds**.
10) So we must **complete our cycle of 4 digits** within this much time.
11) Hence to the onlooker, it **appears** that all 4 digits are **displayed simultaneously**, whereas in **reality**, they are being **displayed one by one**.
12) As microprocessors performs operations in **microseconds**, this time window of about 100 milliseconds is **easily achieved** by the microprocessor.
13) **Port A** is used to **send the seven segment codes** of the 4 digits one by one.
14) **Port B** is used to select the **appropriate digit** one by one.

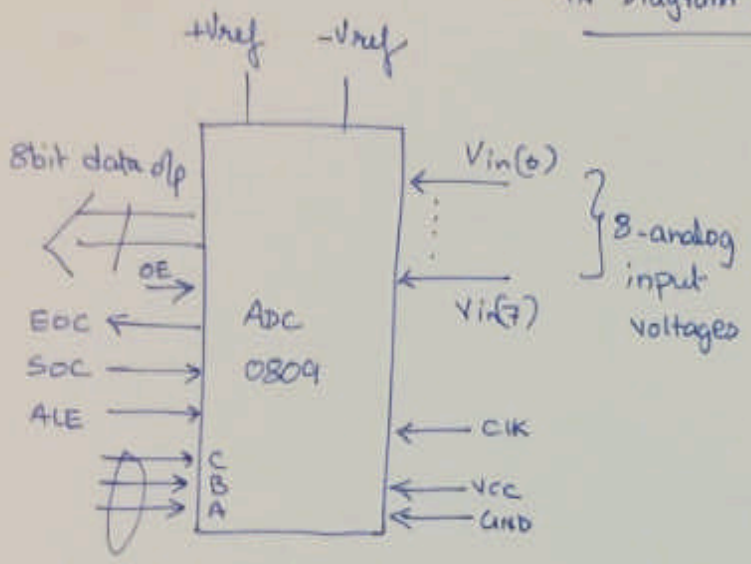

INTERFACE 4 digit 7 segment LED display with 8086 using 8255
send 7 segment code
Select the appropriate digit (one at a time)
I/O map of 8255 at addr 80H
A7 A6 A5 A4 A3 A2 A1 A0
PA 1 0 0 0 0 0 0 0 80H
PB 1 0 0 0 0 0 1 0 82H
PC 1 0 0 0 0 1 0 0 84H
C.W. 1 0 0 0 0 1 1 0 86H
8255
PA
PB3 PB2 PB1 PB0
PPI
8086 up
8282 8bit latch (3)
8286 8bit data XCVR (2)
74138 3:8 decoder
8284 CLK gen
3x
X1 X2
RES RDY
Ready Reset
CLK Reset Ready
ALE
AD0-AD15
A16/S3 A17/S4 A18/S5 BHE/S7
M/IO DT/R DEN RD WR
MN/MX
STB OE
A0-A19 BHE
A2 A1 A0
D0-D15
D0-D7
IOR IOW MEMR MEMW
RD WR
Reset
CS
Vcc
G1 G2A G2B
A7 A6 A5 A4 A3

ADC & DAC INTERFACING

EXTC ONLY

ADC 0809

Pin Diagram of ADC 0809



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.

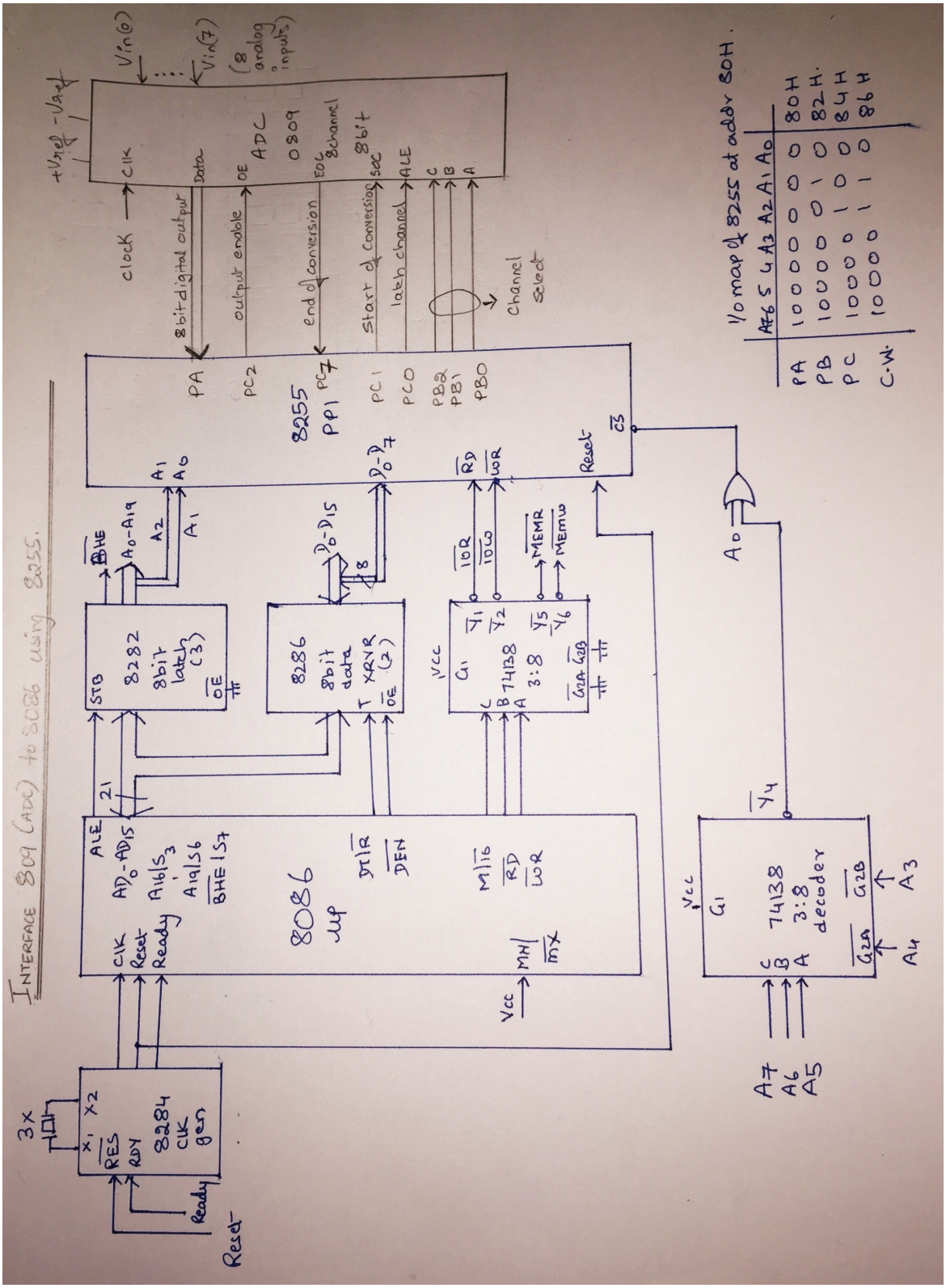
C,B,A: 3bits to select input channel out of 8 options.

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.

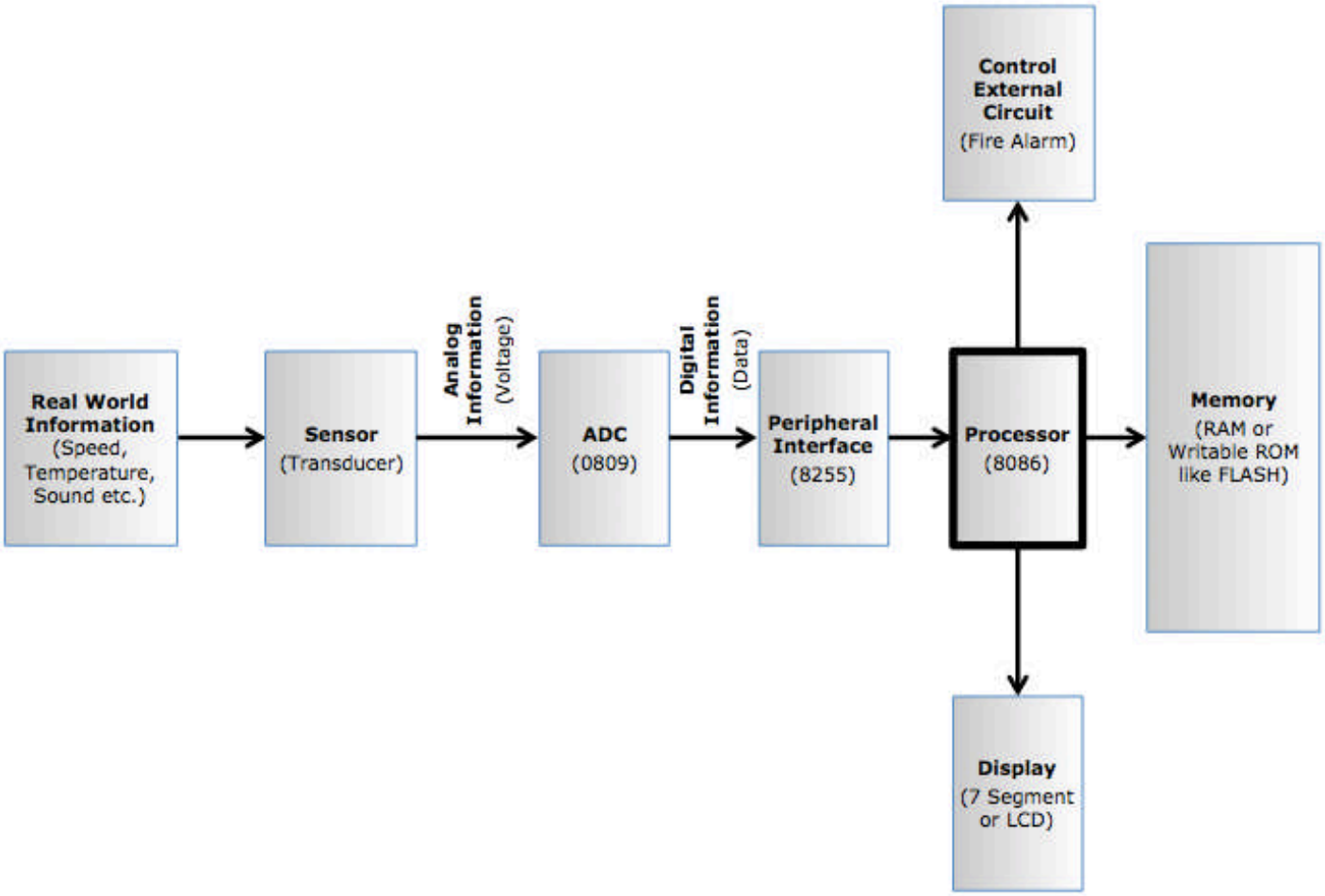

INTERFACE 809 (ADC) to 8086 using 8255.
I/O map of 8255 at addr 80H.
A7 A6 A5 A4 A3 A2 A1 A0
PA 1 0 0 0 0 0 0 0 80H
PB 1 0 0 0 0 0 1 0 82H
PC 1 0 0 0 0 1 0 0 84H
C.W. 1 0 0 0 0 1 1 0 86H
8255 PPI
8086 up
8284 CLK gen
8282 8bit latch (3)
8286 8bit data T XRVR (2)
74138 3:8 decoder
ADC 0809
8bit digital output
output enable
end of conversion
Start of Conversion
latch channel
Channel select
clock
(8 analog inputs)
Reset

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 ;-))

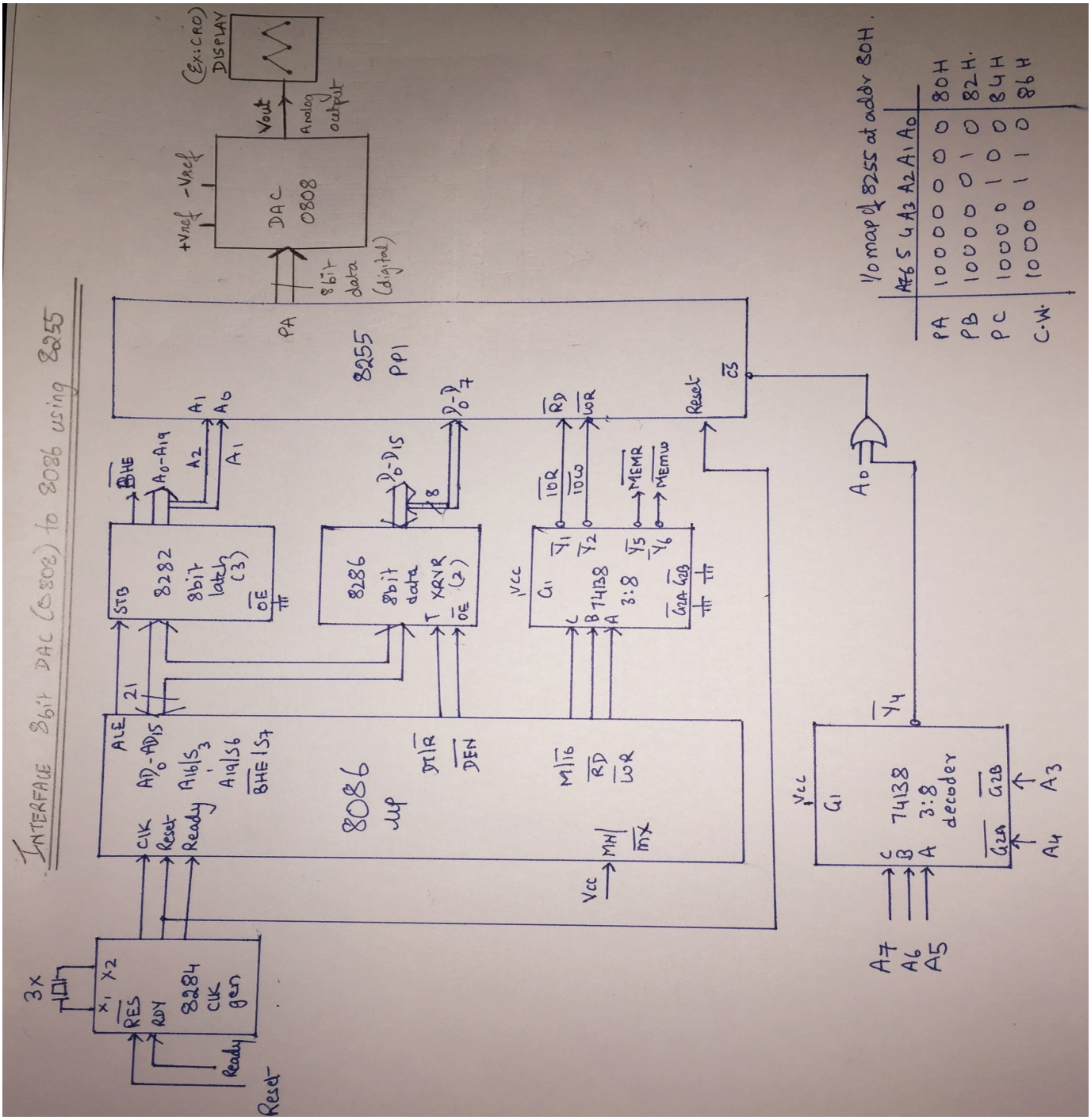

Control External Circuit (Fire Alarm)
Real World Information (Speed, Temperature, Sound etc.)
Sensor (Transducer)
Analog Information (Voltage)
ADC (0809)
Digital Information (Data)
Peripheral Interface (8255)
Processor (8086)
Memory (RAM or Writable ROM like FLASH)
Display (7 Segment or LCD)

DAC 0808

Interface DAC 0808 to 8086 using 8255

1) DAC is an **8 bit Digital to Analog Converter**.
2) It can **convert** an 8 bit digital **data input** into an analog **voltage** output.
3) **Reference voltage** for conversion is provided using **+Vref and –Vref.**
4) The **output can be amplified** (optional) using an op-amp **LF 351**.
5) DACs are used in various applications such as **Waveform generation**, **PWM**, Motor control Applications, DSP etc.
6) Here we connect the output to a display device like a **CRO**.
7) By simple programming we can generate several types of wave forms like Ramp, Saw-tooth, **Triangular waveform** etc.

Note: These programs are already discussed in the earlier chapter on 8255, in the same book.


Interface 8bit DAC (0808) to 8086 using 8255
(Ex: CRO) DISPLAY
Vout
Analog output
+Vref -Vref
DAC 0808
8 bit data (digital)
PA
8255 PPI
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
A1 A0
A2
A1
A0-A19
BHE
STB
8282 8bit latch (3)
OE
D0-D15
8286 8bit data T XRVR OE (2)
D0-D7
RD WR
IOR IOW
MEMR MEMW
Reset
CS
G1 Y1 Y2 Y5 Y6
B 74138 3:8
A
G2A G2B
ALE
21
AD0-AD15
A16/S3 A17/S4 A18/S5 BHE/S7
CLK Reset Ready
8086 up
DT/R DEN
M/IO RD WR
MN/MX
Vcc
3x
X1 X2
RES RDY
8284 CLK gen
Ready
Reset
A0
Y4
G1
74138 3:8 decoder
C B A
G2A G2B
A7 A6 A5
A4 A3

ADVANCED MICROPROCESSORS

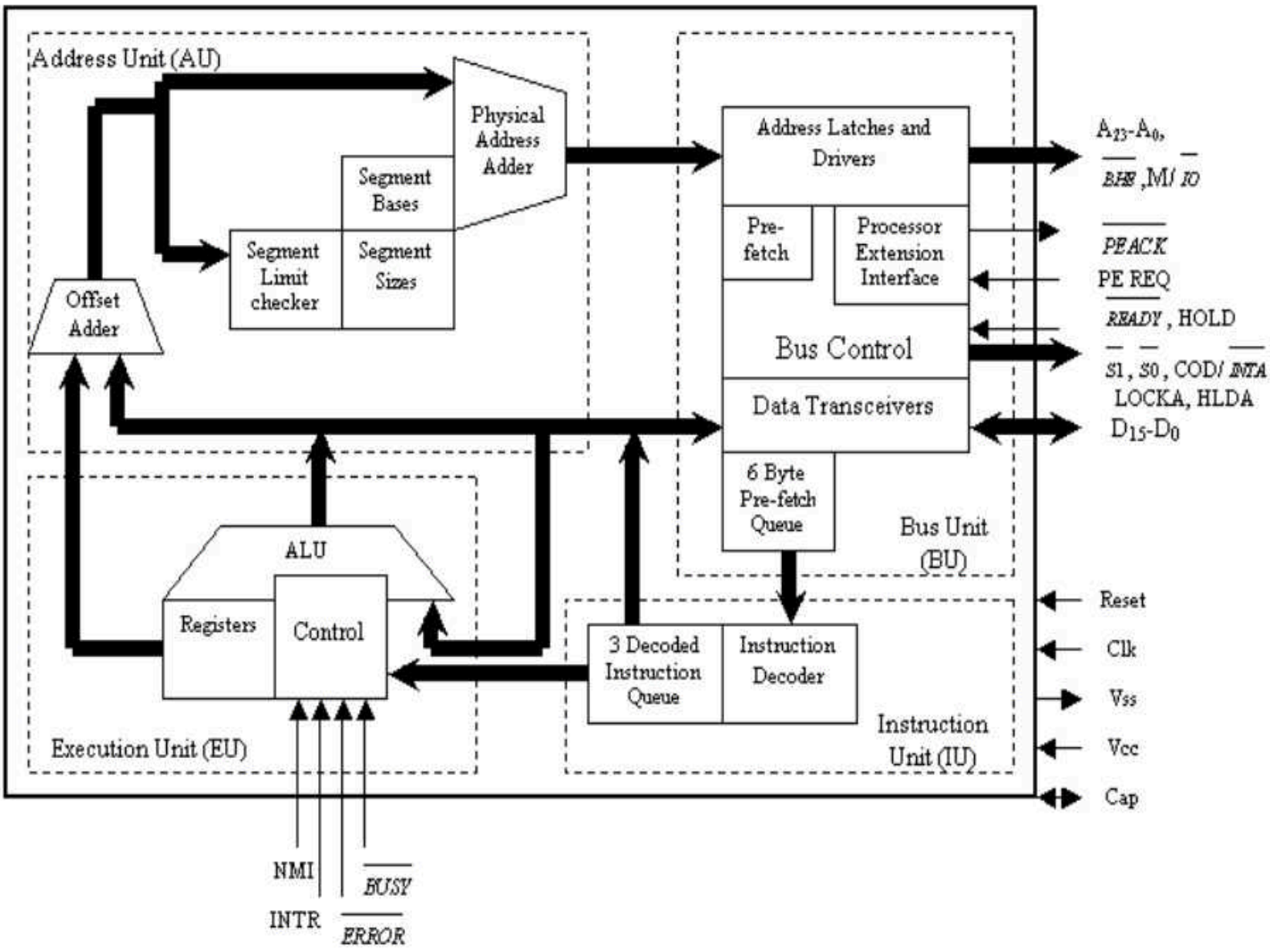
EXTC ONLY

80286

Salient Features of 80286

1) It is a **16 bit Microprocessor.**
2) It has a **16 bit data bus.**
3) It has **2 memory banks.**
4) It has a **24 bit address bus.**
5) It can access **16 MB memory.**
6) It has **3 Pipeline stages.**
7) The Pipeline stages are called: **Fetch, Decode, Execute.**
8) It operates on **6 MHz – 12 MHz** frequency.
9) It has **1,34,000 transistors.**
10) It was released in the year **1982**.

80286 Architecture



The 80286 architecture is divided into 4 functional units.

Bus Unit (BU)

1) It is used to perform **all external data transfers** between the microprocessor and external memory and I/O.
2) It prefetches instructions into the **6 byte instruction prefetech queue**.
3) It handles Bus requests and grants between **microprocessor and co processors** like 80287.
4) It has internal a**ddress latches and internal data transreceivers**.

Instruction Unit (IU)

1) It **decodes instructions** and places them into the decoded queue.
2) **Three instructions** can be stored into the decoded queue.
3) These instructions are then passed on to the execution unit for execution.
4) Thus a **three stage pipeline** is achieved: Fetch, Decode, execute.

Execution Unit (EU)

1) It is the main unit where all **instructions are executed**.
2) It has a **16-bit ALU** used for performing 16-bit arithmetic and logic operations.
3) It uses the **GPRs** (General Purpose Registers) to perform operations and updates the status into the **Flag Register**.
4) If operands are required from the memory, then their **physical address** needs to be calculated by the Address Unit.

Address Unit (AU)

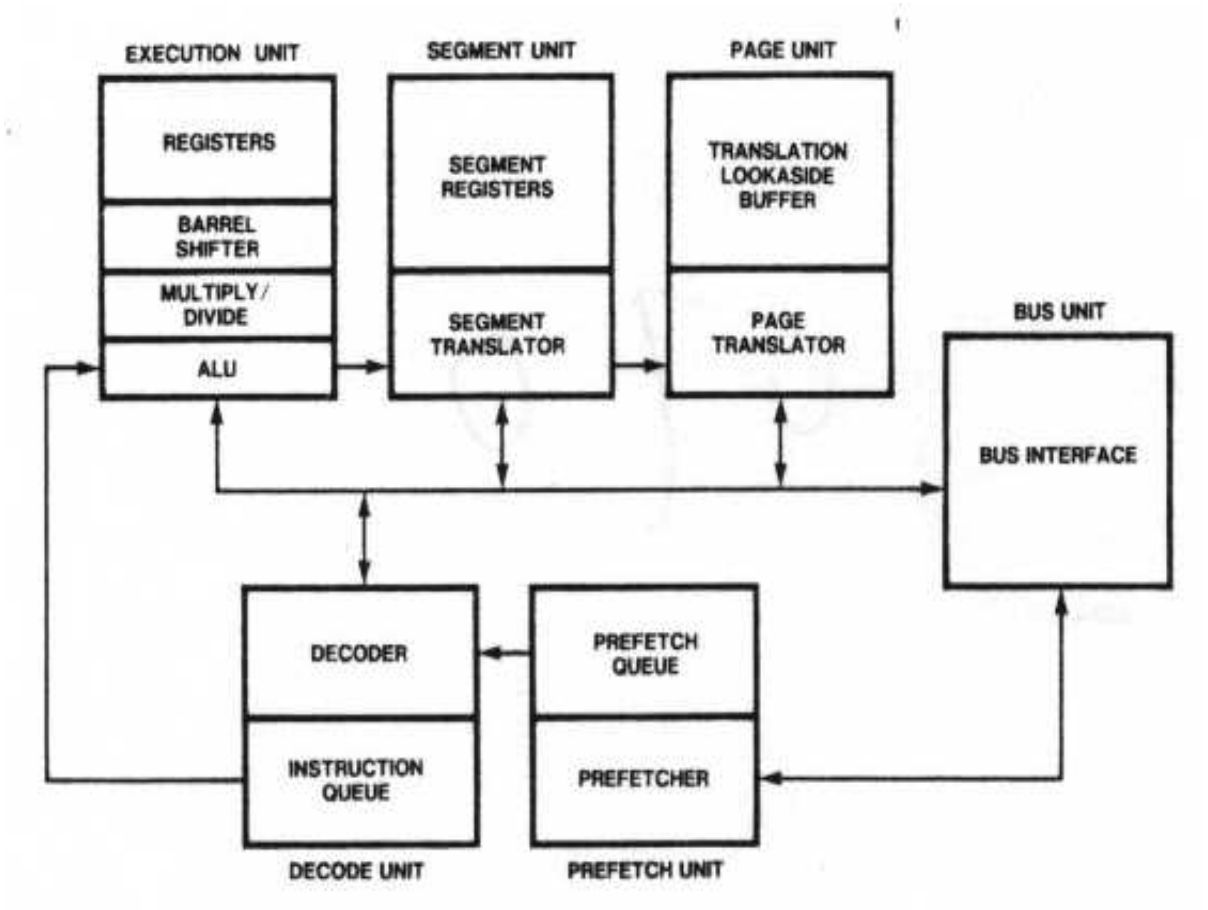
1) Used to **perform address translation** to calculate Physical address of the operands.
2) In **real mode,** physical address is calculated **like 8086** using the formula: Segment address x 10H + offset address.
3) In **Protected mode** physical address is calculated using **segment descriptor tables**.
4) The segment descriptor gives the **base address** to which it **adds the offset address** and hence calculates the physical address.

80386

Salient Features of 80386

1) It is a **32 bit Microprocessor**.
2) It has a **32 bit data bus**.
3) It has **4 memory banks**.
4) It has a **32 bit address bus**.
5) It can access **4 GB memory**.
6) It has **3 Pipeline stages**.
7) The Pipeline stages are called: **Fetch, Decode, Execute**.
8) It operates on **16 MHz – 33 MHz** frequency.
9) It has **2,75,000 transistors**.
10) It was released in the year **1985**.

80386 Architecture



80386 architecture is divided into 5 independent units.

Bus Unit (Bus Interface Unit)

1) **The Bus unit is responsible for transferring data in and out of the µP.**
2) It is connected to the external memory and I/O devices, using the system bus.
3) It gets requests from Prefetch unit for fetching instructions and from execution unit for transferring data.
4) If both requests occur simultaneously preference is given to execution unit.

Prefetch Unit

1) The Pre-fetch unit **fetches further instructions in advance to implement pipelining.**
2) It **fetches the next 16 bytes of the program** and stores it into the **Prefetch Queue**.
3) It **refills the queue** when at least **4 bytes** are empty as 80386 has a 32 bit data bus.
4) During a **branch**, the instructions in the queue are **invalid** and hence are **discarded**.

Decode Unit

1) 80386 µP has a **separate unit for decoding instructions** called the Decode Unit.
2) It **decodes the next three instructions and keeps them ready** in the Decode Queue.
3) The decoded instructions are stored in **Micro-Coded** form.
4) During a **branch**, the instructions in the queue are **invalid** and hence are **discarded**.

Execution Unit

1) Execution Unit performs the main task of **executing instructions**.
2) Normally, execution requires Arithmetic or Logic operations performed by a **32-bit ALU**.
3) It also has dedicated circuits for **32-bit multiplication and division**.
4) A **64-bit barrel shifter** is also provided for faster shifts during multiplication and division.
5) **Operands** for the ALU can either be provided in the **instruction**, or can be taken **from memory** or could be taken from the **32-bit registers** like EAX, EBX etc.
6) Additionally there is a **32-bit Flag register** (EFLAGS) giving the **Status** of the current result.

Memory Unit

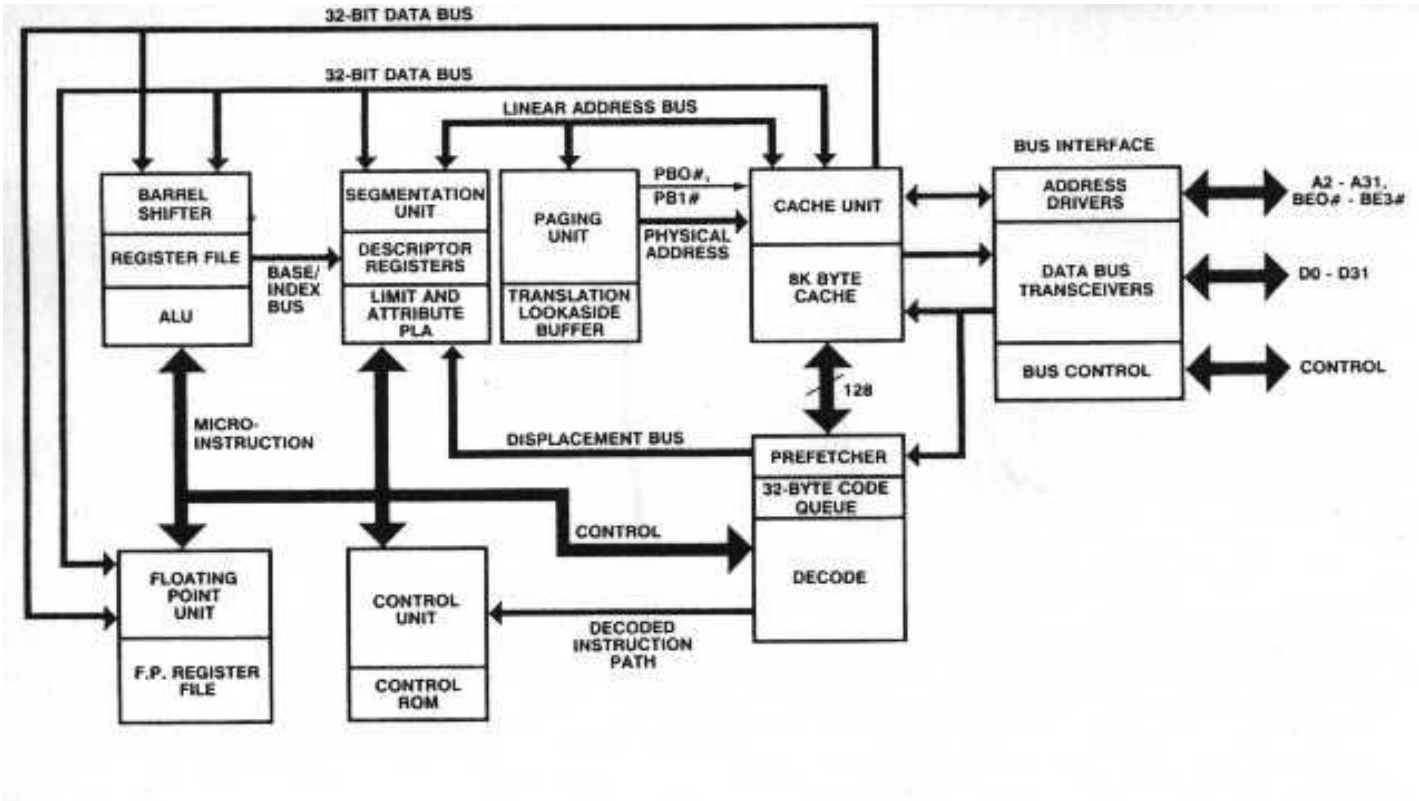
1) The Memory unit converts Virtual Address (Logical address) to Physical Address.
2) 80386 µP implements **64 Terra bytes of Virtual memory using Segmentation and Paging**. Hence the Memory Unit is sub-divided into Segmentation Unit and Paging Unit.
3) **Segmentation is compulsory, while Paging is optional.**
4) **The Segmentation Unit converts the Logical Address into a Linear Address.**
5) **The Paging Unit converts the Linear Address into a Physical Address.**
6) If Paging is not used, then the Linear Address itself is the Physical Address.

80486

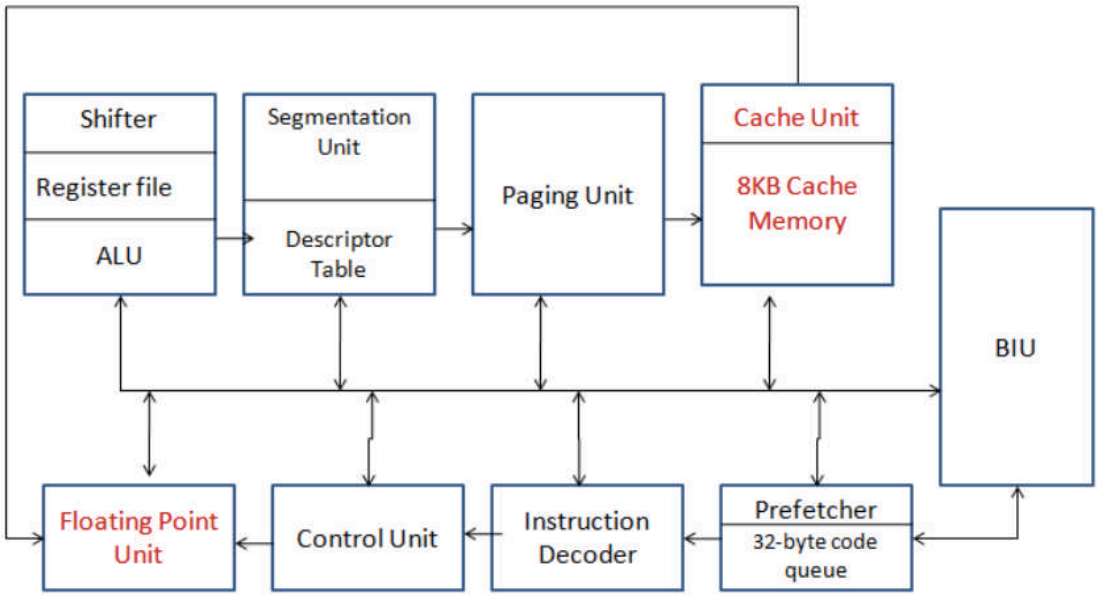
Salient Features of 80486

1) It is a **32 bit Microprocessor**.
2) It has a **32 bit data bus**.
3) It has **4 memory banks**.
4) It has a **32 bit address bus**.
5) It can access **4 GB memory**.
6) It has **5 Pipeline stages**.
7) The Pipeline stages are called: **Prefetch, Decode, Address Gen, Execute, Write Back**.
8) It operates on **25 - 60 MHz** frequency.
9) It has **11,80,235 transistors**.
10) It was released in the year **1989**.
11) It has an internal **Floating Point Unit**.
12) It has **unified 8 KB L1 Cache** (Code and data combined).

80486 Architecture



Simplified Diagram:



80486 Architecture is divided into several independent functional units

Bus Unit (Bus Interface Unit)

1) **The Bus unit is responsible for transferring data in and out of the µP.**
2) It is connected to the external memory and I/O devices, using the system bus.
3) It gets requests from Prefetch unit for fetching instructions and from execution unit for transferring data.

Prefetch Unit

1) The Pre-fetch unit **fetches further instructions in advance to implement pipelining.**
2) It **fetches the next 32 bytes of the program** and stores it into the **Prefetch Queue**.

Decode Unit

1) 80486 µP has a **separate unit for decoding instructions** called the Decode Unit.
2) The decoded instructions are passed on to the Control Unit which **generates Control signals** for its execution.

Execution Unit

1) Execution Unit performs the main task of **executing instructions**.
2) It comprises of an Integer Execution Unit and a Floating Point Exec unit.
3) Integer Unit performs Inter arithmetic and logic operations.
4) The on chip Floating Point Unit performs Floating Point operations thereby eliminating the need of a Math Co Processor.

Memory Unit

1) The Memory unit converts **Virtual Address (Logical address) to Physical Address**.
2) It has two sub sections.
3) The **Segmentation Unit** performs Segment Translation and converts Virtual Address into Linear Address.
4) The **Paging Unit** performs Page translation and converts Linear Addres into Physical Address.

Cache Unit

1) 80486 has onchip **8 KB L1 Cache**.
2) It is a **unified cache** containg both Code and Data.
3) Most recently used 8K Bytes of Code and Data are stored in the onchip L1 Cache.
4) This makes **futher reference to the same data much faster** and tremendously increases the performance of the microprocessor.

80586 - PENTIUM

Salient Features of 80586 – Pentium 1

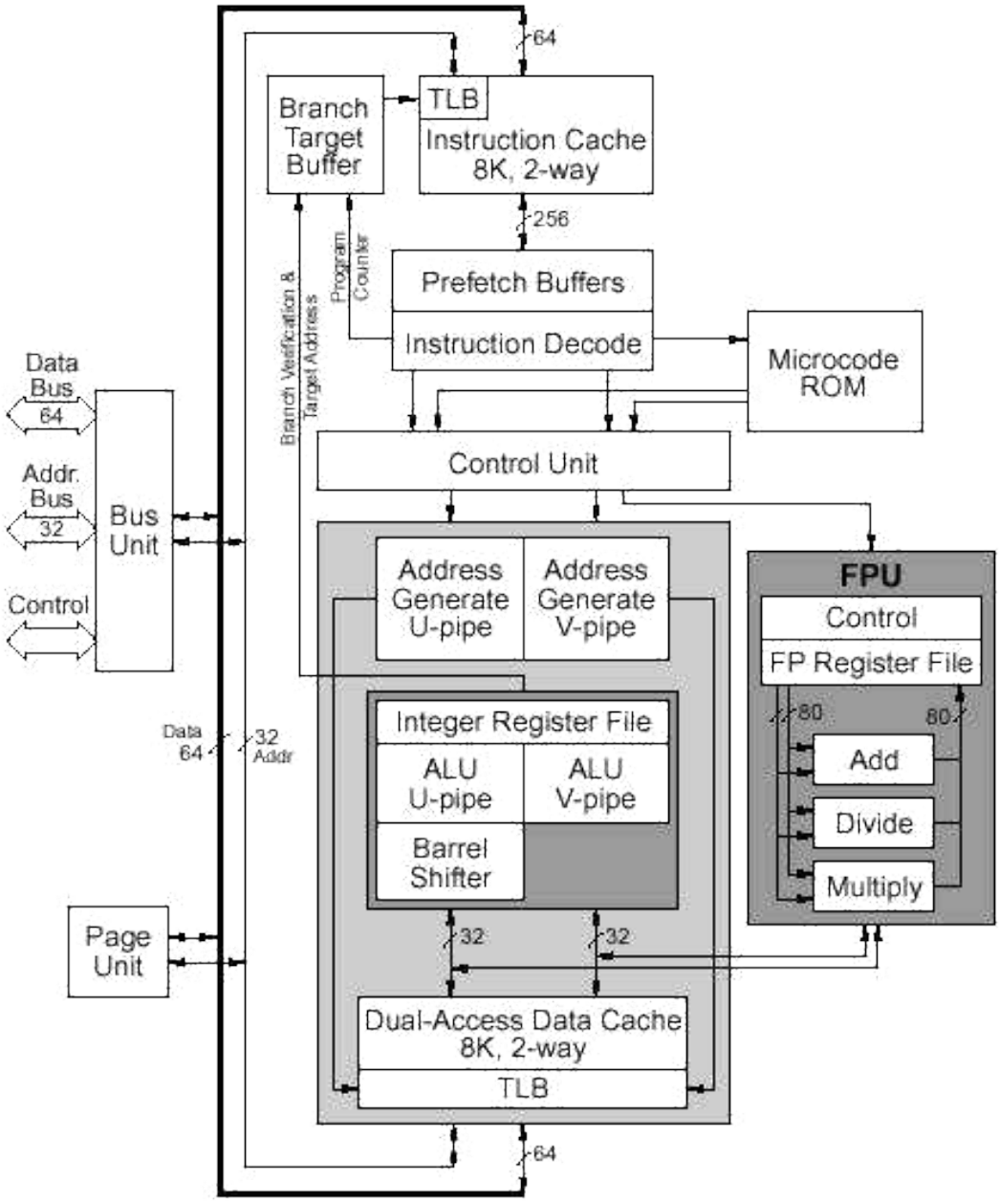
1) It is a **32 bit Microprocessor**.
2) It has a **64 bit data bus**.
3) It has **8 memory banks**.
4) It has a **32 bit address bus**.
5) It can access **4 GB of physical memory**.
6) It has **5 Pipeline stages for integer operations**.
7) It has an **internal Floating point unit**.
8) It has an **8 stage Floating point Pipeline**.
9) It is **2 way superscalar**. This means it has two pipes called the u-pipe and the v-pipe.
10) It operates on **66 MHz – 99 MHz** frequency.
11) The **Integer Pipeline** stages are called:

    PF – Prefetch; D1 – Decode; D2 – Address Translation;

    EX – Execute; WR – Write Back.

12) It has **31,00,000 transistors**.
13) It was released in the year **1993**.
14) It has a protection mechanism with **4 privilege levels**.
15) It has **on-chip L1 Code cache and L1 Data Cache both 8 KB each**.
16) It has a **branch prediction logic** with a **256 entry** Branch Target Buffer (BTB).

80586 Architecture



Pentium has a 2 way superscalar architecture giving extremely superior performance.
It has two pipes called the u–pipe and the v-pipe. Each performs a 5-stage integer pipeline.

Bus Unit

1) **The Bus unit is responsible for transferring data in and out of the µP.**
2) It is connected to the external memory and I/O devices, using the system bus.

L1 Code Cache

1) Pentium has an on chip **8 KB L1 Code cache**. It is 2 – way set associative.
2) It contains the most recently used instructions.

Prefetch Unit

1) It prefetches instructions from the L1 Code cache.
2) It has **two queues each of 32 bytes**.
3) One queue acts as the active queue, where as the other is used during branch prediction.

Decode Unit

1) It **decodes two instructions simultaneously for U and v pipes**.
2) Simple instructions are decoded by the hardwired control unit.
3) Complex instructions are decoded by the micro programmed control unit.

Integer Execution Unit

1) It can handle **two integer instructions simultaneously**.
2) **This first one goes to u-pipe and the second to v-pipe**.
3) There are address generation units for each pipe.
4) If the instruction uses memory operand the address generation unit generates physical address of the operand and fetches it form the **8 KB L1 Data Cache**.
5) There are two separate ALUs for U and V Pipes.
6) The **U-pipe ALU is equipped with a barrel shifter** and hence can handle complex arithmetic like MUL and DIV. Both ALUs are 32-bits each.
7) The integer unit uses 32-bit integer registers like EAX, EBX etc.

Floating Point Unit

1) It performs Floating Point operations.
2) It uses **80-bit F.P. Registers**.
3) It has its own F.P. Control unit and independent circuits for F.P. arithmetic operations.

Branch Prediction Logic

1) Pentium does branch prediction to minimize the pipeline penalty during branch operations.
2) It uses a **Branch Target Buffer with 256 entries**.
3) It gives history of previous branches and helps in **predicting the next branch instruction**.

Comparison of all processors

(Very Important, 5m – features of any one processor)

S No	Attribute	8085	8086	80286	80386	80486	Pentium
1	Processor Size	8 – bit	16 – bit	16 – bit	32 – bit	32 – bit	32 – bit
2	Data Bus	8 – bit	16 – bit	16 – bit	32 – bit	32 – bit	64 – bit
3	Memory Banks	--- NA ---	2 banks	2 banks	4 banks	4 banks	8 banks
4	Address Bus	16 – bit	20 – bit	24 – bit	32 – bit	32 – bit	32 – bit
5	Memory Size	64 KB	1 MB	16 MB	4 GB	4 GB	4 GB
6	Pipeline Stages	--- NA ---	2	3	3	5	5
7	ALU Size	8 – bit	16 – bit	16 – bit	32 – bit	32 – bit	32 – bit
8	No of Transistors	6500	29,000	1,34,000	2,75,000	11,80,235	31,00,000
9	Year of Release	1976	1978	1982	1985	1989	1993
10	Operating Frequency	3 MHz	6 MHz	12 MHz	33 MHz	60 MHz	100 MHz

SYLLABUS

EXTC

Subject Code	Subject Name	Teaching Scheme (Hrs.)			Credits Assigned			
		Theory	Practical	Tutorial	Theory	Practical	Tutorial	Total
ETC 403	Microprocessors and Peripherals	4	--	--	4	--	--	04

Subject Code	Subject Name	Examination Scheme							
		Theory Marks				Term Work	Practical and Oral	Oral	Total
		Internal assessment			End Sem. Exam				
		Test 1	Test 2	Avg. Of Test 1 and Test 2					
ETC403	Microprocessor and Peripherals	20	20	20	80	-	-	-	100

Course pre-requisite:

ETC 303 : Digital Electronics

Course objectives:

- To develop background knowledge and core expertise in microprocessor.
- To study the concepts and basic architecture of 8085, 8086, 80286, 80386, 80486 Pentium processor and Co-processor 8087.
- To know the importance of different peripheral devices and their interfacing to 8086.
- To know the design aspects of basic microprocessor.
- To write assembly language programs in microprocessor for various applications.

Course outcomes:

Students will learn

- The architecture and software aspects of microprocessor 8086
- Assembly language program in 8086 for various applications.
- Co-processor configurations.
- Various interfacing techniques with 8086 for various applications.
- Basic concepts of advanced microprocessors.

Module No.	Unit No.	Topics	Hrs.
1.0		**Architecture of 8085 and 8086 Microprocessor**	08
	1.1	8085 Architecture and pin configuration.	
	1.2	8086 Architecture and organization, pin configuration.	
	1.3	Minimum and Maximum modes of 8086.	
	1.4	Read and Write bus cycle of 8086.	
2.0		**Instruction set and programming of 8086**	10
	2.1	8086 Addressing modes.	
	2.2	8086 Instruction encoding formats and instruction set.	
	2.3	Assembler directives.	
	2.4	8086 programming and debugging of assembly language program.	
3.0		**Peripherals interfacing with 8086 and applications.**	10
	3.1	8086-Interrupt structure.	
	3.2	Programmable interrupt controller 8259A.	
	3.3	Programmable peripheral Interface 8255.	
	3.4	Programmable interval Timer 8254.	
	3.5	DMA controller 8257	
	3.6	Interfacing 8259A, 8255, 8254, 8257 with 8086 and their applications	
4.0		**ADC, DAC interfacing with 8086 and its application**	08
	4.1	Analog to Digital Converter (ADC) 0809	
	4.2	Digital to Analog Convertor (DAC) 0808	
	4.3	Interfacing ADC 0809, DAC 0808 with 8086 and their applications.	
	4.4	8086 based data Acquisition system.	
5.0		**8086 Microprocessor interfacing**	10
	5.1	8087 Math coprocessor, its data types and interfacing with 8086.	
	5.2	Memory interfacing with 8086 microprocessor	
6.0		**Advanced Microprocessors**	06
	6.1	Basic architectures of 80286, 80386, 80486 and Pentium processor.	
		Total	52

Text Books:

1. Gaonkar R.S.: "Microprocessor Architecture Programming and Applications with the 8085" Penram International Pub, 5th Edition.
2. John Uffenbeck: "8086/8088 family: "Design, Programming and Interfacing", Prentice Hall, 2nd Edition
3. B. B. Brey: "The Intel Microprocessors 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium and Pentium Pro Processor", Pearson Pub, 8th Edition

Reference Books:

1. Hall D.V: "Microprocessor and Interfacing Programming and Hardware", Tata McGraw Hill, 2nd Edition.
2. A. K. Ray and K. M. Burchandi: "Advanced Microprocessor and Peripherals, Architecture Programming and Interfacing", Tata McGrawHill, 3rd Edition
3. Don Anderson, Tom Shanley: "Pentium Processor System Architecture", MindShare Inc., 2nd Edition
4. National Semiconductor: Data Acquisition Linear Devices Data Book
5. Intel Peripheral Devices: Data Book.

Internal Assessment (IA):
Two tests must be conducted which should cover at least 80% of syllabus. The average marks of both the test will be considered for final Internal Assessment.

End Semester Examination:
1. Question paper will comprise of 6 questions, each carrying 20 marks.
2. The students need to solve total 4 questions.
3: Question No.1 will be compulsory and based on entire syllabus.
4: Remaining question (Q.2 to Q.6) will be selected from all the modules.

SYLLABUS

ETRX

Subject Code	Subject Name	Teaching Scheme			Credits Assigned			
		Theory	Practical	Tutorial	Theory	TW	Tutorial	Total
EXC 403	Microprocessor and Peripherals	04	--	--	04	--	--	04

Subject Code	Subject Name	Examination Scheme							
		Theory Marks				Term Work	Practical and Oral	Oral	Total
		Internal assessment			End Sem. Exam				
		Test 1	Test 2	Ave. Of Test 1 and Test 2					
EXC 403	Microprocessor and Peripherals	20	20	20	80	--	--	-	100

Course Objective:
To create a strong foundation by studying the basics of Microprocessors and interfacing to various peripherals which will lead to a well designed Microprocessor based System. The course is a pre-requisite for all further courses in Microcontrollers and Embedded systems.

Course Outcome:

1. Students will be able to understand and design Microprocessor based systems.
2. Students will be able to understand assembly language programming
3. Students will be able to learn and understand concept of interfacing of peripheral devices and their applications

Module No.	Topics	Hrs.
1	**Introduction to Intel 8085 Microprocessor:** Basic functions of the microprocessor, System bus, Architecture, Pin Configuration and Programmer's model of Intel 8085 Microprocessor.	06
2	**Intel 8086 Architecture:** Major features of 8086 processor, 8086/88 CPU Architecture and the pipelined operation, Programmer's Model and Memory Segmentation	06
3	**Instruction Set of 8086 and Programming:** Instruction Set of 8086 microprocessor in details, Addressing modes of 8086/88, Programming the 8086 in assembly language, Mixed mode Programming with C-language and assembly language. Assembler Directives Procedures and Macros.	10
4	**8086 Interrupts:** Interrupt types in 8086, Dedicated interrupts, Software interrupts,	04
5	**Designing the 8086 CPU module:** 8086 pin description in details, Generating the 8086 System Clock and Reset Signals, 8086 Minimum and Maximum Mode CPU Modules, Memory interfacing with timing consideration, Minimum and Maximum Mode Timing Diagrams	10
6	**Peripheral Controllers for 8086 family and System Design:** Functional Block Diagram and description, Control Word Formats, Operating Modes and Applications of the Peripheral Controller namely 8255-PPI, , 8259- PIC and 8237-DMAC. Interfacing of the above Peripheral Controllers. Keyword and Display Interface using 8255.	08
7	**Multiprocessor Systems:** Study of Multiprocessor Configurations namely Closely Coupled System (CCS) and Loosely Coupled System (LCS), CCS with the case study of the Maths Coprocessor, Various System Bus Arbitration Schemes in LCS, and Role of the Bus Arbiter (Intel 8289) in the LCS.	08
	Total	52

Recommended Books:

1) Microprocessor architecture and applications with 8085: By Ramesh Gaonkar (Penram International Publication).
2) 8086/8088 family: Design Programming and Interfacing: By John Uffenbeck (Pearson Education).
3) 8086 Microprocessor Programming and Interfacing the PC: By Kenneth Ayala
4) Microcomputer Systems: 8086/8088 family Architecture, Programming and Design: ByLiu & Gibson (PHI Publication).
5) Microprocessor and Interfacing: By Douglas Hall (TMH Publication).

Internal Assessment (IA):

Two tests must be conducted which should cover at least 80% of syllabus. The average marks of both the test will be considered as final IA marks

End Semester Examination:

1. Question paper will comprise of 6 questions, each carrying 20 marks.
2. The students need to solve total 4 questions.
3: Question No.1 will be compulsory and based on entire syllabus.
4: Remaining question (Q.2 to Q.6) will be set from all the modules.
5: Weightage of marks will be as per Blueprint.

QUESTION PAPERS

EXTC

University Question Paper
Extc Sem 4 Microprocessor May 2014

Duration: 3 Hours
Marks: 80

Question 1 is compulsory.
Out of the remaining attempt any three questions.

Q1. A) Explain the functions of the following pins of 8085
a) ALE b) SID/ SOD C) TRAP d) HOLD e) $\overline{\text{INTA}}$ ... 5 m

Q1. B) Explain the Control Word of 8254.
Write Control Word for Counter 0, Mode 2, R/W LSB, BCD Counter. ... 5 m

Q1. C) Write Features of 80286 microprocessor. ... 5 m

Q1. D) What are the advantages of memory segmentation ... 5 m

Q2. A) Draw and Explain architecture of 8085 microprocessor. ... 10 m

Q2. B) Explain Minimum Mode of 8086
Draw & explain timing diagram of write operation in Minimum Mode. ... 10 m

Q3. A) Draw and Explain interfacing of 8086 with 8255 in I/O mapped I/O ... 10 m

Q3. B) Write a program to generate a 1KHz frequency square wave using 8254.
Assume clock frequency of 8086 is 1 MHz ... 10 m

Q4. A) Draw and Explain interfacing of DAC 0808 with 8086 using 8255.
Write a program to generate a square Wave ... 10 m

Q4. B) Draw an interface diagram of 8086 with 8087.
Explain various interfacing signals & working of host & coprocessor ... 10 m

Q5. A) Design an 8086 based Minimum mode system with the following specifications
8086 microprocessor working at 8 MHz
32 KB EPROM using 16 KB Devices
32 KB SRAM using 16 KB Devices
Clearly show memory map with address range. Draw Neat Schematic. ... 10 m

Q5. B) Explain interrupt structure of 8086. ... 10 m

Q6. A) Write a program for 8086 microprocessor to exchange memory blocks
of 10 bytes between locations 30,000H and 40,000H. ... 10 m

Q6. B) Draw and Explain architecture of Pentium Processor. ... 10 m

University Question Paper
Extc Sem 4 Microprocessor Dec 2014

Duration: 3 Hours
Marks: 80

Question 1 is compulsory.
Out of the remaining attempt any three questions.

Q1. A) Explain functions of interrupt pins of 8085. ... 5 m

Q1. B) Explain the Control Word of 8254 Timer
Write Control Word for Counter 1, Mode 3, R/W MSB, binary counter. ... 5 m

Q1. C) Write Features of 80386 microprocessor. ... 5 m

Q1. D) Explain Features of 8087 Co Processor. ... 5 m

Q2. A) Draw and Explain architecture of 8085 microprocessor. ... 10 m

Q2. B) Explain Modes of 8255 PPI. ... 10 m

Q3. A) Draw and Explain interfacing of 8086 in Max Mode
with 8259 in Cascaded Mode. ... 10 m

Q3. B) Explain Max Mode of 8086.
Draw & explain timing diagram of write operation in Max Mode. ... 10 m

Q4. A) Draw and Explain interfacing of DAC 0808 with 8086 using 8255.
Write a program to generate a square Wave ... 10 m

Q4. B) Draw an interface diagram of 8086 with 8087.
Explain various interfacing signals & working of host & coprocessor ... 10 m

Q5. A) Design an 8086 based Minimum mode system with the following specifications
8086 microprocessor working at 8 MHz
16 KB EPROM using 8 KB Devices
16 KB SRAM using 8 KB Devices
Clearly show memory map with address range. Draw Neat Schematic. ... 10 m

Q5. B) What are the different types of interrupts supported by 8086.
Explain Interrupt Vector Table of 8088. ... 10 m

Q6. A) Write a program for 8086 microprocessor to exchange memory blocks
of 10 bytes between locations 30,000H and 40,000H. ... 10 m

Q6. B) Draw and Explain architecture of 80286 Processor. ... 10m

University Question Paper
Extc Sem 4 Microprocessor May 2015

Duration: 3 Hours
Marks: 80

Question 1 is compulsory.
Out of the remaining attempt any three questions.

Q1. **Attempt any four**

A) Explain flag register of 8085. ... 5 m

B) Explain the need & advantages of memory segmentation in 8086. ... 5m

C) Explain Addressing Modes of 8086 microprocessor. ... 5 m

D) Write a program to blink bit 4 of Port C using BSR mode of 8255 ... 5 m

E) Write features of 80486 Microprocessor. ... 5 m

Q2. Design an 8086 based system with the following specifications
1) 8086 microprocessor working at 6 MHz in Minimum Mode
2) 32 KB EPROM using 16 KB Devices
3) 64 KB RAM using 32 KB Devices
4) 2, 8-bit input & 2, 8-bit output ports in Memory Mapped I/O.

Design the system with Absolute Decoding.
Clearly show Memory Address Map and I/O Address Map.
Draw Neat Schematic for Chip Selection Logic. ... 20 m

Q3. A) Draw & Explain Interrupt Structure of 8086 with its IVT. ... 10 m

Q3. B) Draw and Explain interfacing of DAC 0808 with 8086 using 8255. Write a program to generate a square Wave ... 10 m

Q4. A) Explain interfacing of 8087 with 8086 microprocessor. ... 10 m

Q4. B) Draw timing diagrams of Memory Read and Memory Write Machine Cycles of 8086 in Max Mode ... 10 m

Q5. A) Explain Mode 0 and Mode 1 of 8254 PIT with timing Diagrams. ... 10 m

Q5. B) Explain diff modes of operations of 8257 DMA Controller. ... 10 m

Q6. A) Write a program for 8086 microprocessor to Multiply two 32-bit numbers (12345678H x 87654321H) ... 10 m

Q6. B) Write a program for 8086 microprocessor to find the smallest number In an array of 10 numbers ... 10m

University Question Paper
Extc Sem 4 Microprocessor Dec 2015

Duration: 3 Hours
Marks: 80

Question 1 is compulsory.
Out of the remaining attempt any three questions.

Q1. A) Explain the functions of the following pins of 8085
a) ALE b) SID/ SOD c) HOLD ... 5 m

Q1. B) Write Features of 80386 microprocessor. ... 5 m

Q1. C) Explain Interrupt pins of 8085 microprocessor. ... 5 m

Q1. D) Differentiate between Memory Mapped I/O and I/O Mapped I/O ... 5 m

Q2. A) Explain Addressing Modes of 8086 ... 10 m

Q2. B) What is 8087 Math Coprocessor?
Explain its interfacing with 8086 Microprocessor ... 10 m

Q3. A) Describe the importance of DMAC.
Explain interfacing of 8257 DMAC with 8086 Microprocessor. ... 10 m

Q3. B) What is data acquisition system.
Explain 8086 based data acquisition system. ... 10 m

Q4) Design an 8086 based Minimum mode system with the following specifications

- 8086 microprocessor working at 10 MHz
- 64 KB EPROM using 16 KB Devices
- 32 KB SRAM using 16 KB Devices

Clearly show memory map with address range. Draw Neat Schematic. ... 20 m

Q5. A) Write a program to sort a series of numbers in ascending order.
Store results from 08000H onwards ... 10 m

Q5. B) Explain interrupt structure of 8086. ... 10 m

Q6. A) Draw and Explain architecture of Pentium Processor. ... 10 m

Q6. B) Explain the function of ADC 0809.
Describe its interfacing with 8086 Microprocessor. ... 10 m

University Question Paper
Extc Sem 4 Microprocessor May 2016

Duration: 3 Hours
Marks: 80

Question 1 is compulsory.
Out of the remaining attempt any three questions.

Q1. A) Write features of Pentium Processor … 4 m

Q1. B) Differentiate between Memory Mapped I/O and I/O Mapped I/O in 8086 … 4 m

Q1. C) Describe in brief Architecture of 8085 … 4 m

Q1. D) Draw read and write bus cycle of 8086 with example … 4 m

Q1. E) Explain in brief Programmable interval Timer 8254 … 4 m

Q2. A) Explain various Addressing Modes of 8086 with Examples … 10 m

Q2. B) Explain with Suitable Examples the following Instructions of 8086 … 10 m
i) MOVSB ii) LEA iii) ROL iv) CLC v) CBW

Q3. A) Write an 8086 Program to add two 32-bit numbers and draw Flowchart … 10 m

Q3. B) Discuss function of General Purpose Registers of 8086.
Explain function of each register and instruction support for the same … 10 m

Q4. A) Explain function of following pins in Max mode of 8086 … 10 m
i) TEST ii) RQ/GT0 iii) RQ/GT1 iv) QS1, QS0 v) S0, S1, S2

Q4. B) Explain pin diagram of ADC 0809
And its interfacing with 8086 Microprocessor with Suitable Example … 10 m

Q5) Design an 8086 based system in Minimum Mode with the following specifications
ii) 8086 microprocessor working at 10 MHz
iii) 128 KB EPROM using 32 KB Devices
iv) 64 KB RAM using 16 KB Devices
Clearly show Memory Map with address range. Draw neat schematic … 20 m

Q6. A) Explain Interrupt Structure of 8086, and its method of interfacing with 8086 Microprocessor with a suitable example. (Any One Interrupt) … 10 m

Q6. B) Describe in brief and compare the architectures of 80286 and 80486. … 10 m

University Question Paper
Extc Sem 4 Microprocessor Dec 2016

Duration: 3 Hours
Marks: 80

Question 1 is compulsory.
Out of the remaining attempt any three questions.

Q1. A) Write features of 80486 Processor … 4 m

Q1. B) Differentiate between Min Mode and Max Mode of 8086 … 4 m

Q1. C) Describe Pin Diagram of 8085 … 4 m

Q1. D) Draw read and write bus cycle of 8086 with example … 4 m

Q1. E) Explain in brief Programmable Peripheral Interface 8255 … 4 m

Q2. A) Explain various Addressing Modes of 8086 with Examples … 10 m

Q2. B) Explain with Suitable Examples the following Instructions of 8086 … 10 m
i) CBW ii) TEST iii) LAHF iv) XLAT v) LEA

Q3. A) Write an 8086 Program to find Factorial of N and also draw Flowchart … 10 m

Q3. B) Discuss function of General Purpose Registers of 8086.
Explain function of each register and instruction support for the same … 10 m

Q4. A) Explain function of following pins in 8086 … 10 m
i) NMI ii) READY iii) ALE iv) QS1, QS0 v) S0, S1, S2

Q4. B) Explain pin diagram of ADC 0809
And its interfacing with 8086 Microprocessor with Suitable Example … 10 m

Q5) Design an 8086 based system in Minimum Mode with the following specifications
1) 8086 microprocessor working at 10 MHz
2) 32 KB EPROM using 8 KB Devices
3) 32 KB RAM using 8 KB Devices
Clearly show Memory Map with address range. Draw neat schematic … 20 m

Q6. A) Explain DMAC 8257, and its method of interfacing with
8086 Microprocessor with a suitable example. (Any One Interrupt) … 10 m

Q6. B) Describe in brief and compare the architectures of 80286 and 80386. … 10 m

QUESTION PAPERS

ETRX

University Question Paper
Etrx Sem 4 Microprocessor May 2014

Duration: 3 Hours
Marks: 80

Question 1 is compulsory.
Out of the remaining attempt any three questions.

Q1.) Answer the following questions ... 20 m
a) Explain Flag register of 8085 microprocessor.
b) What is REP Prefix? How does it function for String Instructions?
c) Explain the Feature of Pipelining and Queue in 8086 architecture.
d) Explain the significance of HOLD, RESET, READY signals in 8086 microprocessor.
e) For 8086 Opcode Fetch Machine Cycle, Explain the significance of each T-State.

Q2. A) Draw and Explain Instruction Template format of 8086. ... 10m
Q2. B) Explain PIC 8259 Features and operation. ... 10 m

Q3. A) Explain 8086 – 8087 co-processor configuration in Max Mode. ... 10 m
Q3. B) Explain the following instructions. ... 10 m
a) CMPSB
b) DIV AX
c) LOOPE Again
d) REP SCASB
e) XLATB

Q4. A) a) Write a detailed note on Interrupt Structure of 8086. ... 6 m
b) What are the basic modes of operation of 8255?
Explain with the format of the Control Word Register. ... 4 m
Q4. B) Explain the need for DMA and Modes of DMA Transfer. ... 10 m

Q5. A) Explain the architecture of 8086.
What is the Need for Memory Segmentation? ... 10 m
Q5. B) With the help of a neat flowchart/ algorithm, write an 8086 program to sort 10, 8-bit nos initialized in Data Segment in ascending order. ... 10 m

Q6. A) Write a brief note on 8255 PPI and its modes of operations. ... 10 m
Q6. B) Using String instructions, write an 8086 program to copy a block of 10 bytes initialized in Data Segment to Extra Segment. ... 10 m

University Question Paper
Etrx Sem 4 Microprocessor Dec 2014

Duration: 3 Hours
Marks: 80

Question 1 is compulsory.
Out of the remaining attempt any three questions.

Q1. A) Explain Flag Register of 8085 microprocessor. ... 5 m

Q1. B) Define Instruction Cycle, Machine Cycle and T-State ... 5 m

Q1. C) What is REP Prefix? How does it function for String Instructions? ... 5 m

Q1. D) Explain difference between a Jump and a Call instruction. ... 5 m

Q2.) Design an 8086 based system with the following specifications
1) CPU at 10 MHz in Minimum Mode
2) 32 KB SRAM using 8 KB Devices
3) 64 KB EPROM using 16 KB Devices
4) One 8255 for keyboard interface.

Design the system with Absolute Decoding.
Clearly show Memory Address Map and I/O Address Map.
Draw Neat Schematic for Chip Selection Logic. ... 20 m

Q3. A) Explain Interrupt Structure of 8086 microprocessor. ... 10 m

Q3. B) Discuss various addressing modes of 8086.
What are Displacement, Base and Index?
What is an Effective Address or Offset? ... 10 m

Q4. A) Write a program to find out the largest number in an array. ... 10 m

Q4. B) Write a program to find the number of times the letter "e" exists in "exercise". Store the count in memory. ... 10 m

Q5. A) Explain the interfacing of 8087 Co-Processor with 8086. ... 10 m

Q5. B) Sketch and Explain the interface of 8255 with 8086 in Minimum Mode.
Interface four 7 segment LEDs to display a BCD Counter. ... 10 m

Q6.) Write short notes on
a) Explain the functions of various flags of 8086 microprocessor. ... 5 m
b) Explain the functions of S0, S1, S2 of 8086 microprocessor. ... 5 m
c) Operation Modes of 8237 DMA Controller ... 5 m
d) Draw and explain instruction template format of 8086. ... 5 m

University Question Paper
Etrx Sem 4 Microprocessor May 2015

Duration: 3 Hours
Marks: 80

Question 1 is compulsory.
Out of the remaining attempt any three questions.

Q1. A) What is a stack. Explain the use and operation of Stack Pointer ... 5 m
Q1. B) What is an instruction queue? Explain? ... 5 m
Q1. C) Explain Flag Register of 8085 microprocessor. ... 5 m
Q1. D) Explain Lock (bar) and Test (bar) signals. ... 5 m

Q2.) Design an 8086 based system with the following specifications
5) CPU at 10 MHz in Minimum Mode
6) 64 KB SRAM using 8 KB Devices
7) 16 KB EPROM using 4 KB Devices
8) One 8255 for keyboard interface.
Design the system with Absolute Decoding.
Clearly show Memory Address Map and I/O Address Map.
Draw Neat Schematic for Chip Selection Logic. ... 20 m

Q3. A) Explain the first five dedicated interrupts of 8086. ... 10 m
Q3. B) Explain with one example each addressing mode of 8086. ... 10 m

Q4. A) Write an 8086 assembly language program to move a string of words From offset 1000H to offset 6000H. The Length of string is 0CH. ... 10 m
Q4. B) Explain the following assembler directive
CODE, ASSUME, ALIGN, EQU, EVEN, Various Data & Model directives ... 10 m

Q5. A) What are the different multiprocessor configurations? Explain a Closely Coupled configuration. ... 10 m
Q5. B) Sketch and Explain the interface of 8255 with 8086 in Minimum Mode. Interface four 7 segment LEDs to display a BCD Counter. ... 10 m

Q6.) Write short notes on
e) Difference between a Jump and a Call instruction. ... 5 m
f) Flag Register of 8086 microprocessor. ... 5 m
g) Operation Modes of 8237 DMA Controller ... 5 m
h) Procedure of interfacing 8259 with CPU. ... 5 m

University Question Paper
Etrx Sem 4 Microprocessor Dec 2015

Duration: 3 Hours
Marks: 80

Question 1 is compulsory.
Out of the remaining attempt any three questions.

Q1. A) Explain Interrupts of 8085 Microprocessor. ... 5 m
Q1. B) Compare Min Mode and Max Mode of 8086 Microprocessor. ... 5 m
Q1. C) Write an 8086 program to divide a 16-bit Number by a 4-bit Number. ... 5 m
Q1. D) Explain system bus arbitration in a Loosely Coupled System. ... 5 m

Q2. A) What is DMA. Explain 8237 DMAC ... 10 m
Q2. B) What is segmentation? Give usages, advantages of segmentation. ... 5 m
Q2. C) Compare 8085, 8086 and 8088 Microprocessors. ... 5 m

Q3. A) Design an 8086 based system with the following specifications
- 8086 microprocessor working at 3 MHz
- 6 KB EPROM
- 3 KB RAM
- 2 I/O ports

... 15 m
Q3. B)Explain Interrupt Acknowledge (INTA) cycle of 8086. ... 5 m

Q4. A) Explain Parameter Passing methods in 8086 ... 10 m
Q4. B) Write a program to divide a 32-bit number by an 8-bit number ... 5 m
Q4. C) What is instruction pipelining? Give its advantages and drawbacks ... 5 m

Q5. A) Explain interfacing of 8259 with 8086 in cascaded mode ... 10 m
Q5. B) Explain Closely Coupled System ... 5 m
Q5. C) Explain Assembler Directives ... 5 m

Q6. A) Give application of interrupts. Explain Interrupts of 8086 ... 7 m
Q6. B) Explain slow speed (memory) interface with 8086 with wait states with the help of timing diagram. ... 7 m
Q6. C) Explain Math Coprocessor and its usages. ... 5 m

University Question Paper
Etrx Sem 4 Microprocessor May 2016

Duration: 3 Hours
Marks: 80

Question 1 is compulsory.
Out of the remaining attempt any three questions.

Q1) Attempt **any four** from the following … 20 m

A) At reset, Interrupts of 8086 Microprocessor are disabled. Give reason.

B) List the differences between 8088 and 8086 Microprocessor.

C) Explain the feature of Pipelining and Queue in 8086 Architecture.

D) Explain the use of HOLD, RESET and READY pins of 8086 Microprocessor.

E) For 8086 opcode fetch machine cycle, explain significance of each T-State.

Q2. A) Classify and Explain 8086 instruction set … 10 m

Q2. B) Explain PIC 8259: Features and Operation … 10 m

Q3. A) Explain 8086-8087 Co-Processor Configuration in Max Mode … 10 m

Q3. B) Explain the following 8086 instructions … 10 m

- CMPSB
- DIV AX
- LOOPE again
- REP SCASB
- XLATB

Q4. A) Write a detailed note on Interrupt Structure of 8086 … 10 m

Q4. B) Explain the need for DMA and the Modes of DMA Transfer … 10 m

Q5. A) Explain Architecture of 8086. What is the need for Memory Segmentation … 10 m

Q5. B) With the help of a Flowchart/ Algorithm,
Write a program to sort a series of 10, 8-bit numbers,
defined in the data segment in Ascending order … 10 m

Q6. A) Write a brief note on 8255 PPI and its modes of operations … 10 m

Q6. B) Using String Instructions Write a program to transfer a block of 10 bytes
from Data Segment to Extra Segment … 10 m