Edition 2019

MU

Strictly as per the New Revised Syllabus (Rev - 2016) of

**Mumbai University**

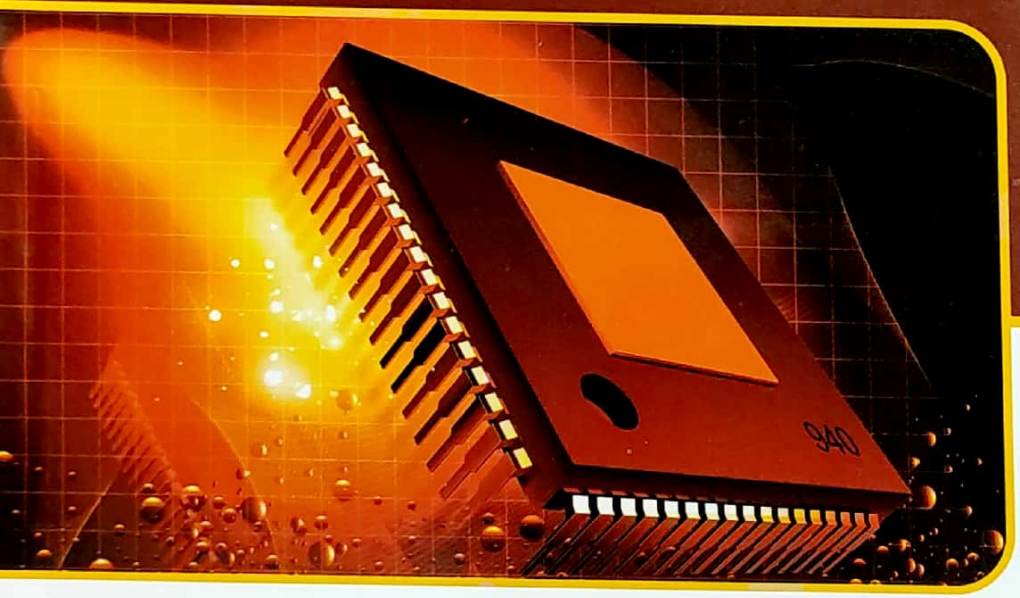
w.e.f. academic year 2018-19

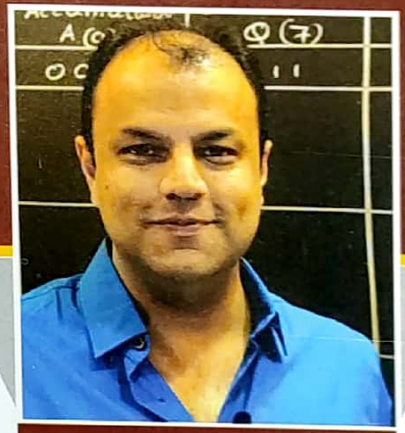
(As per Choice Based Credit and Grading System)

# MICROPROCESSOR

(Code : CSC501)

**Semester 5 - Computer Engineering**





**Bharat Acharya**

SUBJECTS | AUTHORS | LEADERSHIP | EXPERIENCE

OLD meets NEW to become

**BIGGER & BETTER**

with a Trusted Brand

**Feature**

- With Solved University Question Papers.


TECH-NEO PUBLICATIONS

*Where Authors Inspire Innovation*

A Sachin Shah Venture


(A311)

₹ 280/-

# Microprocessor

**(Code : CSC501)**

**Semester V - Computer Engineering** (Mumbai University)

**Strictly as per the Choice Based Credit and Grading System (Revise 2016) of Mumbai University w.e.f. academic year 2018-2019**


## Bharat Acharya

B.E. (Computers) Mumbai - 2000.
Founder | Bharat Acharya Education


Bharat Acharya has been teaching the subjects of Microprocessors and Microcontrollers amongst others since the year 2000. Students love his unique teaching style which emphasizes on practical implementation and discourages mugging up behavior. His methods inspire the students to follow a logical approach towards learning the subject and enjoy the science behind each topic.

In recent years he has started teaching online. He has a Youtube channel: "Bharat Acharya Education" where the initial introduction videos are available for free to watch. The channel has received millions of views and tens of thousands of likes and heartwarming comments praising his unique teaching style and vast knowledge of the subject.

The complete courses of all his subjects are available online on his website www.BharatAcharyaEducation.com and on the Android App: Bharat Acharya Education available on Google Play Store.

Download QR BarCode App

SCAN ME

Scan QR code to know about Author

Bharat Acharya Education

| | |
|---|---|
| Website | www.BharatAcharyaEducation.com |
| Mobile | +91 9820408217 |
| Email | bharatsir@hotmail.com |
| Youtube | youtube.com/c/BharatAcharyaEducation |
| Instagram | @BharatAcharyaEducation |
| Facebook | facebook.com/BharatAcharyaEducation |




TECH-NEO PUBLICATIONS
Where Authors Inspire Innovation
A Sachin Shah Venture

A311

**Microprocessor**

Bharat Acharya

Semester V – Computer Engineering (Mumbai University)

*(MCOMP11) (FM71) (MO36A)*



**First Edition :** July 2019

This edition is for sale in India, Bangladesh, Bhutan, Maldives, Nepal, Pakistan, Sri Lanka and designated countries in South-East Asia. Sale and purchase of this book outside of these countries is unauthorized by the publisher.

| Published by | Branch office | Printed at |
|---|---|---|
| **Mr. Sachin S. Shah & Mr. Rahul S. Shah**<br>Tech-Neo Publications LLP<br>407-412, 4th floor, Decision Tower, Above Hotel Tiranga, Nr. City Pride Theatre, Pune-Satara Road, Pune-411009. Maharashtra State, India.<br>Email : info@techneobooks.com<br>Website : www.techneobooks.com<br>Mobile : 9145531105 / 8668233261 | B/5, Ground floor, Maniratna Complex, Taware Colony, Aranyeshwar Corner, Pune - 411 009. Maharashtra State, India | **Image Offset (Mr. Rahul Shah)**<br>Dugane Ind. Area Survey No. 28/25, Dhayari Near Pari Company,<br>Pune – 41,Maharashtra State, India.<br>E-mail : rahulshahimage@gmail.com |

**Books Delivery Address**

Sr. No. 38/1, Behind Pari Compound, Khedekar Industrial Estate,
Narhe, Maharashtra,
Pune-411041.

# Preface

Dear students,

I am extremely happy to present the book of **" Microprocessor"** for you. I have divided the subject into small chapters so that the topics can be arranged and understood properly. The topics within the chapters have been arranged in a proper sequence to ensure smooth flow of the subject.

A large number of solved programs have been included. So, I am sure that this book will cater all your needs for this subject.

I am thankful to Shri. Sachin Shah for the encouragement and support that they have extended. I am also thankful to the staff members of Tech-Neo Publications and others for their efforts to make this book as good as it is. We have jointly made every possible efforts to eliminate all the errors in this book. However if you find any, please let me know, because that will help me to improve further.

I am also thankful to my family members and friends for their patience and encouragement.

**- Bharat Acharya**

□□□

# Syllabus

## Microprocessor

### (Mumbai University)

| Subject Code | Subject Name | Teaching Scheme (Hrs.) | | | Credits Assigned | | | |
|---|---|---|---|---|---|---|---|---|
| | | Theory | Practical | Tutorial | Theory | Practical | Tutorial | Total |
| CSC501 | Microprocessor | 04 | -- | -- | 04 | -- | -- | 04 |

**Course objectives :**

1. To equip students with the fundamental knowledge and basic technical competence in the field of Microprocessors.
2. To emphasize on instruction set and logic to build assembly language programs.
3. To prepare students for higher processor architectures and Embedded systems

**Course outcomes :** On successful completion of course learner will be able to:

1. Describe architecture of x86 processors.
2. Interpret the instructions of 8086 and write assembly and Mixed language programs.
3. Explain the concept of interrupts.
4. Identify the specifications of peripheral chip.
5. Design 8086 based system using memory and peripheral chips.
6. Appraise the architecture of advanced processors .

**Prerequisite :** Digital Electronics and Logic Design

| Module No. | Unit No. | Topics | Hrs. |
|---|---|---|---|
| 1.0 | | **The Intel Microprocessors 8086/8088 Architecture** | 10 |
| | 1.1 | • 8086/8088 CPU Architecture, Programmer's Model<br>• Functional Pin Diagram.<br>• Memory Segmentation.<br>• Banking in 8086.<br>• Demultiplexing of Address/Data bus.<br>• Study of 8284 Clock Generator.<br>• Study of 8288 Bus Controller.<br>• Functioning of 8086 in Minimum mode and Maximum mode.<br>• Timing diagrams for Read and Write operations in minimum and maximum mode. **(Refer chapters 1, 2 and 4)** | |

| Module No. | Unit No. | Topics | Hrs. |
|---|---|---|---|
| 2.0 | | **Instruction Set and Programming** | 12 |
| | 2.1 | • Addressing Modes<br>• Instruction set – Data Transfer Instructions, String Instructions, Logical Instructions, Arithmetic Instructions, Transfer of Control Instructions, Processor Control Instructions.<br>• Assembler Directives and Assembly Language Programming, Macros, Procedures.<br>• Mixed Language Programming with C Language and Assembly Language.<br>• Programming based on DOS and BIOS Interrupts (INT 21H, INT 10H) **(Refer chapter 2)** | |
| 3.0 | | **8086 Interrupts** | 6 |
| | 3.1 | • Types of interrupts.<br>• Interrupt Service Routine.<br>• Interrupt Vector Table.<br>• Servicing of Interrupts by 8086 microprocessor.<br>• Programmable Interrupt Controller 8259 – Block Diagram, Interfacing the 8259 in single and cascaded mode, Operating modes, programs for 8259 using ICWs and OCWs. **(Refer chapters 3 and 5)** | |
| 4.0 | | **Peripherals and their interfacing with 8086** | 12 |
| | 4.1 | Memory Interfacing - RAM and ROM Decoding Techniques – Partial and Absolute. | |
| | 4.2 | 8255-PPI – Block diagram, Functional PIN Diagram, CWR, operating modes, interfacing with 8086. | |
| | 4.3 | 8253 PIT - Block diagram, Functional PIN Diagram, CWR, operating modes, interfacing with 8086. | |
| | 4.4 | 8257-DMAC – Block diagram, Functional PIN Diagram, Register organization, DMA operations and transfer modes. **(Refer chapters 6, 7, 8 and 9)** | |
| 5.0 | | **Intel 80386DX Processor** | 6 |
| | 5.1 | • Architecture of 80386 microprocessor.<br>• 80386 registers – General purpose Registers, EFLAGS and Control registers.<br>• Real mode, Protected mode, virtual 8086 mode.<br>• 80386 memory management in Protected Mode – Descriptors and selectors, descriptor tables, the memory paging mechanism. **(Refer chapter 10)** | |
| 6.0 | | **Pentium Processor** | 6 |
| | 6.1 | Pentium Architecture.<br>Superscalar Operation, Integer & Floating Point Pipeline Stages, Branch Prediction Logic, Cache Organisation and MESI Model. **(Refer chapter 11)** | |
| | | **Total** | 52 |

□□□

# Index



❑❑❑


# 8086 Microprocessor

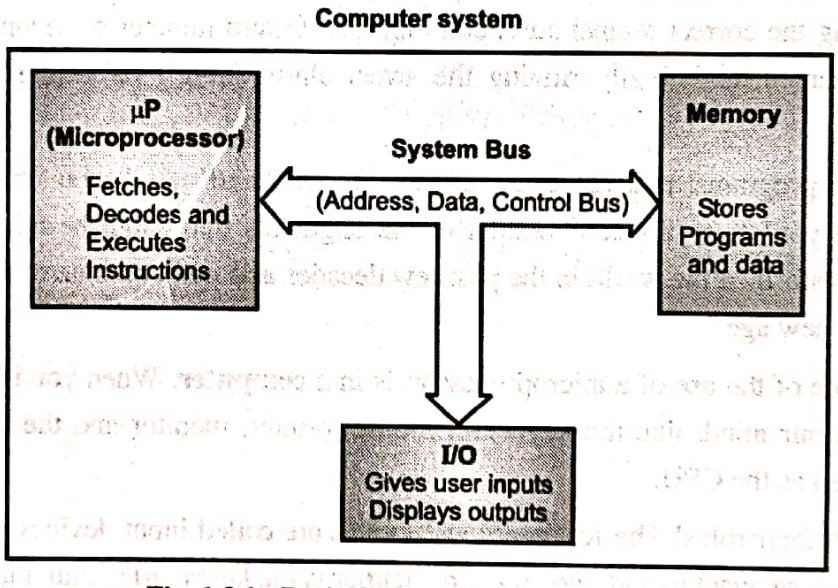
## 1.1 Introduction to Microprocessors

- Drones, stump vision cameras, mobile phones, RFID sensors, autonomous cars, streaming servers, your favorite shopping website... all of these are fruits of a seed called **"Microprocessor"**, planted years ago in mid 1970s and in fact conceived even earlier in 1940s.
- So what does this Microprocessor (henceforth called μP) actually do...Why do we need to learn it... And most importantly, after completing our education how will this knowledge be useful to us...
- **Where do we use a μP?** Is a μP used in a fan, or in a tube light, or in a switch board? No, none of them.
- Is it used in a mobile phone, a computer, a microwave oven, a washing machine? Yes, all of them! This is because they run on programs, and all those programs are executed by the microprocessor within these devices. **This is the main function of a μP, to execute programs.**
- In our day to day encounters we come across several devices and appliances. If you feel any of them works on a program, you should most certainly realize, it must contain a microprocessor. Take a microwave oven as an example.
- The μP inside the oven isn't directly cooking the food. It is running programs that are responsible for rotating the dish, maintaining the correct temperature, counting the desired number of seconds, displaying the time remaining on the screen, and finally ringing the sweet alarm (ding!) informing us that the cooking is complete.
- All of these require programs, that are executed by the oven's μP. And who writes the programs? The engineer, yes that's you! It is this combination of the engineers mind and the microprocessors execution abilities that has transformed the world in the past few decades and will continue to do so as both are getting ever so smart in the new age.
- The simplest example of the use of a microprocessor, is in a **computer**. When you imagine a computer, lots of devices come to our mind, like the keyboard, mouse, printer, monitor and the big "box" also casually referred to by laymen as the CPU.
- Of course you know their roles! The Keyboard and mouse are called input devices. Are they executing our programs? No! When we want to add two numbers, neither is the keyboard adding them nor is the mouse. So are they required? Yes! To give inputs. That's their role.
- To provide inputs to the system. Similarly, the printer and the monitor are used to produce outputs. Hence these are called I/O devices (Input and Output devices).
- This leaves us with that big "box". Open it and you see an electric circuit board also called the motherboard. **On the motherboard, pretty much around the center lies a big chip, around the size of our palm, that's your μP.**
- You may notice in modern motherboards the μP is generally covered with a fan, to dissipate the heat generated by relentlessly performing millions and sometimes billions of operations per second. So how important is the μP?

- Well, remove it and our computer becomes a piece of junk! Every activity of your computer needs a program, lets get you in agreement with that. You watch a movie, play a song, surf the net, be on social media etc., all of these are programs. Executing them keeps the μP busy round the clock, hence the heat and the fan.
- So lets say we are headed to the market right now, to buy the latest computer. Which microprocessor will you find inside? Yes, the Intel Core i7, or maybe Core i5 or i3, Intel also has recently released the Core i9 but its way too expensive to be mass produced as yet.
- So are we learning Core i7, i5, i3... not so early. These are a result of over 40 years of cutting edge development making them way to advanced to be understood by a beginner. We begin with learning the μP that started this whole x86 family... **The 8086 Microprocessor.**

## 1.2 Basic Organization of a Computer

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




(1A1)**Fig. 1.2.1 : Basic diagram of a computer system**

### 1.2.1 The Processor – "μP"

- The heart of the computer is its μP (Microprocessor).Current generation computers use processors like Intel Core i3, i5 or i7 and so on. They have come a long way from the initial processors that you are about to learn E.g.: 8085, 8086 etc.
- Back in the day (1940s), when micro-electronics was not invented, processors looked very different and were certainly not "micro" in appearance. They were created using huge arrays of physical switches which were operated manually and often occupied large rooms.
- In the following decades, with the invention of micro-electronics, scientists managed to embed thousands of

microscopic switches (transistors) inside a small chip, and called it a "**Micro-processor**".

- Over the years, microprocessors grew in strength. From housing a few thousand transistors (8085) to containing more than a billion transistors (Core i7), the computational power has been increasing exponentially. Having said that, some of the basics still remain the same.
- To put it simply, **the main function of a µP is to Fetch, Decode and Execute instructions.**
- **Instructions** are a part of programs. Programs are stored in the memory. First, **µP fetches an instruction** from the memory. It then **decodes the instruction**. This means, it "understands" the binary pattern of the instruction, also called its **opcode.**
- Every instruction when stored in the memory is in its unique binary form, which indicates the operation to be performed. **This is called its opcode**. Upon decoding the opcode, µP understands the operation to be performed and hence "executes" the instruction. This entire process is called an **"Instruction cycle".**
- This process is repeated for the next instruction. Like this, one by one, all instructions of a program are executed. Of course by new concepts like **pipelining, multitasking, multiprocessing** etc., this procedure has become very advanced and efficient today. You will get to learn all of them, in the due course of this ever intriguing subject.
- We begin learning with basic processors like **8085** or **8086**, but make no mistake, non of this is "outdated". Yes, your mobile phone or your computer today uses the most advanced processors (Apple's **A12Bionic** et.al), but to run a **traffic light** or **TV remote** control you don't need a core i7 now, do you?
- And these are used by the millions across the world. They simply use processors of the same grade as an 8085 or an 8086, with different product numbers as they are made by various manufacturers.

### 1.2.2 Memory

- Memory is used to **store information.**
- It stores two kinds of information… **programs and data.** Yes, think about it! Everything that's stored in your computer's memory is either some program or some data. MS Word is a program, The Word Documents are data. Your Image Viewer is a program, the images are data. WhatsApp is a program, its messages are data. Instagram is a program, the feed on the wall is the data… and so on!
- All programs and data are stored in the memory, in digitized form, where every information is represented in 1s and 0s called binary digits or simply bits.
- There are various forms of memory devices.
- The main memory also called primary memory consists of RAM and ROM.
- Other memory devices like Hard disk, Floppy, CD/ DVD etc. are secondary storage devices.
- Additionally there is also a high speed memory called Cache composed of SRAM.
- For the majority portion of this book, you are dealing with the initial processors like 8086.
- It will be in your best interest to think of Primary Memory only, whenever we speak of memory. That is because, secondary memory and high speed memories were implemented much later in the evolution of

processors as the demand for mass storage and high speed performance started increasing. So, from now on in this book, unless specified otherwise, **the word memory refers to primary memory that is RAM and ROM.**

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

### 1.2.3 I/O Devices

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

### 1.2.4 System Bus

- **A Bus is a set of lines.** They are used to transfer information, obviously in binary form.
- A line connected to $V_{CC}$ will carry a logic 1 and if connected to GND it will carry logic 0.
- Hence one line can transfer 1 bit. If we want multiple bits to be transmitted together, then we use a set of lines grouped together and that's called a bus.
- **The size of a bus refers to the number of lines** it contains and is always given in terms of bits.

  E.g.: An 8-bit bus has 8 lines; a 16 bit bus has 16 lines and so on.
- There are three types of buses... Address, Data and Control Bus. Collectively they are called the system bus. Let's take a closer look at them.

#### (i) Address Bus

- **It carries the address where the operation has to be performed.** Say the processor wants to write some data at a memory location. Firstly the processor will give the desired address on the address bus. This address will select a unique memory location. That's when data will be transferred with that location.
- It is therefore obvious that **bigger the address bus, more is the number of memory locations** it can address and hence bigger the total size of the memory. Here is the relation between size of the address bus and size of the memory.
- An address bus of 1 bit can give a total of two addresses: 0 and 1.
- Hence can access a total memory of two locations.
- A 2-bit address bus can generate a total of 4 addresses 00, 01, 10, 11 and hence can access a total of 4

locations.3-bit address bus… 8 locations and so on.

- So here is the rule,

  **An N-bit address bus can totally access $2^N$ locations.**

- As mentioned earlier, one memory location contains one byte of data.
- This brings us to the following conclusion.

  **A processor with an N-bit address bus can access a memory of $2^N$ Bytes.**

- Lets solve a typical VIVA (oral exam) question.
- Given the size of address bus, you have to figure out the size of the memory.

| Address Bus(N-bit) | Memory ($2^N$ Bytes) |
|---|---|
| 4 – bit | $2^4$ = 16 Bytes |
| 6 – bit | $2^6$ = 64 Bytes |
| 10 – bit | $2^{10}$ = 1024 Bytes = 1 KB |
| 16 – bit | $2^{16} = 2^6 \times 2^{10} = 64 \times 1$ K = 64 KB … (8085) |
| 20 – bit | $2^{20} = 2^{10} \times 2^{10} = 1$ K $\times 1$ K = 1 MB … (8086) |
| 30 – bit | $2^{30} = 2^{10} \times 2^{20} = 1$ K $\times 1$ M = 1 GB |
| 32 – bit | $2^{32} = 2^2 \times 2^{30} = 4 \times 1$ G = 4 GB …. (80386 and Pentium) |
| 40 – bit | $2^{40} = 2^{10} \times 2^{30} = 1$K $\times$ 1G = 1 TB … (Typical Hard Disk) |

- In case you found it a little difficult after the 4th row, that is because you may have got a little confused with the powers of 2. This issue will persist all along the subject. The smarter thing to do is once for all, let's get this hurdle past us.
- Let's get all powers of 2 clearly sorted. You will realize in the due course of this subject, how helpful this small exercise will prove to be. Frankly speaking there's rarely a topic in this subject that is not connected with some power of 2. Lets sort this, totally!

**Powers of 2**

| $2^N$ | Value |
|---|---|
| $2^0$ | 0 |
| $2^1$ | 2 |
| $2^2$ | 4 |
| $2^3$ | 8 |
| $2^4$ | 16 |
| $2^5$ | 32 |
| $2^6$ | 64 |
| $2^7$ | 128 |
| $2^8$ | 256 |
| $2^9$ | 512 |
| $2^{10}$ | $1024 = 1K$ |
| $2^{20}$ | $2^{10} \times 2^{10} = 1K \times 1K = 1M$ |
| $2^{24}$ | $2^4 \times 2^{20} = 16 \times 1M = 16M$ |
| $2^{28}$ | $2^8 \times 2^{20} = 256 \times 1M = 256M$ |
| $2^{30}$ | $2^{10} \times 2^{20} = 1K \times 1M = 1G$ |
| $2^{36}$ | $2^6 \times 2^{30} = 64 \times 1G = 64G$ |
| $2^{40}$ | $2^{10} \times 2^{30} = 1K \times 1G = 1T$ |

- Similarly, there is one more very important fundamental that needs to be sorted out before we start learning the bigger concepts. You need to be sure of number representation and number conversions. You may have been familiar with various number systems like Decimal, Hexadecimal, Binary etc... Out of all of them, Hexadecimal and Binary are the two most widely used number systems in our course of learning this subject.
- Every number we write, either in programs or in theory examples, is a hexadecimal number. When this

number gets stored inside the computer, it is in binary form This simply means, you must be very well versed with hex-binary conversions as this will be needed while understanding various examples.

- A single hexadecimal digit ranges from 0H... FH. This has 16 options. Hence, to represent the number in binary we need 4 bits as $2^4 = 16$. The following table shows this conversion.

| Hex | Binary |
|---|---|
| 0 H | 0000 |
| 1 H | 0001 |
| 2 H | 0010 |
| 3 H | 0011 |
| 4 H | 0100 |
| 5 H | 0101 |
| 6 H | 0110 |
| 7 H | 0111 |
| 8 H | 1000 |
| 9 H | 1001 |
| A H | 1010 |
| B H | 1011 |
| C H | 1100 |
| D H | 1101 |
| E H | 1110 |
| F H | 1111 |

- NO! You do not need to mug this up. There is a simple trick of "8421" that can get you any value from the above table. Consider the number 5 which is basically 4 + 1. Now consider the four binary bits corresponding to values 8, 4, 2 and 1.
- Since we need the equivalent of 5 which is 4 plus 1, we need 4 and 1 but we don't need 8 and 2 so in positions of 8 and 2 we put a "0" and in positions of 4 and 1 we put a "1". So we get 0101. Lets take the example of 0 H.
- We don't need any of them (8 or 4 or 2 or 1). So we put a "0" for all of them and hence get 0000. If we need F (which is 15), it is 8 + 4 + 2 + 1 so we need all of them and hence the binary equivalent is 1111. Take 9 as an example, 9 = 8 + 1. So we need 8 and 1 but not 4 and 2 so we put a "1" for 8 and 1 positions and a "0" for 4 and 2 positions giving us 1001. Hope you can now convert any hex digit into binary.
- Now consider a two digit number like 25H. To convert this to binary we need to substitute the 4 bit equivalents of 2 and 5 respectively. 2H is 0010 and 5H is 0101.
- Hence the number 25H in binary will be 0010 0101.Similarly a number like 74H will be 0111 0100 in

binary. 89H will be 1000 1001 and so on.

- As you noticed, the numbers 25H, 74H and 89H are all 2 digits in hexadecimal and hence need 8 bits in binary. Such numbers are called 8 bit numbers. The range of 8 bit and 16 bit numbers is mentioned below :

**8 Bit Numbers (Also called a "BYTE")**

| Hex | Binary |
|---|---|
| 00 H | 00000000 |
| 01 H | 0000 0001 |
| ... | ... |
| 68 H | 0110 1000 |
| ... | ... |
| FE H | 1111 1110 |
| FF H | 1111 1111 |

**16Bit Numbers (Also called a "WORD")**

| Hex | Binary |
|---|---|
| 0000 H | 0000 0000 0000 0000 |
| 0001 H | 0000 0000 0000 0001 |
| ... | ... |
| 4831 H | 0100 1000 0011 0001 |
| ... | ... |
| FFFE H | 1111 1111 1111 1110 |
| FFFF H | 1111 1111 1111 1111 |

### ☞ (ii) Data Bus

- **It carries data to and from the processor.**
- The size of data bus determines how much data can be transferred in one operation (cycle).
- Bigger the data bus, faster the processor, as it can transfer more data in one cycle.

### ☞ (iii) Control Bus

- **It Carries control signals like RD, WR etc.**
- These signals determine the kind of operation that will be performed on the system bus.

## 1.3 8086 | Salient Features

### (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 $\overline{\text{RD}}$ , $\overline{\text{WR}}$ 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 (Uniprocessor).

**(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.**

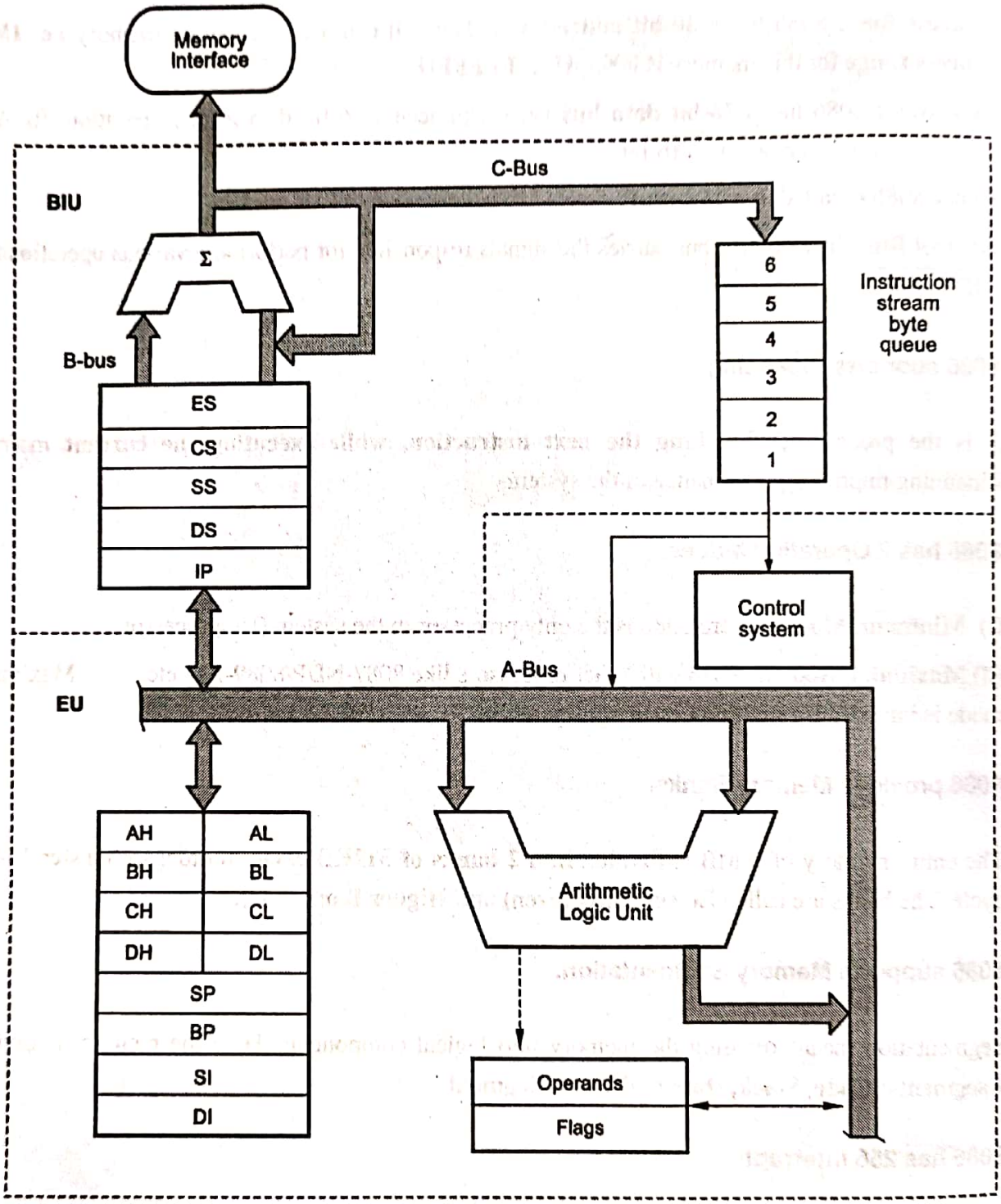
### (6) 8086 has 256 Interrupts.

The ISR addresses for these interrupts are stored in the IVT (Interrupt Vector Table).

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

✓ **Syllabus Topic : 8086/8088 CPU Architecture**

## 1.4 8086 | Architecture and Working




(1A2)**Fig. 1.4.1 : 8086 Architecture**

- 8086 is a 2-stage pipelined processor. The two stages of pipelining are fetching and execution.
- Based on this the 8086 architecture is divided into two functional units :

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

### 1.4.1 Bus Interface Unit (BIU)

- The main function of the BIU is to fetch instructions.
- Instructions are we know are stored in the memory are fetched using the various buses.
- The Bus Interface Unit connects the processor with the external components such as memory and I/O.
- All external data transfers with memory and I/O namely Memory Read, Memory Write, I/O Read and I/O write as well as Instruction Fetches are handled by the BIU. These operations are also called Machine cycles or Bus Cycles. Hence we say the BIU operates with respect to bus cycles.
- To perform any memory operation the BIU needs to calculate the 20-bit Physical Address.
- This is done using the formula Physical Address = (Segment Address × 10H) + Offset Address.
- Instructions are fetched from the code segment. CS register provides the segment address.
- IP register carries the offset address of the nest instruction.
- Hence the Physical Address is calculated as CS × 10H + IP.

  E.g.: If CS = 1234H and IP = 0005H, Physical Address = 12345H
- This calculation is done by the arithmetic circuit shown in the BIU.

#### ☞ Functions of BIU

- It performs the following functions :
  (1) It **generates** the 20-bit **physical address** for memory access.
  (2) **Fetches Instruction** from memory.
  (3) **Transfers data** to and from the **memory and IO.**
  (4) **Manages Pipelining** using the 6-byte instruction queue.
- The main components of the BIU are :
  (1) Segment Registers (2) IP
  (3) Address Generation Circuit (4) Prefetch Queue.

#### ☞ (1) Segment Registers

**CS Register**

- It is a 16-bit register.
- CS holds the **base** (Segment) **address** for the **Code Segment.**
- Code segment contains all the programs and hence is the segment from which instructions are fetched.
- The value of CS register is **multiplied by 10H** ($16_d$), to generate**20-bit physical address** of the starting 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

**DS Register**

- It is a 16-bit 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.

**SS Register**

- It is a 16-bit 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.

**ES Register**

- It is a 16-bit 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.

### ☞ (2) 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 × 10H + IP.**
- IP is **incremented after every instruction byte is fetched.**
- IP gets a new value whenever a branch occurs.

### ☞ (3) Address Generation Circuit

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

**Physical address = Segment Address × 10h + Offset Address**

- **Viva Question : Explain the real procedure to obtain the Physical Address?**

**The Segment address is left shifted by 4 positions, this multiplies the number by 16 (i.e. 10h) and then the offset address is added.**

Eg: If Segment address is 1234h and Offset address is 0005h, then the physical address (12345h) is calculated as follows :

$$1234h = (0001\ 0010\ 0011\ 0100)_{binary}$$

- Left shift by four positions and we get $(0001\ 0010\ 0011\ 0100\ \mathbf{0000})_{binary}$ i.e. 12340h
- Now add $(0000\ 0000\ 0000\ 0101)_{binary}$ i.e. 0005h and we get $(0001\ 0010\ 0011\ 0100\ 0101)_{binary}$ i.e. 12345h.

### ☞ (4) 6-Byte Pre-Fetch Queue

- 8086 implements 2-stage pipelining.
- **Fetching the next instruction while executing the current instruction is called pipelining.**
- The instructions that are pre-fetched are stored in the 6-byte prefetch queue.
- It works in first in first out manner which means the order in which instructions are fetched by the BIU is the same order in which they are removed by the EU for decoding and execution.
- **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.
- As EU removes instructions from the bottom end of the queue, there is empty space created at the top of the queue. **The queue is refilled when atleast two bytes are empty as 8086 has a 16-bit data bus.**

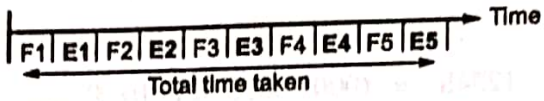
### ☞ Advantages of pipelining

- The obvious advantage of Pipelining is that it **increases** the **efficiency** of µP.
- Time is no longer spent on fetching as it happens alongside execution.
- The following graph shows the positive effect of pipelining on a processors performance.
- Consider a program of 5 instructions stored in the memory.
- The following is the behavior of a non pipelined processor like 8085 as compared to a pipelined processor like 8086. You can clearly see the speed advantage achieved by 8086 due to pipelining.

### ☞ Drawbacks of pipelining

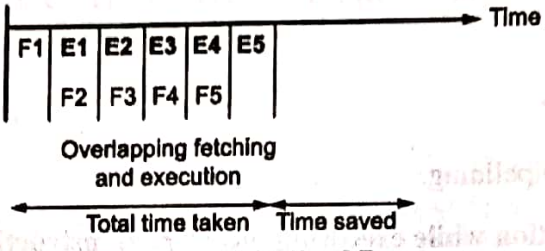
- Pipelining assumes the program will always be executed in a sequential manner.
- Hence it **fails when a branch** occurs, as the pre-fetched instructions are no longer useful. As soon as 8086 detects a branch operation, it clears/discards the entire queue.
- Now, the next six bytes from the new location (branch address) are fetched and stored in the queue and Pipelining continues.
- In advanced processors like Pentium which have very deep and superscalar pipelines, this drawback is almost neutralized due to a highly efficient technique called **Branch Prediction.**

☞ **Non-Pipelined Processor EG: 8085**




(1A3)**Fig. 1.4.2 : Non-Pipelined Processor**

☞ **Pipelined Processor EG: 8086**




(1A4)**Fig. 1.4.3 : Pipelined Processor**

## 1.4.2 Execution Unit (EU)

- The purpose of execution unit is to decode and execute instructions.
- Operations of the EU are confined within the μP.
- It does not directly perform any external memory or I/O based operation.
- Hence it is said that EU does not operate w.r.t. Machine cycles as it is not really performing any external activity. It simply operates w.r.t. clock cycles (T-states) of the processor.
- Whenever EU needs to transfer data from external memory or I/O it requests the BIU to do so.
- This especially happens when the instruction has memory operands.

**Note :** You will understand this more clearly after learning the various types of addressing modes.

- EU has several components such as :

1. Control Section
2. General Purpose Registers
3. ALU
4. Offset Registers
5. Flag Register

### ☞ 1. Control Section

- Instructions are already prefetched by the BIU and stored in the queue.
- The EU removes the first instruction from the queue.
- It goes into the Control section for decoding. Every instruction has a unique binary code that represents the operation to be performed called the opcode. Decoding simply means analyzing the opcode to understand the operation to be performed.
- The control section then generates internal control signals that inform all parts of the architecture as to what needs to be done for the execution of the corresponding instruction.

## 2. 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 independent **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)**

- AX is probably the most useful register from a programmers point of view.
- It is also called the Accumulator, which means it hold the first operand and the result in complex arithmetic operations like Multiply and Divide.
- **E.g. :: MUL BL;** BL will be multiplied with AL and the result will be stored in AX.

**Note :** Please include such examples of instructions in your exam answer to ensure maximum marks. You will find detailed explanation of these instructions in later chapters of this book.

- Moreover, **All IO data transfers** using IN and OUT instructions use "A" register.
- AL or Al for 8-bit data transfers and AX for 16-bit.
- It functions as accumulator during **string operations such as LODS and STOS** etc.

**BX Register (16-Bits)**

- BX is also called a memory pointer.
- It is the only General Purpose register that can **hold a memory address in Indirect addressing mode.**
- **E.g.:: MOV CL, [BX];** CL register gets the data from the memory location pointed by BX

**CX Register (16-Bits)**

- "C" register acts as a default count in many instructions.
- For Rotate and Shift instructions the count is in CL register.
- For Loop and String instructions the count is in CX register.
- **E.g.::LOOP Back :** This instruction creates a loop. The loop count is equal to the value in CX Register.

**DX Register (16-Bits)**

- DX acts as an extension of the accumulator when 32 bits are needed.
- E.g.:: In 16bit × 16 bit multiplication like **MUL BX,** the answer is 32-bits and is stored in a combination of DX and AX wherein DX gets the higher 16-bits and AX gets the lower.
- DX is also used to **hold the address** of the **IO Port** in **indirect IO addressing** mode.

### 3. ALU - Arithmetic and Logic Unit

- 8086 is a 16-bit μP. This means it has a 16-bit ALU.
- The ALU can perform 8-bit and 16-bit operations.
- **E.g.:: ADD BL, CL;** BL gets the result of BL + CL. This is an 8-bit operation.
- **E.g.:: ADD BX, CX;** BX gets the result of BX + CX. This is a 16-bit operation.
- Whenever temporary storage is needed in operations such as XCHG or MUL etc. the ALU uses a temporary register called "Operands". This is not available to the programmer.

### 4. Special Purpose Registers (Offset Registers)

**SP – Stack Pointer**

- SP is 16 bit offset register. SP 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.
- Physical Address of top of stack = SS × 10H + SP.
- It used during instructions like PUSH, POP, CALL, RET etc.
- During **PUSH** instruction, SP is **decremented** by 2.
- During **POP** instruction, SP is **incremented** by 2.

**BP – Base Pointer**

- BP is 16 bit offset register.
- BP can hold **offset address of** any location in the **stack segment.**
- It is used to access random locations of the stack.

**SI – Source index**

- SI is 16 bit offset register.
- It is normally used to hold the **offset address** for **Data segment** but can also be used for other segments using Segment Overriding.
- During String instructions like **"MOVSB"**, SI holds **offset address** of **source data** in Data Segment, hence the name Source Index.

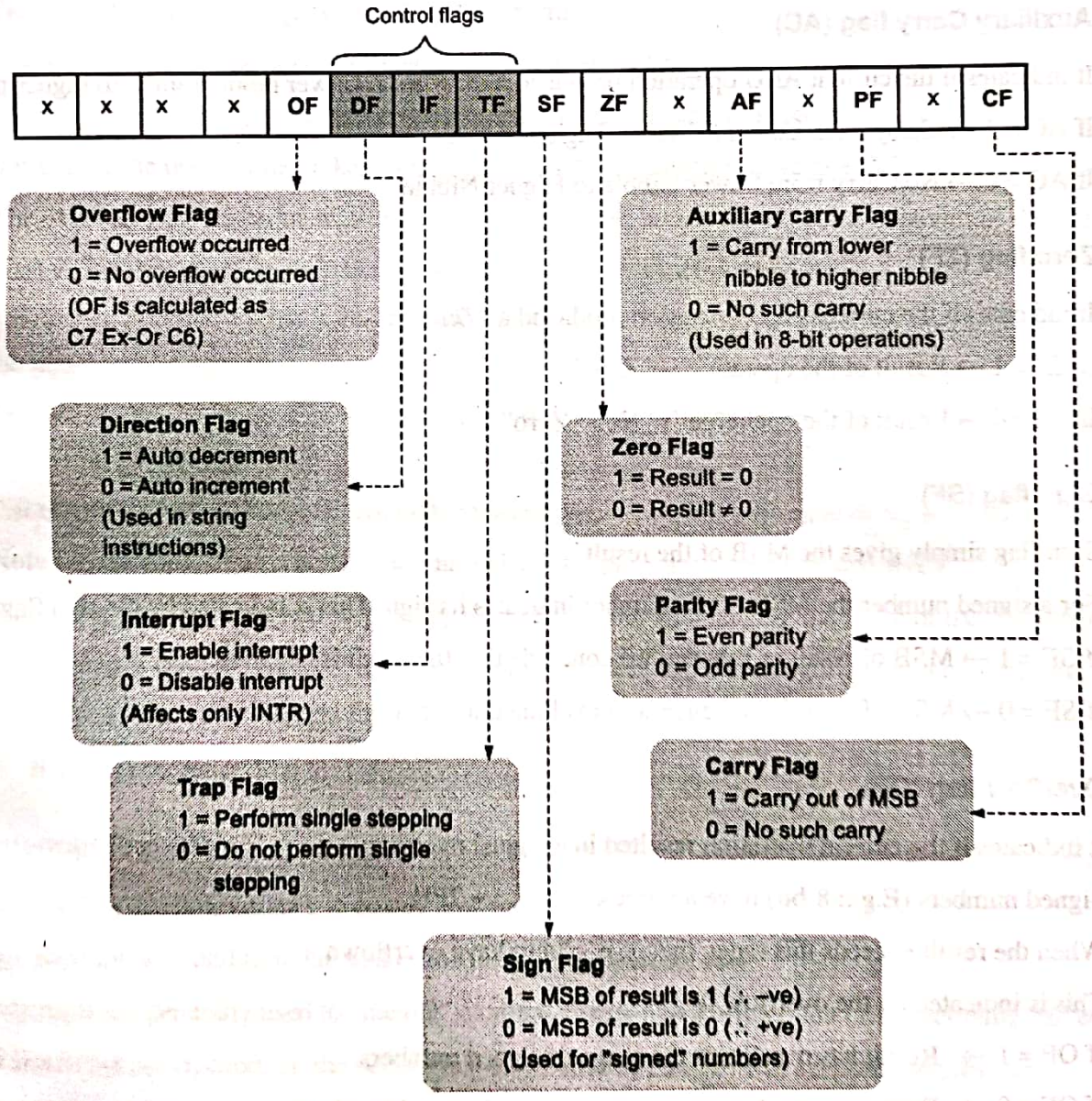
**DI – Destination index**

- DI is 16 bit offset register.
- It is normally used to hold the **offset address** for **Extra segment** but can also be used for other segments using Segment Overriding.
- During String instructions like **"MOVSB"**, DI holds **offset address** of **destination data** in Extra Segment, hence the name Destination Index.

## 5. Flag Register (Status Register)

**Q.** Explain flag register bits of 8086 **(May 18, 5 Marks)**

- 8086 has a 16-bit flag register.
- It has **9 Flags,** the rest are don't care bits denoted by the "x" symbol.
- It is a continuation of the flag register of 8085. The lower 8-bits are exactly the same as 8085 flags.
- These 9 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.




(1A5)**Fig. 1.4.4 : 8086 Flag Register**

**Status Flags**

☞ **(1) Carry flag (CF)**

- It indicates if the current ALU operation produced a carry out of the MSB.
- If CF = 1 → Carry out of MSB (or borrow in case of subtraction)
- If CF = 0 → No Carry out of MSB

☞ **(2) Parity flag (PF)**

- It indicates the "Parity" of the current result.
- If PF = 1 → Even Parity (Even number of 1's in the result)
- If PF = 0 → Odd Parity (Odd number of 1's in the result)

☞ **(3) Auxiliary Carry flag (AC)**

- It indicates if the current ALU operation produced a carry from Lower nibble (4bits) to higher nibble.
- If AC = 1 → Carry from Lower Nibble to Higher Nibble
- If AC = 0 → No Carry from Lower Nibble to Higher Nibble

☞ **(4) Zero flag (ZF)**

- It indicates if the current ALU operation produced a "Zero" result.
- If ZF = 1 → Result of the operation is "Zero"
- If ZF = 0 → Result of the operation is "Non - Zero"

☞ **(5) Sign flag (SF)**

- Sign flag simply gives the MSB of the result.
- For a signed number the MSB of the number indicates its sign. This is indicated by the sign flag.
- If SF = 1 → MSB of result is 1, hence we conclude that the result is negative.
- If SF = 0 → MSB of result is 0, hence we conclude that the result is positive.

☞ **(6) Overflow flag (OF)**

- It indicates if the current operation resulted in a signed overflow.
- Signed numbers (E.g.:: 8-bit) have a range of -80H...+ 7FH.
- When the result exceeds this range then it is said to have overflown.
- This is indicated by the overflow flag.
- If OF = 1 → Result has overflown the range of signed numbers
- If OF = 0 → Result has NOT overflown the range of signed numbers

**Important VIVA tip :** *Overflow makes the Sign flag acquire a wrong value. Always check the overflow flag before checking the sign flag. If OF = 1 then sign flag is wrong. If OF = 0, then sign flag is correct. Overflow is determined by C6 Ex-Or C7*

## Control Flags

### 1. Trap Flag (TF)

- It is used to start **Single Stepping Mode** also called Tracing or trapping the program.
- Single stepping is a debugging method.
- Sometimes when we run our program, we do not get the desired result.
- The program has finished execution that means there is no syntax error or runtime error.
- But the result produced is far from the expected value.
- This means there is a "Logical Error" in the program.
- Single Stepping is a tool to identify and solve such logical errors.
- Once in single stepping mode the processor executes one instruction and stops.
- It waits for the user to press a key, until then the program doesn't move ahead.
- The user can view the value of all registers and compare it with the expected behavior of the program.
- If all is fine that means the program logic is fine up to this instruction.
- Every time the user presses a key (generally - T) the program moves ahead by one instruction and then the same process repeats.
- This allows the programmer to see the output after every instruction and hence identify exactly at which instruction the program logic fails.
- ISR of INT1 is invoked after every instruction to pause the program and wait for the user to press a key.

  **Note :** In the topic of Interrupts, you will understand the link between TF and INT1.
- Single stepping is started and stopped by the programmer by changing the value of the Trap flag.
- As this flag is used to "control" the process of single stepping it is categorized as a control flag.

  If TF = 1 → "Start" Single stepping

  If TF = 0 →"Stop" Single stepping.

### 2. Interrupt Flag (IF)

- It is used by the programmer to enable and disable interrupts.
- An interrupt is a condition that makes the program execute an ISR.
- Interrupts are generally used to "handle" external events like a printer jam or an incoming call etc.
- If Interrupts are enabled, on the occurrence of an interrupt the µP will suspend the program.
- It will then service the interrupt by executing its ISR (Interrupt Service Routine).

- After completing the ISR, μP will come back and resume the earlier program from the next instruction.
- But if interrupts are disabled μP will simply ignore the interrupt when it occurs.
- In such a case the ISR will not be executed.
- As this flag is used to "control" the service it is categorized as a control flag.

  If IF = 1 → "Enable" interrupt

  If IF = 0 → "Disable" interrupt

**Important VIVA Tip :**

*In 8086 there are 256 interrupts.*

*We can never disable a software interrupt.*

*There are only two hardware interrupts namely NMI and INTR.*

*NMI stands for Non-Maskable Interrupt and hence cannot be disabled.*

*So the only interrupt that actually gets disabled by making IF=0 is INTR.*

*Please do not say in Viva that by making IF=0, "all" interrupts are disabled as that's a wrong answer!*

### 3. Direction Flag (DF)

- It is used by the programmer to Select the direction during string operations.
- A string instruction operates on a continuous block of data stored in the memory.
- The programmer may chose the incrementing direction which means after every operation the address of data will be automatically incremented or the decrementing direction.
- Based on this the offset address in SI and (or) DI may either be incremented or decremented after every operation, during a string instruction.
- As this flag is used to "control" the direction of operation, it is categorized as a control flag.

  If DF = 1 → "auto decrement" direction

  If DF = 0 → "auto increment" direction

**Note :** All flags by default are "0" on reset.
Thereafter as we perform operations, the ALU keeps modifying the status flags after every operation, reflecting the status of each result.
The control flags on the other hand remain unaffected and are changed by the programmer.

- Numerical examples of effects on status flags due to operations

**Example 1 :** 31H + 23H = 54H.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | 1 | | | | 1 | 1 | |
| | 31 H | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| + | 23 H | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| = | 54 H | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |

| Flags Affected: | OF | SF | ZF | AC | PF | CF |
|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 |

**Explanation**

1. Carry flag is "0" as there is no carry out of the MSB
2. Parity flag is "0" as there is Odd parity (Result has three 1s)
3. Auxiliary carry flag is "0" as there is no carry from lower nibble to higher nibble
4. Zero flag is "0" as the result is not zero (it is 54H)
5. Sign flag is "0" as the MSB of the result is "0"
6. Overflow flag is "0" as there is no overflow. The result is within the range of signed numbers.

**Example 2 :** 39H + 27H = 60H.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | 1 | 1 | 1 | | 1 | 1 | 1 |
| | 39 H | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| + | 27 H | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| = | 60 H | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

| Flags Affected: | OF | SF | ZF | AC | PF | CF |
|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 1 | 1 | 0 |

**Explanation**

1. Carry flag is "0" as there is no carry out of the MSB
2. Parity flag is "1" as there is Even parity (Result has two 1s)
3. Auxiliary carry flag is "1" as there is a carry from lower nibble to higher nibble
4. Zero flag is "0" as the result is not zero .
5. Sign flag is "0" as the MSB of the result is "0"
6. Overflow flag is "0" as there is no overflow. The result is within the range of signed numbers.

**Example 3 :** 42H + 44H = 86H.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 1 | | | | | | | |
| | 42 H | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| + | 44 H | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| = | 86 H | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

| Flags Affected: | OF | SF | ZF | AC | PF | CF |
|---|---|---|---|---|---|---|
| | 1 | 1 | 0 | 0 | 0 | 0 |

**Explanation**

1. Carry flag is "0" as there is no carry out of the MSB.
2. Parity flag is "0" as there is Odd parity (Result has three 1s).
3. Auxiliary carry flag is "0" as there is no carry from lower nibble to higher nibble.
4. Zero flag is "0" as the result is not zero.
5. Sign flag is "1" as the MSB of the result is "1"
6. Overflow flag is "1" as there is an overflow. The result is outside the range of signed numbers.

☞ **More clarity about overflow**

- 8-bit signed numbers have a range of – 80H to + 7FH.
- The result 86H is out of range for a "Signed" Number as it has become greater than +7FH.
- Such an event is called a "Signed Overflow".
- In such a case the MSB of the result gives a wrong sign.
- Though the result is +ve (+86H) the MSB is "1" indicating that the result is –ve.
- That is why before checking the Sign flag you may want to first check the overflow flag.
- Overflow is determined by doing an Ex-Or between the 2nd last Carry and the last Carry.
- Here the 2nd last Carry (the one coming into the MSB) is "1".
- The final carry (The one going out of the MSB) is "0".
- As "1" Ex-Or "0" = "1", the Overflow flag is "1".

➡ **Note :** In the exam show at least one, ideally two examples.
For more examples of positive and negative numbers watch my detailed video of flag register on my website www.BharatAcharyaEducation.com.

✓ **Syllabus Topic : Memory Segmentation**

## 1.5 8086 | Memory Segmentation

**Q.** Write short note Advantages of memory segmentation in 8086. **(Dec. 14, 5 Marks)**

**Q.** Write short note Advantages of memory segmentation in 8086. **(May 15, May 19, 5 Marks)**

**Q.** What is memory segmentation ? State Advantages of memory segmentation. **(Dec. 15, 5 Marks)**

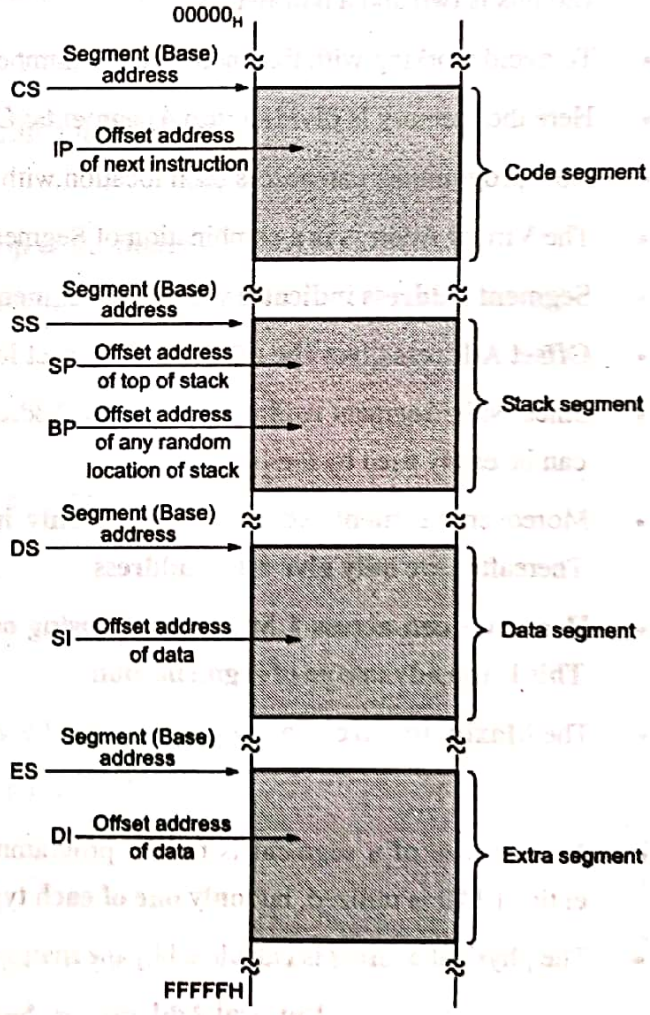
**Q.** Explain memory segmentation with PROS & CONS **(May 16, 8 Marks)**

**Q.** What is memory segmentation ? State Advantages of memory segmentation. **(May 17, 5 Marks)**

**Q.** Explain memory segmentation of 8086. **(May 16, 10 Marks, May 17, Dec. 17, 5 Marks, Dec. 18, 10 Marks)**

### Need For Segmentation/ Concept of Segmentation

- Memory is used to store information. This information can be either programs or data.
- Programs are a set of instruction stored in a sequential manner.
- After every instruction the address gets incremented.
- Data on the other hand can be either stored sequentially or just randomly at any location.
- Moreover there is an organized form of structured data called the stack, which is also present in the memory.
- Stack grows in the reverse manner, which means after every push operation the address is decremented.
- It is fairly simple to understand that all of these entities: programs, stack and data will eventually start overwriting on one another and will become increasingly difficult to manage as they all grow larger in size.
- To prevent this the memory is organized into 4 segments.
- The segments are named: Code, Stack, Data and Extra.
- Programs are stored in the Code Segment.




(1A6)**Fig. 1.5.1 : Memory Segmentation**

- Stack is present in the Stack Segment.
- Data Segment and Extra Segment are used to store ordinary data.
- These segments are initialized by the programmer right in the beginning of the program.
- Thereafter they are managed by the µP.
- This completely removes the fear of them overwriting on each other.
- Besides organizing the memory, segmentation presents another major advantage.
- It allows us to access the memory using 16-bit address as opposed to 20 bit.
- 8086 has a **20-bit address bus**, hence it can access $2^{20}$ Bytes i.e. **1MB** memory.
- But this also means that **Physical address** will now be **20 bit**.
- 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).
- To avoid working with this incompatible number, we **create a virtual model** of the memory.
- Here the memory is **divided into 4 segments**: Code, Stack Data and Extra.
- Now programmer can access each location with a **VIRTUAL ADDRESS**.
- The Virtual Address is a combination of Segment Address and Offset Address.
- **Segment Address indicates where the segment is located in the memory (base address)**
- **Offset Address gives the offset of the target location within the segment.**
- Since both, Segment Address and Offset Address are **16 bits each**, they both are **compatible numbers** and can be easily used by the programmer.
- Moreover, **Segment Address is given only in the beginning** of the program, to initialize the segment. Thereafter, we **only give offset address**.
- **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.**
- **The Maximum Size** of a segment is **64KB because offset addresses are of 16 bits.**

$$2^{16} = 64KB.$$

- 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.**
- The physical address is calculated by the microprocessor, using the formula :

$$\textbf{Physical Address} = \textbf{Segment Address} \times \textbf{10H} + \textbf{Offset Address}$$

Ex.: if Segment Address = 1234H and Offset Address is 0005H then

$$\text{Physical Address} = 1234H \times 10H + 0005H = 12345H$$

- This formula automatically ensures that the **minimum size of a segment is 10H bytes (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.

## 1.5.1 Advantages of Segmentation

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

## 1.5.2 Disadvantage of Segmentation

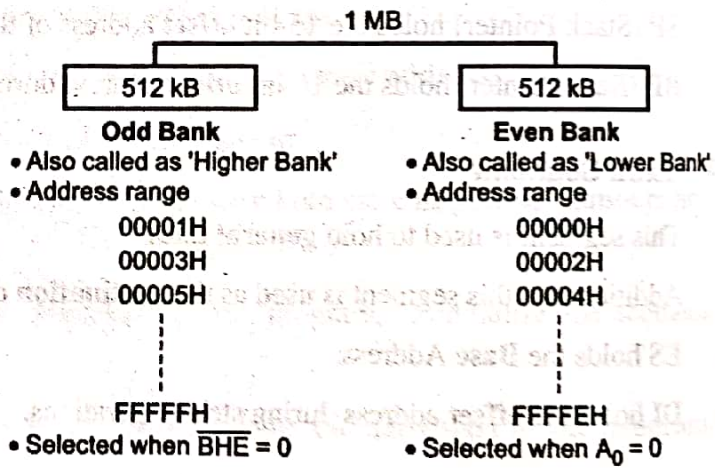
1. Although the total memory is 16*64 KB, at a time only 4*64 KB memory can be accessed.

✓ **Syllabus Topic : Banking In 8086**

## 1.6 8086 | Memory Banking

**Q.** Explain Memory banks for 8086 Processor. **(May 18, 5 Marks)**

- 8086 has a 16 bit data bus. This means it needs to transfer 16 bit data in one cycle.
- But we know that one location carries only one byte (8-bit) data.
- 16-bit data will be stored in two consecutive locations.
- Hence we need to device a method to read two consecutive locations in one cycle.
- To do so, memory is divided into two banks, such that consecutive locations fall into two separate chips.
- So one chip has even locations: 0, 2, 4… etc. called the even bank.
- The other has odd locations: 1, 3, 5… etc. called the odd bank.
- 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.**
- 16-bit data cannot be stored in one location.
- Reason is, if we make every location 16-bit wide then it means every operation will have to be a 16-bit operation. 8086 can perform 8-bit as well as 16 operations.
- Consider the programmer wanting to store an 8 bit number like 25H in the memory.
- If every location is 16-bit then we would have to occupy a 16-bit location in storing an 8-bit data.
- This simply means the other 8-bit will get wasted.




(1A7)**Fig. 1.6.1 : 8086 Banking**

| $\overline{BHE}$ | A0 | Operation |
|---|---|---|
| 0 | 0 | Read or Write **16-bits** from both banks |
| 0 | 1 | Read or Write **8-bits** from higher bank |
| 1 | 0 | Read or Write **8-bits** from lower bank |
| 1 | 1 | No Operation (Processor is Idle) |

- 8086 μP has a 20 bit address bus ($A_{19}...A_0$)
- Out of this : $A_{19}... A_1$ are used to identify the location within the bank.
- $A_0$ and $\overline{BHE}$ are used to select the banks.
- μP generates $A_0$ and $\overline{BHE}$ on the basis on the instruction written by you, the programmer.
- This is explained in the following examples :

☞ **8-bit operation on even bank**

- **E.g.:: MOV BL, [2000H]**
- This is an **8 bit** operation.
- **BL** register gets the 8-bit value stored at offset **address 2000H.**
- Since it is an **8-bit operation** μP will select only **one bank.**
- Since the **address is even,** μP will select **Even Bank.**
- Hence here **$A_0$** will be **"0"** and **$\overline{BHE}$** will be **"1".**

☞ **8-bit operation on odd bank**

- **E.g.:: MOV BH, [2001H]**
- This is an **8 bit** operation.
- **BH** register gets the 8-bit value stored at offset **address 2001H.**
- Since it is an **8-bit operation** μP will select only **one bank.**
- Since the **address is odd,** μP will select **Odd Bank.**
- Hence here **$A_0$** will be **"1"** and **$\overline{BHE}$** will be **"0".**

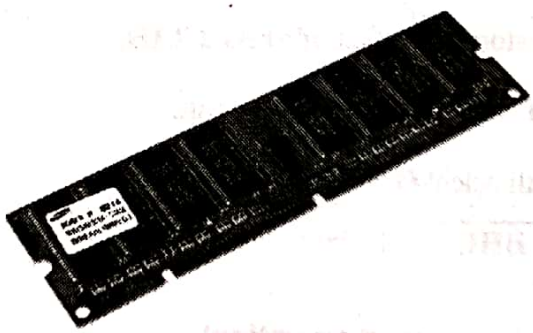
☞ **16-bit operation on both banks (aligned operation)**

- **E.g.:: MOV BX, [2000H]**
- This is an **16 bit** operation.
- **BX** register gets the 16-bit value stored at offset **address 2000H and 2001H.**
- Since it is an **16-bit operation** μP will select **both banks.**
- When a 16 bit operation **begins with an even address** it is said to be **aligned.**
- Aligned data can be accessed in one cycle.

- Since the **address is even**, it is an "aligned operation" µP will select **Both Banks in one cycle.**
- Hence here $A_0$ will be **"0"** and $\overline{\text{BHE}}$ will also be **"0".**

### ☞ 16-bit Misaligned operation (Important for Viva)

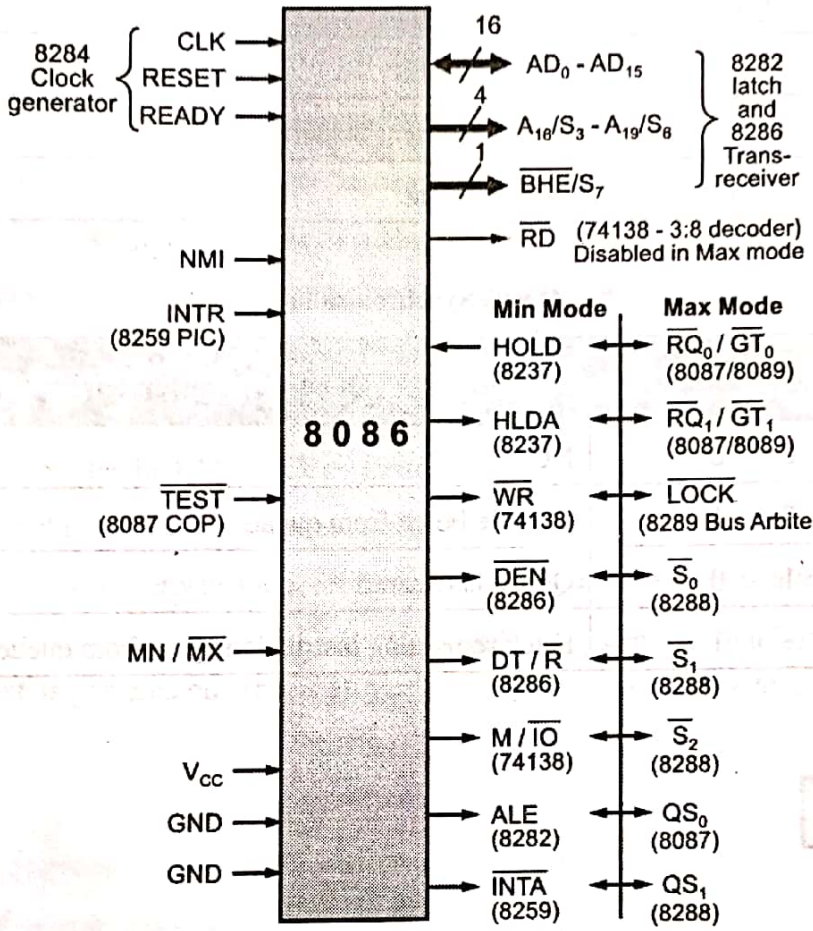
- **E.g.:: MOV BX, [2001H]**
- This is an **16 bit** operation.
- **BX** register gets the 16-bit value stored at offset **address 2001H and 2002H.**
- When a 16 bit operation begins with an odd address it is said to be misaligned.
- Though misaligned data is valid data, such data cannot be accessed in once cycle.
- µP will break it down into two cycles and access from 2001H first and then from 2002H and two separate 8 bit operations.
- Similarly, a processor with a 32-bit data bus will need to access 4 locations.
- Hence the memory will be divided into 4 banks (E.g.:: 80386)
- A processor with a 64 bit data bus will need to access 8 memory locations.
- Hence the memory will be divided into 8 banks (E.g.:: Pentium and all modern processors)
- This is why modern RAM comes as a set of 8 chips.
- Typical image of a modern RAM supporting 64 bit operation using 8 memory banks.



(1A8)**Fig. 1.6.2 : 8 Memory banks of a typical modern RAM**

✓ **Syllabus Topic : Functional Pin Diagram**

## 1.7 8086 | Pin Diagram




(1A9)**Fig. 1.7.1 : 8086 Pin Diagram**

**Machine Cycles**

| $\overline{S_2}$ | $\overline{S_1}$ | $\overline{S_0}$ | 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 from queue |

## Pin Descriptions

### 1.7.1 CLK

- This pin provides the clock input to the processor 8086.
- The purpose of clock is to trigger operations.
- Each clock pulse acts as a trigger to cause a state change (a new activity) in the processor.
- Hence a clock pulse is also called a T-State (transition state).
- 8086 works on a frequency of 6 MHz.
- Hence one T-state = 1/(6 MHz) = 1.667 μsec or 166 nanoseconds.
- An external clock generator (8284) provides the clock signal.
- 8086 requires a 33% duty cycle, TTL clock signal.
- Hence the crystal connected to 8284 is of 18 MHz.
- 80884 will divide the crystal frequency by 3 to generate a clock of 6 MHz at 33% duty cycle.

## 1.7.2 RESET

**Q.** Write short note on generation of RESET Signals in 8086 based system **(Dec. 15, 5 Marks)**

**Q.** Explain power on reset circuit used in 8086 system. **(Dec. 17, 5 Marks)**

- This is the reset input signal used to "reset" the μP and hence the whole computer.
- On reset : CS becomes FFFFH and IP becomes 0000H.
- This automatically means the Physical address becomes FFFF0H (CS × 10H + IP)
- This address "FFFF0H" is called the reset vector address of 8086.
- It means whenever 8086 is reset it will always go to this address to fetch the first instruction.
- The program stored at this address is the first program to be executed by the μP on reset.
- It is called the "monitor program" also called the BIOS program, used to "boot up" the system.
- The reset signal is provided by the 8284 clock generator.
- Generation of reset signal is done using a **power on reset circuit.**
- The purpose of this circuit is to activate the reset signal as soon as we supply power to 8086.
- This is done so that CS & IP acquire the above values and the system can initialize the monitor program (BIOS) as soon as 8086 is powered on. This is also called COLD starting the system.

## 1.7.3 READY

- This signal is used to synchronize the μP with **slower peripherals.**
- Devices inform the μP whether they are ready or not.
- μP **samples** (checks)the READY input **during $T_3$ state** of every Machine Cycle
- If devices are "ready", the Ready signal will be "1".
- In such a case μP simply continues the operation as usual.
- If devices are "not ready", the Ready signal will be "0".
- In such a case μP will enter "wait state".
- Wait states are basically empty clock cycles inserted between $T_3$ and $T_4$ states of any machine cycle.
- It simply means μP is simply allowing slow devices more time to become ready and complete the operation.
- μP remain in wait state for as long as Ready signal remains 0.
- Once Ready signal becomes "1" μP comes out of wait states and proceeds ahead with the operation.
- Hence we say Ready signal helps synchronize the μP with slower devices.

### 1.7.4 $\overline{\text{TEST}}$

- It is an active low **input** line dedicated for 8087 Co-processor.
- **$\overline{\text{TEST}}$ can only be connected with 8087 in Max mode.**
- Otherwise this signal is of no consequence.
- **$\overline{\text{TEST}}$** is used by 8086 to check whether 8087 is busy or not.
- BUSY pin of 8087 is connected to **$\overline{\text{TEST}}$** of 8086.
- µP checks **$\overline{\text{TEST}}$** when we write WAIT instruction. WAIT is written just before an 8087 instruction.
- If **$\overline{\text{TEST}}$** is "0" it means 8087 is not busy. In this case µP will simply continue.
- If **$\overline{\text{TEST}}$** is "1" it means 8087 is still busy completing the previous instruction.
- In such a case µP must not go ahead with the following instruction as the forthcoming instruction is again if 8087 and needs 8087 to be free for its execution. Hence µP will enter wait state. This allows 8087 extra time to complete its operation.
- Once 8087 finishes its execution, it makes BUSY signal "0".This makes **$\overline{\text{TEST}}$** signal "0".
- Now µP comes out of wait state and will proceed ahead with the program.

**Important for VIVA :** *If there is no 8087 then $\overline{\text{TEST}}$ must be grounded. This is so that if the programmer has accidentally written a WAIT instruction, the µP will check the $\overline{\text{TEST}}$. As it is grounded, $\overline{\text{TEST}}$ will be "0" and hence µP will not be forced to enter wait state indefinitely.*

### 1.7.5 MN/ $\overline{\text{MX}}$

- This is an **input** signal to 8086.
- It is used to inform the µP whether it is working in minimum mode or maximum mode.
- If **MN/ $\overline{\text{MX}}$** is "1" that means the µP is working in minimum mode.
- In minimum mode there is only on processor that is 8086.
- If **MN/ $\overline{\text{MX}}$** is "0" that means the µP is working in maximum mode.
- In maximum mode there can be multiple processors alongside 8086 in the same circuit.
- We could connect an 8087, which could work as the numeric processor for powerful arithmetic operations such as trigonometry, log etc..
- We could also connect an 8089 which works as a dedicated I/O processor used to perform sophisticated I/O data transfers.
- The behavior of a lot of pins of µP changes when it switches from min mode to max mode.
- Hence on reset µP immediately checks this signal to determine how the pins and the µP itself must function.

### 1.7.6 NMI

- It is one of the two **hardware interrupt** input lines to 8086.
- The other one being INTR.
- NMI is the **higher priority** interrupt.
- It means if NMI and INTR occur together, NMI will be serviced first.
- NMI is **Non Maskable**, hence the name.
- It means NMI cannot be disabled by the CLI instruction. CLI stands for Clear Interrupt Flag.
- NMI is a **Vectored interrupt.**
- It means NMI has a fixed vector number in the IVT. Its vector number is 2.
- On receiving an interrupt on NMI line, the μP executes **INT 2** i.e. and takes control to location $2 \times 4 = 00008H$ in the Interrupt Vector Table (IVT), to get the value for CS ad IP.
- NMI is **edge-triggered.**
- It means it is recognized on the rising edge (when it goes from low to high) by the μP.
- NMI is typically used for hardware failures which require immediate attention such as failure, overheating etc.

### 1.7.7 INTR

- It is one of the two **hardware interrupt** input lines to 8086.
- The other one being NMI.
- INTR is the **lower priority** interrupt.
- It means if NMI and INTR occur together, NMI will be serviced first.
- INTR is **Maskable,** hence the name.
- It means INTR cannot be disabled by the CLI instruction.
- INTR is a **Non Vectored interrupt.**
- It means INTR does not have a fixed vector number in the IVT.
- On receiving an interrupt on INTR line, the μP executes **2 $\overline{\text{INTA}}$** cycles.
- **On FIRST $\overline{\text{INTA}}$** pulse, the interrupting device (8259) prepares **to send a vector number "N".**
- **On SECOND $\overline{\text{INTA}}$** pulse, the interrupting device (8259 PIC) **sends vector number "N"** to μP.
- Now μP will multiply $N \times 4$ and go to the IVT to obtain the ISR address i.e. values for IP and CS.
- On receiving an interrupt on NMI line, the μP executes **INT 2** i.e. and takes control to location $2 \times 4 = 00008H$ in the Interrupt Vector Table (IVT), to get the value for CS ad IP.
- All of this is explained in far more detailed manner in the topic of 8086 interrupts.

- INTR is **level-triggered.**
- It means it is recognized on the high level (stable logic 1) by the μP.
- INTR is used to **increase the number of interrupt sources** to the μP.
- Being non vectored, INTR provides the flexibility to connect multiple interrupt inputs. This is achieved by using an interrupt controller like 8259 explained later in this book.

### 1.7.8 Distinguish between NMI and INTR

| Sr. No. | NMI | INTR |
|---|---|---|
| 1. | Non Maskable | Maskable |
| 2. | Edge Triggered | Level Triggered |
| 3. | High Priority | Low Priority |
| 4. | Vectored | Non Vectored |
| 5. | Acknowledgement not needed | Acknowledgement needed using $\overline{\text{INTA}}$ signal |
| 6. | Rigid as it is vectored | Non vectored hence flexible |
| 7. | Used for critical hardware failures like disk error etc… | Used to increase the number of interrupts using an interrupt controller like 8259 PIC. |

### 1.7.9 Vcc and GND

- Used for power supply.
- 8086 is based on the universal TTL logic.
- Here logic 1 is 5V and logic 0 is 0V.
- Two grounds are provided for easier dissipation of current.
- 8086 has around 29,000 transistors.
- Releasing current through two ground pins reduces the load on those pins.
- You will notice this as you study higher processors with much larger transistor count ranging in the millions. As the internal circuit gets more and more dense, the number of Vcc and GND pins increase to reduce and hence balance the load on those pins.

## 1.7.10 Address and Data Buses

**Q.** Explain the multiplexed buses of 8086.

- 8086 has a multiplexed address/ data bus.
- **Multiplexing means reducing the number of pins by combining signals.**
- The address bus is 20 bit: $A_{19} - A_0$
- Along with this μP also generates $\overline{\text{BHE}}$.
- This makes it 21 pins.
- Similarly, the date bus is 16-bit: $D_{15} - D_0$.
- Along with these the μP also gives 5 status signals $S_7 - S_3$.
- This also adds up to another 21 pins.
- If they were not multiplexed, just these signals would have consumed 42 pins of the μP.
- Interestingly, address and data are never given out simultaneously in any operation.
- **In any operation (machine cycle), μP first generates address.**
- This address selects the desired target location. Then data is transferred.
- Hence we could use a "common" bus for address and data.
- This will tremendously reduce the number of pins. The earlier mentioned 42 pins are reduced to just 21 pins.
- $A_0 - A_{15}$ are multiplexed with $D_0 - D_{15}$ giving $AD_0 - AD_{15}$
- $A_{16} - A_{19}$ are multiplexed with $S_3 - S_6$ giving $A_{16}/S_3 - A_{19}/S_6$.
- $\overline{\text{BHE}}$ is multiplexed with $S_7$ giving $\overline{\text{BHE}}/S_7$.
- The signal ALE is used to identify which signal is present in the multiplexed bus.
- When **ALE = 1** the respective buses carry $A_0 - A_{15}$, $A_{16} - A_1$ and $\overline{\text{BHE}}$
- When **ALE = 0** the respective buses carry $D_0 - D_{15}$, $S_3 - S_7$.
- As you learn timing diagrams you will know that ALE is high during the 1st T-state of any machine cycle.
- At that time the buses carry address and $\overline{\text{BHE}}$. Thereafter ALE remains low for the rest of the machine cycle. At that time the buses carry data and status.

| Multiplexed Signals → | $AD_0 - AD_{15}$ | $A_{16}/S_3 - A_{19}/S_6$ | $\overline{\text{BHE}}/S_7$ |
|---|---|---|---|
| When ALE = 1 → | $A_0 - A_{15}$ | $A_{16} - A_{19}$ | $\overline{\text{BHE}}$ |
| When ALE = 0 → | $D_0 - D_{15}$ | $S_3 - S_6$ | $S_7$ |

- ALE stands for address data enable. At a later point in the subject you will know that ALE is used to latch the address out of this multiplexed bus using address latches like IC 8282.

**Status signals : $S_3$ - $S_7$**

Status signals are given out by the μP to indicate the status of its current activities.

**$S_3$ $S_4$: These signals indicate which segment is being accessed by the μP in the current operation.**

| $S_4$ $S_3$ | Segment |
|---|---|
| 0 0 | Extra Segment |
| 0 1 | Stack Segment |
| 1 0 | Code Segment (or idle) |
| 1 1 | Data Segment |

**$S_5$ : Indicates if interrupt is enabled or disabled.**

If $S_5 = 1 \rightarrow$ This means the Flag IF is 1. Hence Interrupt is enabled.

If $S_5 = 0 \rightarrow$ This means the Flag IF is 0. Hence Interrupt is disabled.

**#Viva**

- Please do keep in mind that IF (Interrupt Flag) only affects hardware interrupts.
- We can never disable a software interrupt.
- Out of the two hardware interrupts also, NMI is non Maskable.
- This means NMI cannot be disabled.
- Hence IF only decides if INTR interrupt is enabled or disabled.

**$S_6$ : Indicates if 8086 is the bus master or not.**

If $S_6 = 1 \rightarrow$ 8086 is not the bus master.

If $S_6 = 0 \rightarrow$ 8086 is the bus master.

- Bus master is the device who "controls" the system bus.
- Generally 8086 is the bus master.
- It performs operations with memory and I/O using the system bus.
- But there are other candidates who can also take control of the system from 8086 and become the bus master.
- It could be a DMA controller like 8237/ 8257.
- In maximum mode it could also be one of the other processors like 8087 NDP or 8089 I/O Processor.

**$S_7$ : Reserved for further development**

### 1.7.11 $\overline{RD}$

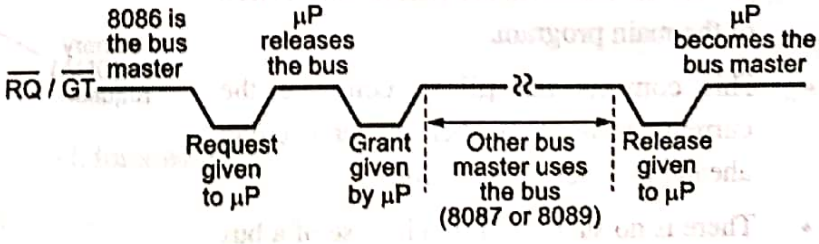
- It is an active low output signal.
- It becomes active (goes low, becomes "0") during a read operation.
- **$\overline{RD}$ along with $\overline{WR}$ and M/$\overline{IO}$** are decoded together to decide if the μP is performing a memory read, memory write, I/O read or an I/O write operation.
- It is interesting to note that this signal only operates in minimum mode.
- In Maximum mode this signal gets disabled.

### 1.7.12 MIN Mode / Max Mode Signals (10M question — Important)

- There are several signals that change their behavior from minimum mode to maximum mode.
- 8086 μP checks the value of MN/$\overline{MX}$ and based on that decides if it is working in minimum or maximum mode. Effectively the behavior and the function of these pins changes accordingly.

### 1.7.13 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 control of the system bus.
- μPin forms DMAC that the bus has been released by giving HLDA signal.
- In **Maximum** Mode it carries the bi-directional **$\overline{RQ_0}$ / $\overline{GT_0}$** signal (Request/Grant).
- The **$\overline{RQ_0}$ / $\overline{GT_0}$** signal is used by t = other processors like 8087 to control of the system bus from 8086.
- By default 8086 is the bus master.
- At that time the **$\overline{RQ_0}$ / $\overline{GT_0}$** signal is inactive and hence it is "high".
- If 8087 needs to become the bus master, it sends a low pulse on **$\overline{RQ_0}$ / $\overline{GT_0}$**
- In response 8086 releases control of the system bus and gives another low pulse to 8087 on the same **$\overline{RQ_0}$ / $\overline{GT_0}$** pin, to indicate that 8087 has become the bus master.
- Now 8087 performs its required operation on the system bus.
- Finally 8087 sends a low pulse on **$\overline{RQ_0}$ / $\overline{GT_0}$** signal (casually called "Release") to inform 8086 that the bus operation is complete and hence 8086 once again becomes the bus master.




(1A10)**Fig. 1.7.2 : $\overline{RQ}$ / $\overline{GT}$**

☞ **#VIVA**

- There are two Request/ Grant pins : $\overline{RQ_0}$ / $\overline{GT_0}$ and $\overline{RQ_1}$ / $\overline{GT_1}$ . This is because two processors can be connected directly to 8086: 8087 and 8089.
- We can connect any of the two processors to any of the two Request/ Grant lines.
- In the event that both processors request for the bus simultaneously, the processor that is requesting using the line $\overline{RQ_0}$ / $\overline{GT_0}$ will get **higher priority**.

## ☞ 1.7.14 HLDA – $\overline{RQ_1}$ / $\overline{GT_1}$

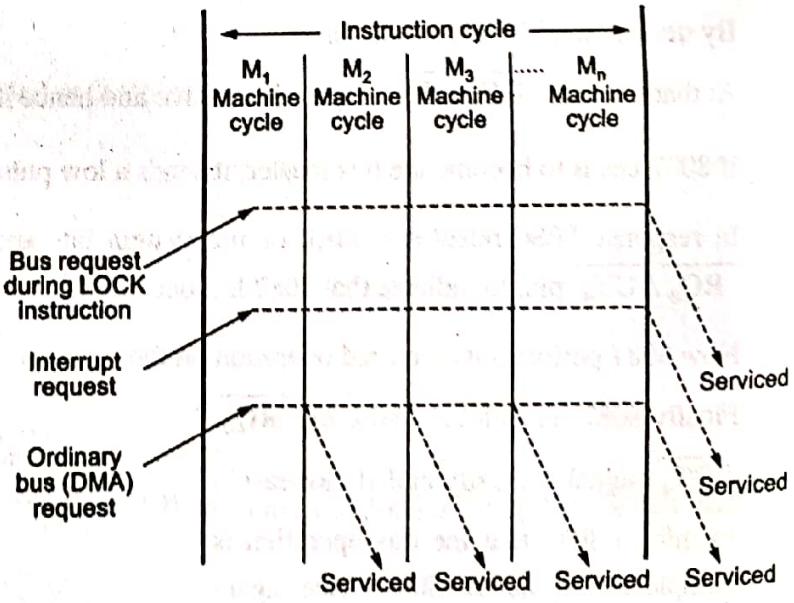
Explained along with the previous pins already.

## ☞ 1.7.15 $\overline{WR}$ – $\overline{LOCK}$

- 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

- An instruction requires one or more machine cycles for its execution.
- A machine cycle in turn has 4 T-states.
- While the μP is performing an instruction, it may get an interrupt request or a bus request.
- If an interrupt occurs, the μP can only service it after finishing the current instruction.
- This is because, after completing the ISR μP has to return to the NEXT instruction of the main program.
- This compels the μP to complete the current instruction before proceeding ahead to service an interrupt.
- There is no such condition in case of a bus request.




(1A11)**Fig. 1.7.3 : Lock and $\overline{LOCK}$**

- When µP gets a bus request, it simply has to release control of the bus.
- µP now enters a state called "Hold" state.
- Whenever µP gets back the bus it can resume from there on.
- It doesn't have to start from the NEXT instruction.
- Hence, whenever µP gets a bus request, it simply releases the bus immediately after completing the current machine cycle, not the whole instruction.
- This is shown the given diagram.
- Now consider an instruction which is very critical in our program.
- For some reason, we do not wish anything to affect the µP during the instruction.
- We are not worried about interrupts as even if an interrupt occurs µP will only service it after finishing the current instruction.
- But a bus request may take away the bus in the middle of an instruction and that may have an adverse effect on our program.
- To avoid this we write the LOCK prefix, next to the instruction.
  E.g.:: **LOCK MOV CX, [2000H]**
- Now, 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 $\overline{\text{LOCK}}$ signal low throughout the instruction to indicate that it is performing an instruction with LOCK prefix.

☞ **#VIVA**

$\overline{\text{LOCK}}$ signal is given to 8289 Bus Arbiter in Loosely Coupled Systems, to prevent 8289 from releasing the system bus to other bus masters.

### 1.7.16 $\overline{\text{DEN}}$ – $\overline{\text{S0}}$

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

### 1.7.17 DT/$\overline{\text{R}}$ – $\overline{\text{S}_1}$

- In **Minimum** Mode it carries the **DT/** $\overline{\text{R}}$ signal
- This signal goes **low** for a **Read** operation and high for a write operation.
- In **Maximum** Mode it carries the $\overline{\text{S}_1}$ signal.

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

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

### 1.7.18 M/ $\overline{IO}$ – $\overline{S_2}$

- 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.
- In minimum mode, M/ $\overline{IO}$, $\overline{RD}$ and $\overline{WR}$ are decoded to decide which of the basic operations are to be performed namely memory read, memory write, I/O read, I/O write. This is shown in the following table :

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

- In Maximum mode: $\overline{S_2}$, $\overline{S_1}$ and $\overline{S_0}$ are decoded by the 8288 bus controller to decide which operation has to be performed and hence which control signal has to be generated

| $\overline{S_2}$ | $\overline{S_1}$ | $\overline{S_0}$ | Processor State | 8288 Active Output |
|---|---|---|---|---|
| 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 |

### 1.7.19 ALE – $QS_0$

- 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.**

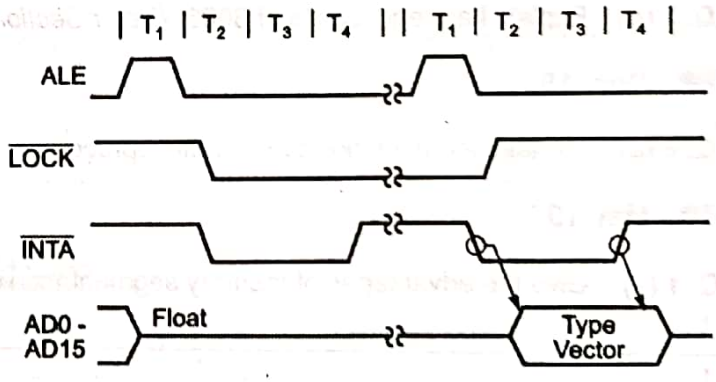
### 1.7.20 $\overline{INTA}$ – $QS_1$

- In **Minimum** Mode it carries the $\overline{INTA}$ 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.
- **In Maximum Mode, Bus Controller gives the $\overline{INTA}$ signal.**

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

### 1.7.21 Timing Diagram for 2 Back-to-back $\overline{INTA}$ Cycles

- As shown above there are two $\overline{INTA}$ cycles.
- Each $\overline{INTA}$ cycle is of 4 T-states
- In the 1st $\overline{INTA}$ cycle, the interrupting device (8259) **starts preparing** the vector number "N".
- In the 2nd $\overline{INTA}$ cycle, 8259 **sends the vector number (Type Number)** "N", to the microprocessor, through the multiplexed address data bus.




(1A12)**Fig. 1.7.4 : Timing Diagram for 2 Back-to-back $\overline{INTA}$ Cycles**

- 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.**
- $\overline{LOCK}$ signal is **held low between** the two $\overline{INTA}$ cycles, **so that the bus is not released** in the process.

## 1.8 University Questions and Answers

**Dec.15**

**Q.3 (b)** What is memory segmentation ? State Advantages of memory segmentation. *(Refer section 1.5)* **(5 Marks)**

**Q. 6 (b)** Write short note on generation of RESET Signals in 8086 based system. *(Refer section 1.7.2)* **(5 Marks)**

**May 16**

**Q. 2 (a)** Explain memory segmentation with PROS & CONS. *(Refer sections 1.5, 1.5.1 and 1.5.2)* **(8 Marks)**

**May 17**

**Q.1 (a)** What is memory segmentation ? State Advantages of memory segmentation. *(Refer sections 1.5, and 1.5.1)* **(5 Marks)**

**Dec. 17**

**Q. 1 (c)** Explain power on reset circuit used in 8086 system. *(Refer section 1.7.2)* **(5 Marks)**

**Q. 6 (c)** Explain memory segmentation of 8086. *(Refer section 1.5)* **(5 Marks)**

**May 18**

**Q. 1 (a)** Explain Memory banks for 8086 Processor. *(Refer Section 1.6)* **(5 Marks)**

**Q. 1 (d)** Explain flag register bits of 8086 *(Refer Section 1.4.2)* **(5 Marks)**

**Dec. 18**

**Q. 6 (a)** Explain segmentation of 8086 microprocessor. Give its advantages. *(Refer Section 1.5)* **(10 Marks)**

**May 19**

**Q. 1 (a)** Give the advantages of memory segmentation of 8086 microprocessor. *(Refer Section 1.5)* **(5 marks)**

*Chapter ends...*


# 8086 Assembly Language

✓ **Syllabus Topic : Addressing Modes**

## 2.1 8086 | Addressing Modes

**Q.** Explain different addressing modes of 8086 microprocessor.

**(Dec. 16, Dec. 17, 5 Marks, Dec. 18, May 19, 10 Marks)**

- **Addressing modes is the manner in which an operand is given in an instruction.**
- When we write an instruction, we usually intend to perform an operation on some operand.
- How we tell the operand to the μP, is what is called Addressing Mode.
- Suppose we want to add the operand 25H to BL register.
- Here our operand value is 25H.
- There are several ways of telling the μP that the operand is 25H.
- We may simply write ADD BL, 25H, thereby giving the operand in the instruction itself.
- Or we may put 25H in a register like CL and then say ADD BL, CL.
- Or we may store the operand 25H in the memory at some location like 2000H and give its address in the instruction like ADD BL, [2000H]; and so on…

### ☞ Important tip for beginners

As a student it is extremely important for you to understand the various addressing modes before you start learning the instructions. These addressing modes essentially help you understand the language of the instruction. Over the years of teaching this beautiful subject I have heard a common grouse from students that programming is tough. ***No! It isn't.*** On the contrary it is one of the aspects in this subject where the true creativity and problem solving capability of the student is exhibited.

The reason why people rate programming to be tough is because of the wrong approach taken. Before you start programming you must be thorough with the working and the real life use of the various instructions. And before you start learning instructions, you must first understand how to read them, which means understand addressing modes. Quite frankly, your journey of learning and from there on mastering Assembly Language programming actually begins from this topic.

**▶ 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 are based on various categories.

☞ **Data Memory Addressing Modes**

1. Immediate addressing mode
2. Register addressing mode
3. Direct addressing mode
4. Indirect addressing mode
   - 4.1 Register indirect addressing mode
   - 4.2 Register relative addressing mode
   - 4.3 Base indexed addressing mode
   - 4.4 Base relative plus indexed addressing mode
5. Implied addressing mode

☞ **I/O Addressing modes**

1. Direct addressing mode (Fixed Port addressing mode)
2. Indirect addressing mode (Variable port addressing mode)

☞ **Program Memory Addressing modes**

(Not to be included in the answer unless asked specifically)

1. Intra segment direct addressing mode
2. Inter segment direct addressing mode
3. Intra segment indirect addressing mode
4. Inter segment indirect addressing mode

## 2.2 Data Memory Addressing modes

### 2.2.1 Immediate Addressing Mode

- Here the **operand** is specified in the **instruction** itself.
- We give the operand (that means the data) in the instruction itself.
- As soon as µP fetches the instruction, it gets the data "immediately" within the instruction.
- Hence the name Immediate addressing mode.
- It is mainly use when initialize values in the beginning of a program.

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

- Similarly, instead of MOV, we could also use other instructions like ADD, SUB etc.

```
Eg. :  ADD CL, 12H      ; CL ← CL + 12H
       SUB CL, 12H      ; CL ← CL – 12H
```

- The important point here is not whether we are Moving or Adding or Subtracting.
- Your focus should actually be in understanding how we provide the operand in the instruction.
- So that your attention doesn't get diverted, we will use MOV for all examples.
- In the exam you may use ADD or SUB or anything else once you have thoroughly studied the full instruction set in the next chapter.

### 2.2.2 Register Addressing Mode

- Here the **operand** is given using a **register**.
- The data that we need to operate on, is present in some register.
- We write the name of the register in the instruction.
- µP will access the data from the register and perform the operation.
- It is used once values are already present in some register and we want to operate on them.

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

### 2.2.3 Direct Addressing Mode

- Here the **address** of the operand is directly specified **in the instruction.**
- The data that we need to work on is present in some memory location.
- We provide the address of the data in the instruction.
- First the µP fetches the instruction.
- In the instruction µP gets the address.
- Then µP goes to this address in the memory to perform the operation.
- Do keep in mind that programmer never gives the physical address.
- We write the offset address in the instruction.
- Here **only** the **offset address is specified**, the segment being indicated by the instruction.

**Eg. :** **MOV CL, [2000H]** ; Moves data from location 2000H in the data
; segment into CL
; The physical address is calculated as
; **DS * $10_H$ + 2000**
; Assume DS = 5000H
; ∴P A= 50000 + 2000 = 52000H
; ∴CL ← [52000H]

Data Segment

| Offset Addresses | Data | |
|---|---|---|
| 0000H | | |
| | | |
| 2000H | **78H** | → **CL gets 78H** |
| 2001H | **56H** | |
| 2002H | **34H** | |
| 2003H | **12H** | |
| | | |
| FFFFH | | |

```
Eg. :  MOV CX, [2000H]   ; Moves data from location 2000H and 2001H
                         ; in the data segment into CX i.e. CL and CH registers.
                         ; CL ← DS:[2000H] and CH ← DS: [2001H] following the
                         ; principle of "lower byte ~ lower address" also called the
                         ; "Little Endian Rule"
```

### 2.2.4 Indirect Addressing Modes

There are 4 types of indirect addressing modes :

1. Register Indirect Addressing Mode
2. Register Relative Addressing Mode
3. Base Indexed Addressing Mode
4. Base relative Plus Indexed Addressing Mode

#### 1. Register Indirect Addressing Mode

- Here the **address** of the operand is **given using a register.**
- The data that we need to operate on is present in some memory location.
- The address of that memory location, is present in a register.
- We provide the name of the register in the instruction.
- First the µP fetches the instruction.
- From the instruction µP identifies the register.
- From the register µP gets the address of the operand.
- Then µP goes to this address in the memory to perform the operation.
- As always, the register will provide the Offset address.

```
Eg. :  MOV CL, [BX]      ; CL gets 8-bit data from a memory location whose
                         ; 16-bit offset address is given by BX, in the Data Segment.
                         ; Hence CL ← DS: [BX]
                         ; The physical address is calculated as DS * 10H + BX
                         ; Assume DS = 4000H and BX is 2250
                         ; ∴P A= 40000 + 2250 = 42250H ∴CL ← [42250H]
```

Data Segment

| Offset Addresses | Data | |
|---|---|---|
| 0000H | | |
| | | |
| (BX) 2250H | **78H** | → **CL gets 78H** |
| 2251H | **56H** | |
| 2252H | **34H** | |
| 2253H | **12H** | |
| | | |
| FFFFH | | |

**#VIVAQuestion**

**What is the advantage of Indirect Addressing mode?**

**Answer**

It is used to access a Series of locations in a Loop. As the address is given using a register, we can increment/ decrement the register in the loop so that it points to a different location in every iteration of the loop.

**Eg. :** **MOV CX, [BX]** ; CX gets 16-bit data from two memory locations whose
; 16-bit offset address is given by BX and BX+1, in the Data Segment.
; Hence CL ← DS: [BX]; CH ← DS: [BX+1]

### ☞ 2. Register Relative Addressing Mode

Here the address of the operand is given as a sum of a register and a displacement

**Eg. :** **MOV CL, [BX+4]** ; Moves a byte from the location pointed by BX + 4 in
; Data Segment to CL.
; Physical Address: DS * $10_H$ + BX + 4H
; If DS is 4000H and BX is 2250H the physical address is 42254H.

The same instruction can also be written like this…

**Eg. :** **MOV CL, 04H[BX]** ; Moves a byte from the location pointed by BX+04H in
; the Data Segment to CL.
; Physical Address: DS * $10_H$ + BX + 4H
; If DS is 4000H and BX is 2250H the physical address is 42254H.

### ☞ 3. Base Indexed Addressing Mode

Here, operand address is given as a sum of **Base register plus** an **Index** register.

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

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

**#Viva Question :**

**Can we use ANY two registers?**

**Answer :**

NO. It must be one base register and one index register. The base registers are BX and BX. The Index Registers are Si and DI. Hence there are only four possible combinations: BX+SI, BX+DI, BP+SI, BP+DI.

### ☞ 4. Base relative Plus Indexed Addressing Mode

Here, operand address is given as a sum of **Base register plus** an **Index** register plus a 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

## 2.2.5 Implied Addressing Mode

In this addressing mode the operands are implied and are hence not specified in the instruction.

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

**Eg. : CLD** ; Clears the Direction Flag.

## 2.3 I/O addressing Modes of 8086

*(5 Marks – Important Question, can be asked independently)*

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

### 2.3.1 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 port addresses.**
- 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.**

### 2.3.2 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 port addresses.**
- 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.**

## 2.4 Addressing Modes for Program Memory

*(Optional – Write this in your exam answer only if specifically asked)*

- This addressing mode is required for instructions that **cause a branch** in the program.
- Normally a program is executed in a sequential manner.
- This happens by constantly incrementing IP after every instruction is fetched.
- In case of a branch, the program wants to go elsewhere, and not to the immediately next instruction.
- This simply means the value of IP needs to be changed.
- If the branch is within the same Code Segment, then only the offset address that means IP will change.
- The segment address that means CS register will remain the same.
- This is called an Intra-segment branch, also called a Near branch.
- If the branch is from one Code Segment to another, then the offset address as well as the segment address, both will change. Hence CS will get a new segment address and IP will get a new offset address.
- This is called an Intra-segment branch, also called a Near branch.

### 2.4.1 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
```

### 2.4.2 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_1ENDS
Code_2      SEGMENT
                .
                .
NextSeg :   .....
                .
Code_2          ENDS
```

### 2.4.3 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. :** **JMPWORD PTR [BX]** ; IP ← {DS:[BX], DS:[BX+1]}

### 2.4.4 Inter Segment Indirect Addressing Mode

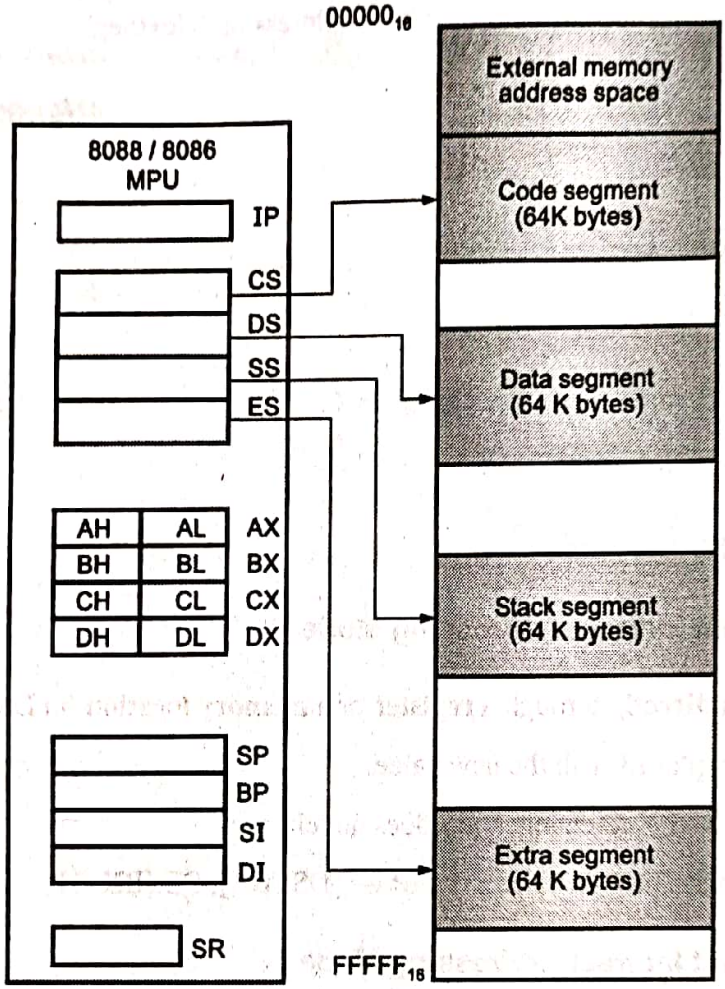
- The new Branch location is **specified indirectly** through a **register** or a **memory location** (in DS only).
- Both CS and IP get new values, as this is an inter-segment branch.

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

✓ Syllabus Topic : Programmer's Model

## 2.5 8086 Software Model

**Q. Explain programming model of 8086. (May 16, 5 Marks)**




(1A13)**Fig. 2.5.1 : Programmer's Model of 8086**

- Software Model, also called **Programmers model,** Means all the registers available to the programmer. If Programmers Model is asked in the exam, 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.**
- The explanation of these registers has already been included in architecture.

✓ Syllabus Topic : Instruction Set

## 2.6 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 and 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

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.

✓ **Syllabus Topic : Data Transfer Instructions**

## 2.7 Data Transfer Instructions

**Q.** Explain following instructions : DAA ,AAA, XLAT, LAHF (May 18, 10 Marks)

### 2.7.1 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, in the same instruction.
- Please note that the data will be "copied" and not cut.
- This means the data will remain in the source and will also get copied into the destination.

```
E.g. :  MOV CX, 0037H       ; CX gets the 16-bit data 0037H
                            ; CX ← 0037H
        MOV BL, [4000H]     ; BL gets 8-bit data from offset address 4000h in Data Segment
                            ; BL ← DS:[4000H]
        MOV DL, [BX]        ; DL gets 8-bit data from a location pointed by BX in Data Segment
                            ; DL ← DS:[BX]
        MOV DS, BX          ; DS gets the 16-bit value of BX
```

**Note :** A segment register like (DS, SS, ES) can only be initialized using a general purpose register like AX, BX, CX or DX as shown above.

**Segment Overriding**

- Most instructions by default operates on a particular segment like Data Segment

```
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
```

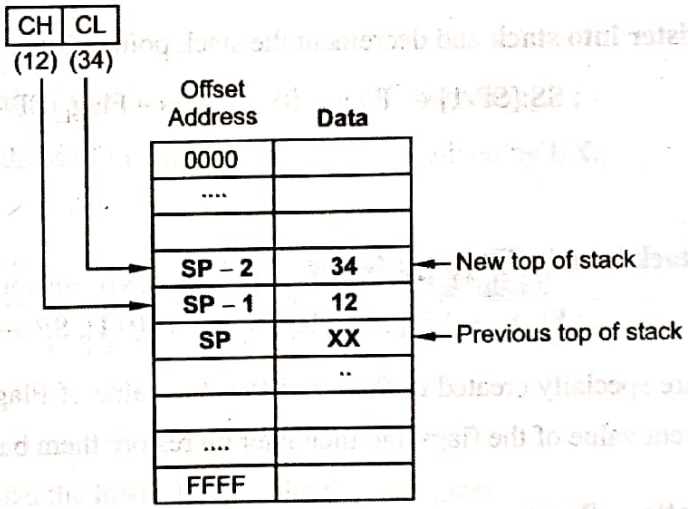
## 2.7.2 PUSH source

- **Push** the 16-bit **source** data **into** the top of stack.
- SP always points to the location where the current top of stack is stored.
- Hence SP needs to be decremented to store this new data as the top of stack.
- Since we are pushing 16-bit data, it will require two locations.
- Hence SP is decremented by two in the entire operation.
- Remember that all stack operations (Push and Pop) are 16-bit operations.
- **Source :** Register, Memory Location
- **Destination :** Top of Stack

**Eg. :** **PUSH CX** ; SS:[SP-1] ← CH,

; SS:[SP-2] ← CL

; SP ← SP – 2




(1A14)**Fig. 2.7.1 : PUSH operation**

**Note :** You don't need to mug up as to which register goes to which address.

Remember the Little Endian rule, "Lower Byte- Lower Address; Higher Byte – Higher Address"!

Between CH and CL, CH is the higher byte, CL is the lower byte.

SP-1 is the higher address compared to SP-2.

Hence the higher byte CH goes to the higher address SP-1 and CL goes to SP-2.

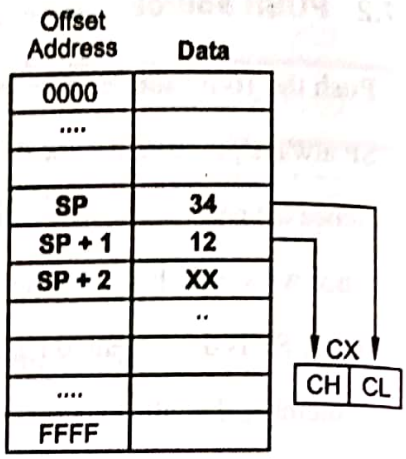
### 2.7.3 POP destination

- This instruction will POP 16-bit data from the top of stack and store it into the destination.
- **Source :** Top of Stack
- **Destination :** Register (Except CS), Memory Location

**Eg. :** **POP CX** ; CH ← SS:[SP]

; CL ← SS:[SP+1]

; SP ← SP + 2




(1A15)**Fig. 2.7.2 : POP operation**

**Note :** Between CH and CL, CH is the higher byte, CL is the lower byte.

SP is the lower address compared to SP+1.

Hence the higher byte CH gets its value from the higher address SP+1 and CL gets from SP.

### 2.7.4 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

### 2.7.5 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

These two instructions are specially created to Push and Pop the value of Flag register. They are used when we want to store the current value of the flags and then later on restore them back to the same values.

### 2.7.6 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

### 2.7.7 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.

### 2.7.8 LAHF

**Q.** Explain the following instructions in 8086 : LAHF **(Dec. 15, 2 Marks)**

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

### 2.7.9 SAHF

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

### 2.7.10 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.

### 2.7.11 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]}

### 2.7.12 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]}

### 2.7.13 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.

### 2.7.14 OUT destination port, source register

- Loads the destination I/O port with the contents of the source register.
- **Destination :** 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.**
- **Source :** It has to be some form of "A" register, in which we will get data from the I/O device.
- If we are sending 8-bit data, it will be AL or AH register.
- If we are sending 16-bit data, it will be AX 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

☞ **#VIVA Questions**

**Question :** **Send Data 25H to output port address 80H**

**Solution :**
```
MOV AL, 25H
OUT 80H, AL
```

**Question :** **Send Data 1234H to output port address 80H**

**Solution :**
```
MOV AX, 1234H
OUT 80H, AX
```

**Question :** **Send Data 25H to output port address 8000H**

**Solution :**
```
MOV AL, 25H
MOV DX, 8000H
OUT DX, AL
```

**Question :** **Send Data 1234H to output port address 8000H**

**Solution :**
```
MOV AX, 1234H
MOV DX, 8000H
OUT DX, AX
```

✓ **Syllabus Topic : Arithmetic Instructions**

## 2.8 Arithmetic Instructions

**Q.** Explain following instructions : DAA ,AAA, XLAT, LAHF **(May 18, 10 Marks)**

### 2.8.1 ADD destination, source

- This is a basic addition instruction.
- It will add the source data to the destination.
- The result will be stored in the destination.
- It can be used to perform 8-bit as well as 16-bit addition.
- If we add 8-bit numbers the answer will be either 8 bit or 9 bit.
- The $9^{th}$ bit is the carry.
- The 8-bits of the result will be stored in the destination.
- The $9^{th}$ bit will be stored in the carry flag.
- Likewise for a 16-bit addition.
- **Source :** Register, Memory Location, Immediate Number

- **Destination :** Register

**Eg. :** **ADD AL, 25H** ; AL ← AL + 25H

**ADD BL, CL** ; BL ← BL + CL

### 2.8.2 ADD destination, source

- This is add with carry.
- It will add the destination value, the source value and the value of Carry flag.
- The result will be stored in the destination.
- The carry involved in the addition is basically the carry of the previous operation.
- **Source :** Register, Memory Location, Immediate Number
- **Destination :** Register

**Eg. :** **ADC AL, 25H** ; AL ← AL + 25H + Carry Flag

**ADC BL, CL** ; BL ← BL + CL + Carry Flag

**#VIVA Question**

**Question : When do we use ADC instruction ?**

**Solution :**

- ADC is used when we want to add two large numbers.
- Consider the case when we want to add 12FFH with 0001H giving a result 1300H.
- Both are 16-bit numbers.
- Of course we can add them directly using 16-bit addition, but for some reason if we need to add them as two independent 8-bit additions, then first we will add the lower bytes (FFH + 01H) and then we will add the higher bytes (12H+00H).
- While adding the lower bytes we do a simple ADD.
- It produces a sum 00H and a carry "1" that will be stored in the carry flag.
- To add the higher bytes we then use ADC.
- It will not only add 12H plus 00H but will also include the carry "1" in the addition.
- Hence the net result will be 1300H.

### 2.8.3 SUB destination, source

- This is a basic subtraction instruction.
- It will subtract the source data to the destination.
- The result will be stored in the destination.
- It can be used to perform 8-bit as well as 16-bit subtraction.

- Remember to check the carry flag even after subtraction.
- Why? To check for a borrow.
- If the source is greater than the destination, this subtraction will be performed taking a borrow.
- This will make carry flag "1".
- Likewise for a 16-bit subtraction.
- **Source :** Register, Memory Location, Immediate Number
- **Destination :** Register

**Eg. :** **SUB AL, 25H** ; AL ← AL – 25H

**SUB BL, CL** ; BL ← BL – CL

### 2.8.4 SBB destination, source

- This is subtract with borrow.
- Just like in addition we ADD the previous carry, in subtraction we SUBTRACT the previous borrow.
- It will do Source – Destination – Carry Flag (Borrow).
- The result will be stored in the destination.
- Likewise for a 16-bit subtraction.
- **Source :** Register, Memory Location, Immediate Number
- **Destination :** Register

**Eg. :** **SBB AL, 25H** ; AL ← AL – 25H – Carry Flag

**SBB BL, CL** ; BL ← BL – CL – Carry Flag

### 2.8.5 INC operand

- It will increment the operand.
- The result will be stored in the operand itself.
- It can be used to perform 8-bit as well as 16-bit increment.
- **Operand :** Register, Memory Location

**Eg. :** **INC BL** ; BL ← BL + 1

**INC BX** ; BX ← BX + 1

**#Viva Question**

**Question : What happens if We perform INC BL and BL was FFH.**

**Answer :**

- In such a case BL will roll over to 00H
- Contrary to common assumption, here Carry Flag will NOT BE AFFECTED.
- Instead Zero Flag will become "1" as the resultant value of BL has become 00H.

### 2.8.6 DEC operand

- It will decrement the operand.
- The rest is same as increment, no point repeating the text

**Eg. :** **DEC BL** ; BL ← BL-1

**DEC BX** ; BX ← BX-1

### 2.8.7 MUL operand

- This is an unsigned multiplication instruction.
- It will multiply the operand with the accumulator.
- The result will be stored in the accumulator.
- 8-bit accumulator is AL.
- 16-bit accumulator is AX.
- 32-bit accumulator is a combination of DX and AX where DX is higher and AX lower.
- **Operand :** Register, Memory Location

**Eg. :** **8 bit multiplication:**

**MUL BL** ; AX ← AL × BL

**16 bit multiplication:**

**MUL BX** ; DX.AX ← AX × BX

☞ **#Important Tip**

Do not check for "carry" in a multiplication program.8-bit multiplication can NEVER produce a 17bit result. Even in the worst case we are multiplying FFH × FFH. The result is FE01H and that's a 16-bit number.

☞ **#Important Tip**

- When we use a register as an operand like BL or BH it is obvious whether we are doing 8-bit or 16-bit multiplication. But when we use a memory operand like [2000H], it can an become ambiguous instruction.
- To solve the ambiguity we use BYTE PTR or WORD PTR.

**MUL BYTE PTR [2000H]** ; Multiply an 8-bit number stored at offset address 2000H with AL.

**MUL WORD PTR [2000H]** ; Multiply a16-bit number stored at offset address 2000H and 2001H with AX.

### 2.8.8 IMUL operand

- This is a signed multiplication instruction.
- It is used to multiply signed numbers.
- The rest is the same as MUL so no point in repeating the text.

- In the exam when you get a program to multiply two numbers, please check the question carefully.
- If it is asked for unsigned numbers, use MUL instruction. For signed numbers, use IMUL instruction.
- The rest of the program remains unchanged. Using the wrong instruction will give unexpected results.
- If nothing is mentioned in the question, make a suitable assumption and continue.

### 2.8.9 DIV divisor

- This is an unsigned division instruction.
- It will divide the accumulator by the operand (divisor).
- The result will be stored in the accumulator.
- **Operand :** Register, Memory Location

**Eg. : 16-bit ÷ 8-bit division:**

```
DIV BL          ; Will perform AX × BL
                ; AL gets the quotient, AH gets the remainder.
```

**Eg. : 32-bit ÷ 16-bit division:**

```
DIV BX          ; Will perform DX.AX ÷ BX
                ; AX gets the quotient, DX gets the remainder.
```

### 2.8.10 IDIV operand

- This is a signed division instruction.
- It is used to divide signed numbers.
- The rest is the same as DIV.

### 2.8.11 CBW

- This instruction converts a byte (8 bit) to a word (16 bit).
- It works on AL register and the result will be stored in AX.
- It will take the 8 bit value in AL and "sign extend" it into 16 bits and store result in AX.
- The MSB of AL will be extended into all bits of AH.
- Assume AL is 0100 0010.
- After CBW, AX will become **0000 0000** 0100 0010
- Assume AL is 1100 0010.
- After CBW, AX will become **1111 1111** 1100 0010
- Unsigned numbers are always extended with leading zeros.
- Signed numbers are extended by the value of their MSB.

☞ **#Important Tip**

CBW must be used ONLY for signed numbers.

### 2.8.12 CWD

- This instruction converts a word (16 bit) to a double word (32 bit).
- It works on AX register and the result will be stored in DX.AX.
- It will take the 16 bit value in AX and "sign extend" it into 31 bits and store result in DX.AX.
- The MSB of AX will be extended into all bits of DX.

### 2.8.13 CMP operand1, operand2

- This is a compare instruction.
- Comparison is done by subtraction.
- It will perform operand1 – operand2.
- The result will not be stored but the flags will be affected.
- To know the outcome of the comparison we must check the flags.

**Eg. :** **CMPBL, CL** ; BL – CL

| Condition | Carry Flag: CF | Zero Flag: ZF |
|---|---|---|
| BL > CL | 0 | 0 |
| BL = CL | 0 | 1 |
| BL < CL | 1 | 0 |

☞ **#Viva Question**

**Question : What is the difference between SUB and CMP**

**Answer :**

- Both perform subtraction.
- SUB will store the result and affect the flags.
- CMP will not store the result. It will ONLY affect the flags.

### 2.8.14 NEG operand

- This is a negate instruction.
- It will negate the operand and store the result at the operand itself.
- Negation means it will convert the operand into its own 2's complement.

**Eg. :** **NEGBL** ; BL ← 2's complement of BL

## 2.9 Decimal Adjust Instructions

**Q. Explain following instructions : DAA ,AAA, XLAT, LAHF (May 18, 10 Marks)**

### 2.9.1 DAA

- Decimal adjust after addition. It works only on AL register.
- It is used when we want to perform addition of two decimal numbers (BCD addition).
- We first enter the two decimal numbers. We add them using a normal ADD instruction.
- Then we perform DAA instruction.DAA will adjust the addition to appear as a decimal addition.
- The logic of DAA is as follows :
  If the lower nibble of AL is > 9 or Auxiliary carry flag is "1" then Add 06 to AL.
  If the higher nibble of AL is > 9 or Carry flag is "1" then Add 60 to AL.

**Assume**

AL = 14H and 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, $(14)_{10} + (28)_{10} = (42)_{10}$

There are umpteen examples of DAA covered in my videos at www.BharatAcharyaEducation.com

### 2.9.2 DAS

- Decimal adjust after subtraction. It works only on AL register.
- It is used when we want to perform subtraction of two decimal numbers (BCD subtraction).
- We first enter the two decimal numbers. We subtract them using a normal SUB instruction.
- Then we perform DAS instruction. DAS will adjust the subtraction to appear as a decimal subtraction.
- The logic of DAS is as follows :
  If the lower nibble of AL is > 9 or Auxiliary carry flag is "1" then subtract 06 from AL.
  If the higher nibble of AL is > 9 or Carry flag is "1" then subtract 60 from AL.

**Assume**

AL = 86H and 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, $(86)_{10} - (57)_{10} = (29)_{10}$

## 2.10 ASCII Adjust Instructions

**Q.** Explain following instructions : DAA ,AAA, XLAT, LAHF (May 18, 10 Marks)

### 2.10.1 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 value of 4.

CL = 0011 1000 ... ASCII value of 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.

### 2.10.2 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.

### 2.10.3 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.

### 2.10.4 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

✓ Syllabus Topic : Logical Instructions

## 2.11 Logic Instructions (Bit Manipulation Instructions)

### 2.11.1 AND destination, source

- This instruction performs a LOGIC AND operation.
- It will AND the source and the destination.
- The result will be stored in the destination.
- **Source :** Register, Memory Location, Immediate Number
- **Destination :** Register

**Eg. :** **AND AL, 25H** ; AL ← AL AND 25H

**AND BL, CL** ; BL ← BL AND CL

☞ **#Viva Question**

**Question : What is the use of AND instruction ?**

**Answer :** AND is used to make any bit ZERO.

**To make any bit ZERO: AND that bit with "0" and the remaining bits with "1"**

E.g. : Assume AL is 35H i.e. 0011 0101.

- We want to make its lower nibble "0000" that means AL should become 30H.
- To do this, we must AND AL register with the value F0H i.e. 1111 0000.

**Truth table of AND gate**

| Inputs | | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

The bits that get AND with "1" remain the same but the bits that gets AND with "0" become "0". Now AL will become 0011 0000.

### 2.11.2 OR destination, source

- This instruction performs a LOGIC OR operation.
- It will OR the source and the destination.
- The result will be stored in the destination.
- **Source :** Register, Memory Location, Immediate Number
- **Destination :** Register

**Eg. :** **OR AL, 25H** ; AL ← AL OR 25H

**OR BL, CL** ; BL ← BL OR CL

#### #Viva Question

**Question : What is the use of OR instruction?**

**Answer :** OR is used to make any bit ONE.

**To make any bit ONE:OR that bit with "1" and the remaining bits with "0"**

E.g. : Assume AL is 35H i.e. 0011 0101.

- We want to make its lower nibble "1111" that means AL should become 3FH.
- To do this, we must OR AL register with the value 0FH i.e. 0000 1111.

**Truth table of OR gate**

| Inputs | | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

- The bits that get OR with "0" remain the same but the bits that get OR with "1" become "1".
- Now AL will become 0011 1111.

### 2.11.3 XOR destination, source

- This instruction performs a LOGIC XOR operation.
- It will XOR the source and the destination.
- The result will be stored in the destination.
- **Source :** Register, Memory Location, Immediate Number
- **Destination :** Register

**Eg. :** **XOR AL, 25H** ; AL ← AL XOR 25H

**XOR BL, CL** ; BL ← BL XOR CL

☞ **#Viva Question**

**Question : What is the use of XOR instruction ?**

**Answer :** XOR is used to COMPLEMENT any bit.

**To COMPLEMENT any bit: XOR that bit with "1" and the remaining bits with "0"**

E.g. : Assume AL is 35H i.e. 0011 0101.

- We want to complement its lower nibble to "1010" that means AL should become 3AH.
- To do this, we must XOR AL register with the value 0FH i.e. 0000 1111.

**Truth table of XOR gate**

| Inputs | | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

- The bits that get XOR with "0" remain the same but the bits that get XOR with "1" get COMPLEMENTED.
- Now AL will become 0011 1010.

### 2.11.4 NOT operand

- This instruction performs a LOGICAL NOT operation.
- It will NOT the operand value, like an inverter.
- The result will be stored at the operand itself.
- **Operand :** Register, Memory Location

**Eg. : NOT AL** ; AL ← 1's complement of 25H

☞ **#Viva Question**

**Question : What is the use of NOT instruction?**

**Answer :**

- To produce derived gates such as NAND, NOR and XNOR.
- AND followed by NOT is a NAND.
- OR followed by NOT is a NOR.
- XOR followed by NOT is a XNOR.

### 2.11.5 TEST operand1, operand2

- This instruction performs a LOGICAL AND operation.
- It will AND operand1 with operand2.
- The result will not be stored but the flags will get affected.

**Eg. : TEST AL, BL** ; AL AND BL

**#Viva Question**

**Question : What is the use of TEST instruction ?**

**Answer :**

- TEST is used to check the value of a SEMAPHORE in a multiprocessor system to implement principle of mutual exclusion.
- For students of Mumbai University, your syllabus does not take you to the point where you can explore this use. Curiosity on the other hand has no limits.

## 2.12 Shift Instructions

### 2.12.1 SHL/SAL destination, count

- **This instruction 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.

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

**Assume**

**Before Operation: BL = 0011 0011 and CF = 1**


(1A16)**Fig. ** : SHL instruction**

**More examples**

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

**SAL BL, CL** ; Left-Shift BL bits CL (5) number of times.

### 2.12.2 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.

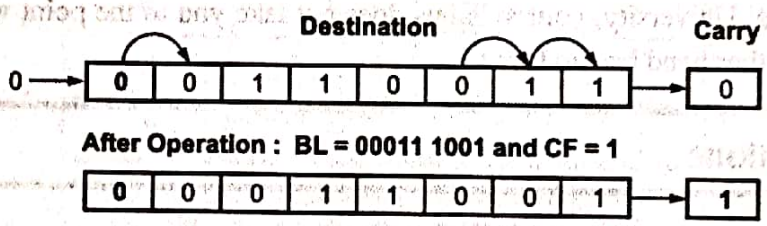
  If count is 1, it is directly specified in the instruction.

  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**




(1A17)**Fig. 2.12.1 : SHR instruction**

### 2.12.3 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.

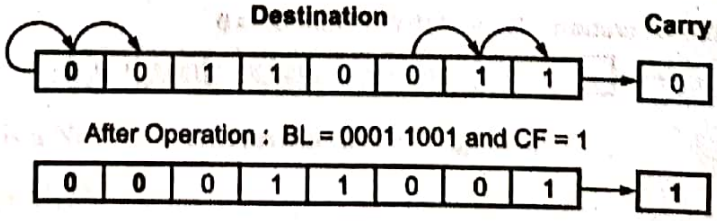
  If count > 1 it has to be given using CL register.

- **Destination :** Register, Memory Location

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

**Assume**

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




(1A18)**Fig. 2.12.2 : SAR instruction**

## 2.13 Rotate Instructions

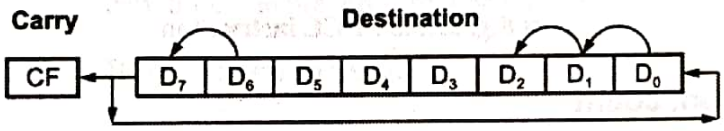
### 2.13.1 ROL destination, count

- **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.




(1A19)**Fig. 2.13.1 : ROL instruction**

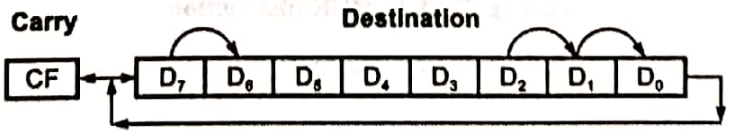
### 2.13.2 ROR destination, count

- **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.




(1A20)**Fig. 2.13.2 : ROR instruction**

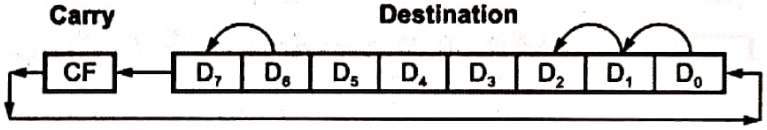
### 2.13.3 RCL destination, count

- **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.




(1A21)**Fig. 2.13.3 : RCL instruction**

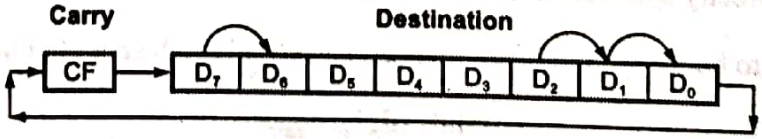
### 2.13.4 RCR destination, count

- **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.




(1A22)**Fig. 2.13.4 : RCR instruction**

## 2.14 Branch Instructions (Program Execution and Transfer)

- 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.**

### 2.14.1 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. :  JMPPrev      ; IP ← offset address of "Prev".
       JMPNext      ; 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.**

**JMPNextSeg** ; 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]}

### 2.14.2 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 |

### 2.14.3 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.**

### 2.14.4 RET – Return Instruction

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

**Intra-segment-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.

### 2.14.5 Differentiate between JMP Instruction and CALL Instruction

| No. | 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. |

### 2.14.6 Differentiate between Procedure (Function) and Macro

**Q.** Differentiate Procedure and macro with example. **(Dec. 18, May 19, 5 Marks)**

| No. | 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** |

## 2.15 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.**

### 2.15.1 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.
```

### 2.15.2 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.
```

### 2.15.3 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.
```

## 2.16 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**

### 2.16.1 STC

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

### 2.16.2 CLC

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

### 2.16.3 CMC

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

**For Direction Flag**

### 2.16.4 STD

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

### 2.16.5 CLD

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

**For Interrupt Enable Flag**

### 2.16.6 STI

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

### 2.16.7 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.
```

## 2.17 External Hardware Synchronization Instructions

### 2.17.1 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.

### 2.17.2 WAIT

- This instruction is used to synchronize 8086 with 8087 Co-Processor via the $\overline{\text{TEST}}$ input pin of 8086.

  Whenever 8087 is busy it puts a "1" on its BUSY o/p line connected to the $\overline{\text{TEST}}$ input of the µP.

- **The WAIT instruction makes the µP check the $\overline{\text{TEST}}$ pin.**
- If the µP checks the $\overline{\text{TEST}}$ pin and finds a "1" on it, 8086 understands that 8087 is busy and so it enters wait state. Here it does no processing.
- 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.

### 2.17.3 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 $\overline{\text{LOCK}}$ signal so that no other bus master takes control of the system bus.

### 2.17.4 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.

### 2.17.5 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.**

✓ Syllabus Topic : Control Instructions

## 2.18 Interrupt Control Instructions

### 2.18.1 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.

### 2.18.2 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).

### 2.18.3 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.

✓ Syllabus Topic : String Instructions

## 2.19 String Instructions of 8086

**Q.** Briefly explain string instructions of 8086. **(May 17, 5 Marks)**

- 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**.

### 2.19.1 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

### 2.19.2 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

### 2.19.3 STOS : STOSB/STOSW (Store String)

**Q.** Explain the following instructions in 8086 : STOSB **(Dec. 15, 2 Marks)**

- 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
```

### 2.19.4 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
```

### 2.19.5 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
```

## 2.19.6 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.**
- 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

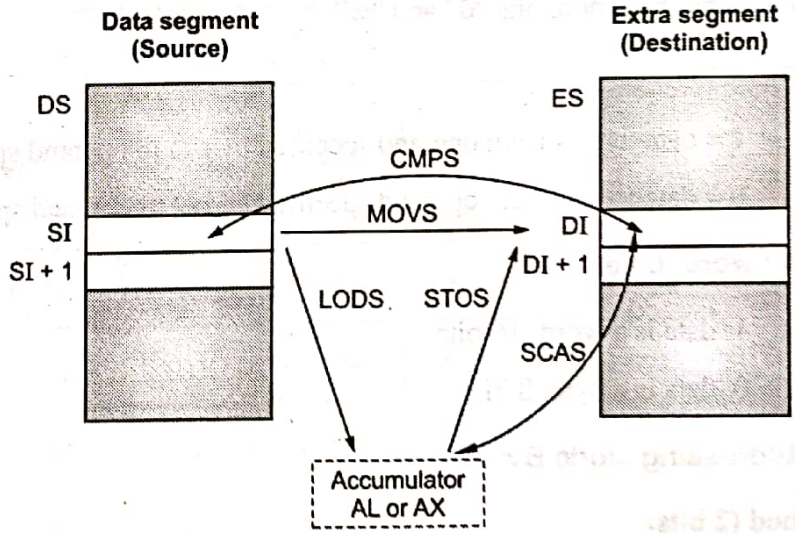
  to be executed 23H times (as CX = 23H) in auto incrementing mode (as DF is cleared).

## 2.19.7 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.

## 2.19.8 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.




(1A23)**Fig. 2.19.1 : String instructions**

**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.

## 2.20 8086 I Instruction Template / Format

- 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/w | | | 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 |

✓ **Syllabus Topic : Assembler Directives**

## 2.21 8086 | 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
```

✓ **Syllabus Topic : Programming based on DOS Interrupts (INT 21H)**

## ➲ 2.22 8086 | DOS Interrupt | INT 21H

- DOS provides **various interrupt services** that are used by the system programmer.
- The **most commonly used** interrupt is **INT 21H.**
- It **invokes inbuilt DOS functions** which can be used to perform various tasks.
- The most common tasks are reading a user input char from the screen, displaying result on the screen, exiting the program etc.
- One must remember that the same interrupt INT 21H can be invoked for performing many tasks.
- Then how do indicate which task we wish to perform?
- That is done by putting the correct value (parameter) in AH register before invoking INT 21H.
- The value in the AH register **selects the INT 21H function** which is required by the user.
- The most commonly used INT 21H functions are as shown :

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

### ☞ To Input a single character from the user

- You will need this to accept inputs from the user in lab Practicals.
- As an example, when you write a program to add two numbers, you may want to accept the numbers from the user instead of initializing them yourself. This is how you can accomplish that.

```
Code:   MOV AH, 01H
        INT 21H
```

- When you execute these two lines, a cursor will blink on the screen waiting for the user to enter a value.
- This is similar to "scanf" or "cin" if you are familiar with C/C++ programming.
- The value entered by the user will be stored in AL register.
- You must read the AL register to get the value.
- Keep in mind, you will get the ASCII value of the key that has been entered.

| Key | ASCII | |
|---|---|---|
| 0 | 30H | These are numerical keys 0…9 on your keyboard |
| 1 | 31H | |
| 2 | 32H | |
| 3 | 33H | |
| 4 | 34H | |
| 5 | 35H | |
| 6 | 36H | |
| 7 | 37H | |
| 8 | 38H | |
| 9 | 39H | |
| A | 41H | These are ASCII values of A…F in upper case. |
| B | 42H | |
| C | 43H | |
| D | 44H | |
| E | 45H | |
| F | 46H | |

- Consider the user enters 4
- What we will get is 34H, the ASCII value of 4
- We must SUB AL, 30H
- This eliminates the higher nibble "3" and gives the raw value 04H.
- Here we assumed that the user has entered a number between 0…9!
- What if the number entered is from A…F
- In that case the ASCII values are between 41…46H
- Assume the user enters A
- We get 41H, the ASCII value of A
- Now we must first subtract 7 and then subtract 30, which means essentially subtract 37H
- Note that 41H – 37H = 0AH, our desired value.
- So here is the general code to be used by the programmer while accepting an input from the user:

```
Code :   MOV   AH, 01H
         INT   21H
         CMP   AL, 39H
         JLE   Skip
         SUB   AL, 07H
Skip :   SUB   AL, 30H
```

- Now AL will get the desired input value.
- Lets take this logic ahead for a bigger number like an 8-bit number entered by the user E.g. : 45
- We will get 34 and 35 respectively for 4 and 5
- Applying the same logic we to both values individually, we will get 04H and 05H
- Now we must rotate the value (04H) four times using this code…

```
Code :   MOV   CL, 04H
         ROL   AL, CL
```

- This will make AL = 40H
- Now ADD the 2nd number 05H to AL
- This will finally give us the user entered number 45H!
- The same method must be used even to display a character.

✓ **Syllabus Topic : 8086 Programming**

## 2.23 8086 I Programming

### 2.23.1 Program to Add Two 16 Bit Numbers

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

**Algorithm (Logic)**

- Declare two variables for input numbers and two variables for sum and carry in the data segment
- Input first number in AX.
- Add the second number to AX.
- Check for Carry.
- Store sum.
- Store Carry.

**Program**

```
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
```

## 2.23.2 Program to Add Two 16 Bit BCD Numbers

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

### Algorithm (Logic)

- Declare two variables for input numbers and two variables for sum and carry in the data segment
- Input the two numbers in AX and BX.
- Add the lower bytes (AL and BL)
- Perform DAA to adjust the result to BCD.
- Store lower byte of result in CL.
- Take higher byte from AH to AL.
- ADD the higher bytes AH (taken in AL) and BH with carry.
- Perform DAA again.
- Store higher byte of sum in CX.
- Check for final carry.
- Store sum. Store Carry.

### Program

```
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     CL, AL
        MOV     AL, AH
        ADC     AL, BH
        DAA
        MOV     CH, AL
        JNC     Skip
        MOV     Carry, 01H
Skip:   MOV     Sum, CX
        INT3
Code    ENDS
END
```

### 2.23.3 Program to Add series of 10 Numbers

**Q.** 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.

#### Algorithm (Logic)

- Initialize segment registers.
- Initialize count of 10 in CX (000AH)
- Initialize registers for sum (AL) and carry (AH)
- Initialize SI as pointer to the series
- ADD every number pointed by SI to AL.
- If there is a carry, Increment AH.
- Repeat 10 times.
- The final 16-bit answer is in AL and AH.
- Store the result at the next two locations pointed by SI.

#### Program

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

• NOTES •

## 2.23.4 Block Transfer Program using string instructions

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

### Algorithm (Logic)

- Initialize segment registers DS and ES.
- Initialize offset registers SI and DI
- Initialize count of 10 in CX (000AH)
- Give increment direction using CLD in direction flag
- Transfer the data using the string instruction MOVSB
- Use REP prefix to transfer the whole block, CX number of times

### Program

```
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.** WAP to transfer a block of 10 bytes from Data Segment to Extra Segment

### Algorithm (Logic)

- Define Block1 in Data segment as the source block.
- Define Block2 in Extra segment as the destination block.
- Initialize segment registers DS and ES.
- Initialize offset registers SI and DI
- Initialize count of 10 in CX (000AH)
- Give increment direction using CLD in direction flag
- Transfer the data using the string instruction MOVSB
- Use REP prefix to transfer the whole block, CX number of times

### Program

```
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
```

## 2.23.5 Inverted block transfer Program

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

### Algorithm (Logic)

- Define Block1 in Data segment as the source block
- Define Block2 in Extra segment as the destination block
- Initialize segment registers DS and ES.
- Initialize offset registers SI and DI
- DI must be initialized with the end of Block2
- Initialize count of 10 in CX (000AH)
- Give increment direction using CLD in direction flag
- Perform LODSB to take the source data
- Give decrement direction using STD in direction flag
- Perform STOSB to take the source data
- Put the process in a loop using LOOP instruction

### Program

```
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
```

• NOTES •

### 2.23.6 Palindrome Program

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

#### Algorithm (Logic)

- Define Block1 in Data segment as the given string to be tested for being a palindrome
- Define Block2 in Extra segment as the inverted block
- Initialize segment registers DS and ES.
- Initialize offset registers SI and DI
- DI must be initialized with the end of Block2
- Initialize count of 10 in CX (000AH)
- Give increment direction using CLD in direction flag
- Perform LODSB to take the source data
- Give decrement direction using STD in direction flag
- Perform STOSB to take the source data
- Put the process in a loop using LOOP instruction
- This will perform inversion.
- Now compare Block1 and Block2
- For this, Initialize offset registers SI and DI
- Initialize count of 10 in CX (000AH)
- Give increment direction using CLD in direction flag
- Compare the blocks using the string instruction MOVSB
- Use REPZ prefix to Compare only as long as the two blocks are equal
- Check zero flag; If 1 that means the blocks are equal hence it is a palindrome.
- If zero flag is 0 that means the blocks are not equal hence it is not a palindrome.
- Declare the result using the variable Pal
- If Pal is 1 then it's a palindrome
- If Pal is 0 at the end of the program then it is not a palindrome

#### Program

```
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
```

## 2.23.7 16 bit multiplication Program

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

### Algorithm (Logic)

- Declare two 16-bit variables for input numbers: A and B
- Declare a 32bit variable for the result: Result
- Input 1st number in AX.
- Input 2nd number in BX.
- Multiply AX and BX
- Result is in DX and AX
- Make SI point to the Destination location (variable Result)
- Store AX at the location pointed by SI
- Store DX at the location pointed by SI+2

### Program

```
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
        MOV  BX, A
        MUL  BX
        LEA  SI, Result
        MOV [SI], AX
        MOV [SI+2], DX
        INT3
    Code ENDS
        END
```

## 2.23.8 Find Highest number Program

Q. WAP to find "highest" in a given series of 10 numbers beginning from location 20,000H.

Store the result immediately after the series.

Q. Write a program to find the largest number from an array. (May 19, 5 Marks)

### Algorithm (Logic)

- Initialise segment address DS=2000H
- Initialise offset address SI=0000H
- Initialise count CX = 000AH
- Initialise AL = 0 as the assumed highest number.
- Compare AL with every number pointed by SI
- If the number pointed by SI is greater than it means we have found a new highest number.
- Store that in AL, Else skip
- Increment the pointer, SI
- Put this in a loop, CX times
- When loop is over AL has the highest number
- Store result (AL) to the new location pointed by SI

### Program

```
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
```

## 2.23.9 Program to find negative numbers

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

### Algorithm (Logic)

- Initialise segment address DS=2000H
- Initialise offset address SI=0000H
- Initialise count CX = 000AH
- Initialise AH = 0 as the number of -ve numbers.
- Read every number pointed by SI into AL
- Rotate AL left so that the MSB comes into CF.
- If CF is 1 that means number is negative hence increment the negative counter AH, Else skip
- Increment the pointer, SI
- Put this in a loop, CX times
- When loop is over AH has no. of –ve numbers.
- Store result (AH) to the new location pointed by SI

### Program

```
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
```

## 2.23.10 Sorting Program

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

### Algorithm (Logic)

- Initialise segment address DS=2000H
- Initialise outer loop count of 9 in CH
- Initialise offset address SI=0000H
- Initialise inner loop count of 9 in CL
- Compare two adjacent numbers pointed by SI and SI+1
- If they are in correct order then leave them else interchange them
- Repeat the inner loop CL times
- Repeat the outer loop CH times
- Voila! The series is sorted.

### Program

```
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
```

## 2.23.11 Factorial Program

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

**Q.** Write a program to find the factorial of a number using procedure. **(Dec. 18, 5 Marks)**

### Algorithm (Logic)

- Initialise segment address DS=2000H
- Take data from offset address 4000H into CL
- Use this as the counter
- Initialise AX as 0001H
- Multiply CL with AL such that result is in AX
- Decrement CL
- Loop this by decrementing CL till CL is not zero
- AX has the final result
- Store AX at offset address 4001H and 4002H

### Program

```
Code    SEGMENT
        ASSUME  CS: Code
        MOV   AX, 2000H
        MOV   DS, AX
        MOV   CL, [4000H]
        MOV   AH, 00H
        MOV   AL, 01H
Back:   MUL   CL
        DEC   CL
        JNZ   Back
        MOV   [4001H], AX
        INT3
Code    ENDS
        END
```

### 2.23.12 Hex to Decimal (Binary to BCD) Program

**Q.** 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.

*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!*

```
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       ; Stor 34H at [4001H]
                              ; So finally [4002H] and [4001H] have the final
                              ; Result = 01 34
        INT3
Code    ENDS
        END
```

### 2.23.13 Decimal to Hexadecimal (BCD to Binary) Program

**Q.** 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!*

*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 × 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
```

### 2.23.14 Search "e" in a String Program

**Q.** 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
```

## 2.24 8086 | Passing Parameters to Subroutines

- **A Parameter is any value passed by the main program to the subroutine.**
- Passing parameters makes the subroutine **more flexible** and can **enhance the usage** of the subroutine.
- A subroutine that uses parameters can become a handy tool for solving various programming concerns.

  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.

### 2.24.1 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**

Cannot be used if there are multiple parameters as there are very few registers.

### 2.24.2 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.

### 2.24.3 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.

### 2.24.4 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.

```
Main:
        MOV     BX, 1234H       ; Put Parameter 1234H into BX.
        PUSH    BX              ; (parameter value 1234H Pushed into top of stack)
        Call    Sub             ; µP 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.

## 2.25 Mixed Language Programming using Assembly and C

**Q. Write a short note on mixed language programming. (Dec. 16, May 17, Dec. 18, May 19, 5 Marks)**

- 'C' generates an object code that is extremely fast and compact, but it is not as fast as the object code generated by a good programmer using assembly language.
- It is true that the time needed to write a program in assembly language is much more than the time taken in Higher Level Languages like C.
- However, there are special cases where a function is coded in assembly language to reduce execution time.
- E.g. : The Floating Point math package must be coded in assembly language as it is used frequently and its execution speed will have great effect on the overall speed of the program that uses it.
- There are also occasions when some hardware devices need exact timing and then it is necessary to write assembly level programs to meet such strict timing restrictions.

- In addition, certain instructions cannot be executed in Higher Level Languages like C.
- E.g. : C does not have an instruction for performing bit-wise rotation operation.
- Thus in spite of C being very powerful, routines must be written in assembly language to :
  - Increase the speed and efficiency of the routine.
  - Perform Machine specific functions not available in Microsoft C or in Turbo C.
  - Use third party routines.

### ☞ Combining C and assembly

There are 2 ways of combining C and Assembly language.

**Method 1 :**

- Here Built-In-Inline assembler is used to include assembly language routines in the C-program, without any need for a specific assembler.
- Such assembly language routines are called in-line assembly.
- They are compiled right along with C Routines rather than being assembled separately and then linked together using linker modules provided by the C Compiler.
- Turbo C (TC) has inline assembler.

**Method 2 :**

- There are times when programs written in one language have to call modules written in other languages.
- This is called as mixed language programming.

E.g. : when a particular sub-routine is available in a language, different from the language currently used in a program, or when algorithms are described in a different language, we need to use more than one language.

- Mixed language calls involve calling functions in separate modules.
- Instead of compiling all source programs using the same compiler, different compilers or assemblers are used as per the language used in the program.
- Microsoft C supports Mixed Language Programming.
- Therefore, it can combine assembly language routines in C as a separate language.
- C program calls assembly language routines that are separately assembled by MASM (MASM assembler) or TASM (Turbo assembler).
- These assembled modules are linked with the compiled C modules to get the combine executable file.

- Fig. 2.25.1 below shows Compile, Assemble, and link processes using C compiler, MASM Assembler and TLINK.

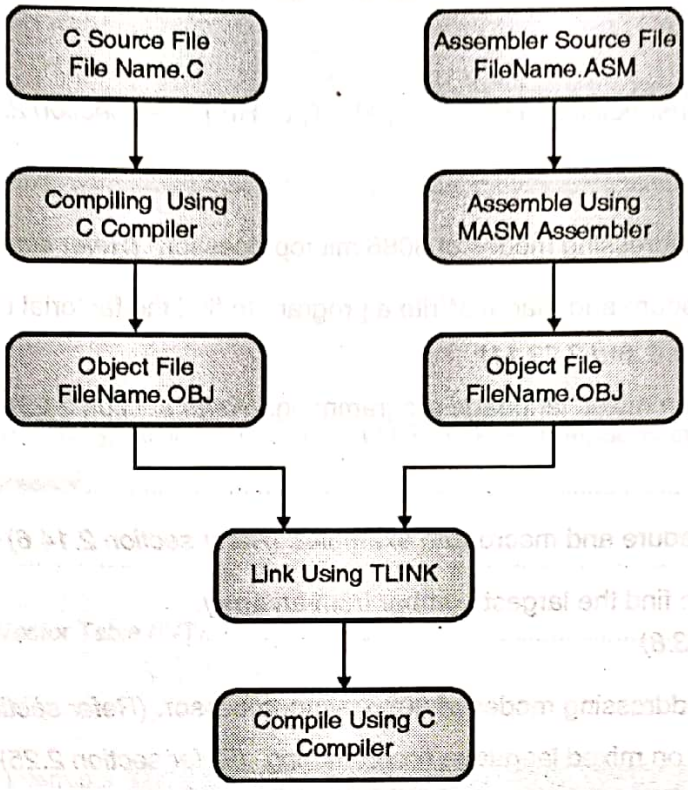



**Fig. 2.25.1**

## 2.26 University Questions and Answers

**Dec.15**

**Q. 1 (b)** Explain the following instructions in 8086 : LAHF and STOSB *(Refer section 2.7.8 and 2.19.3)* **(5 Marks)**

**May 16**

**Q. 1 (a)** Explain programming model of 8086. *(Refer section 2.5)* **(5 Marks)**

**Dec. 16**

**Q. 1(d)** Explain different addressing modes of 8086 microprocessor. *(Refer section 2.1)* **(5 Marks)**

**Q. 6(a)** Write a short note on mixed language programming. *(Refer section 2.25)* **(5 Marks)**

**May 17**

**Q. 1 (d)** Briefly explain string instructions of 8086. *(Refer section 2.19)* **(5 Marks)**

**Q. 6(a)** Write a short note on mixed language programming. *(Refer section 2.25)* **(5 Marks)**

➡ **Dec. 17**

**Q. 6(a)** Explain different addressing modes of 8086 microprocessor. *(Refer section 2.1)* (5 Marks)

➡ **May 18**

**Q. 4(a)** Explain following instructions : DAA ,AAA, XLAT, LAHF *(Refer section 2.7 to 2.10)* (10 Marks)

➡ **Dec. 18**

**Q. 6(b)** Explain different addressing modes of 8086 microprocessor. *(Refer section 2.1)* (10 Marks)

**Q. 5(a)** Differentiate procedure and macro .Write a program to find the factorial of a number using procedure. *(Refer section 2.14.6 and 2.23.11)* (10 Marks)

**Q. 1(b)** Write a short note on mixed language programming. *(Refer section 2.25)* (5 Marks)

➡ **May 19**

**Q. 1(b)** Differentiate Procedure and macro with example. *(Refer section 2.14.6)* (5 Marks)

**Q. 3(a)(ii)** Write a program to find the largest number from an array. *(Refer section 2.23.8)* (5 Marks)

**Q. 5(a)** Explain different addressing modes of 8086 microprocessor. *(Refer section 2.1)* (10 Marks)

**Q. 3(a)(i)** Write a short note on mixed language programming. *(Refer section 2.25)* (5 Marks)

---

*Chapter ends...*

❑❑❑


# 8086 Interrupts

✓ **Syllabus Topic : 8086 Interrupts, Types of Interrupts Interrupt Service Routine, Servicing of Interrupts by 8086 Microprocessor**

## 3.1 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 INT n 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)

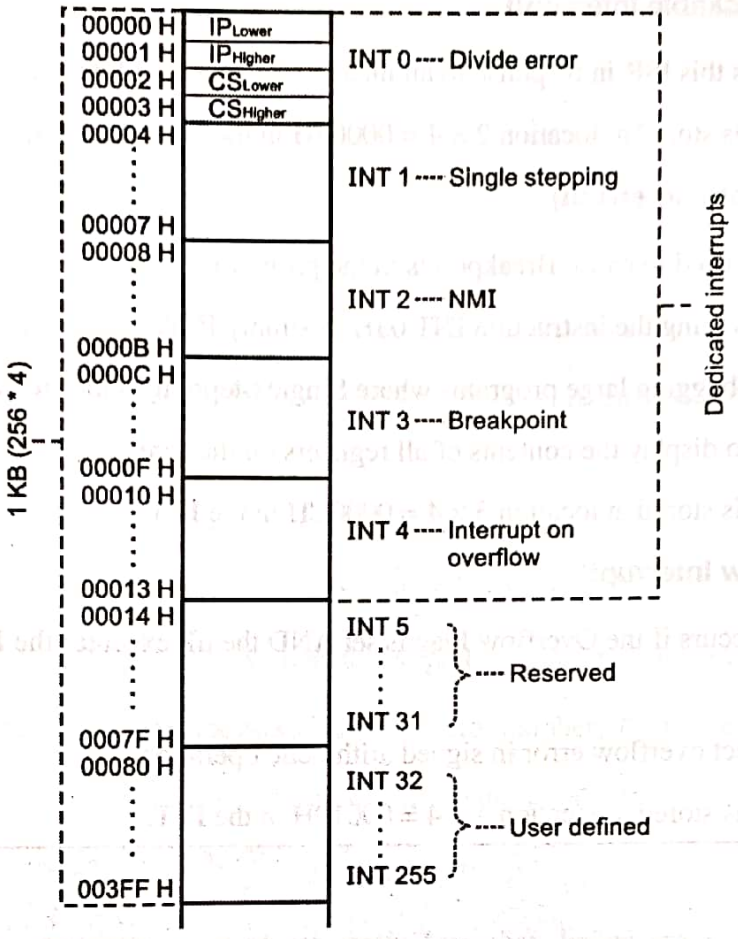
- 8086 is interrupted when some special conditions occur while executing certain instructions in the program.
- Eg: An error in division automatically causes the INT 0 interrupt.

✓ **Syllabus Topic : Interrupt Vector Table (IVT)**

## 3.2 Interrupt Vector Table (IVT)

**Q.** Explain : Types of interrupts **(May 17, May 18, Dec. 18, 10 Marks, May 19, 5 Marks)**

- 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.




(1A24) **Fig. 3.2.1 : IVT**

## 3.2.1 Dedicated Interrupts (INT 0 ... INT 4)

### 1. INT 0 (Divide Error)

- This interrupt occurs whenever there is division error i.e. when the result of a division is too large to be stored.
- This condition normally occurs when the divisor is very small as compared to the dividend or the divisor is zero.
- Its ISR address is stored at location $0 \times 4 = 00000H$ 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.
- Its ISR generally displays contents of all registers.
- Its ISR address is stored at location $1 \times 4 = 00004H$ 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 $2 \times 4 = 00008H$ 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 $3 \times 4 = 0000CH$ 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).
- It is used to detect overflow error in signed arithmetic operations.
- Its ISR address is stored at location $4 \times 4 = 00010H$ in the IVT.

**Please Note :**

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

## 3.2.2 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.

## 3.2.3 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 $n \times 4$ in the IVT.

### 3.2.4 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 $2 \times 4 = 00008H$ 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 $\overline{\text{INTA}}$ pulses.

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

2nd $\overline{\text{INTA}}$ 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.

## 3.3 How 8086 Responds to an Interrupt

**Steps Performed when an Interrupt "INT N" Occurs**

1. **The µP will PUSH Flag register into the Stack.**

   SS:[SP–1], SS:[SP–2] ← Flag

   SP ← SP – 2

2. **Clear IF and TF in the Flag register and thus disables INTR interrupt.**

   IF ← 0, TF ← 0

3. **PUSH CS into the Stack**

   SS : [SP–1], SS:[SP–2] ← CS

   SP ← SP – 2

4. **PUSH IP into the Stack**

   SS:[SP–1], SS:[SP–2] ← IP

   SP ← SP – 2

5. **Load new IP from the IVT**

   IP ← [N × 4], [N× 4 + 1]

6. **Load new CS from the IVT**

   IP ← [N × 4 + 2], [N × 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.

☞ **Steps performed to return to main program using IRET instruction**

1. The µP will restore IP from the stack

   IP ← SS:[SP], SS:[SP+1]

   SP ← SP + 2

2. The µP will restore CS from the stack

   CS ← SS:[SP], SS:[SP+1]

   SP ← SP + 2

3. The µP will restore FLAG register from the stack

   Flag ← SS:[SP], SS:[SP+1]

   SP ← SP + 2

## ➲ 3.4 Interrupt Priorities

| Divide Error, INT n, INTO | (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
2. Ability to interrupt another ISR

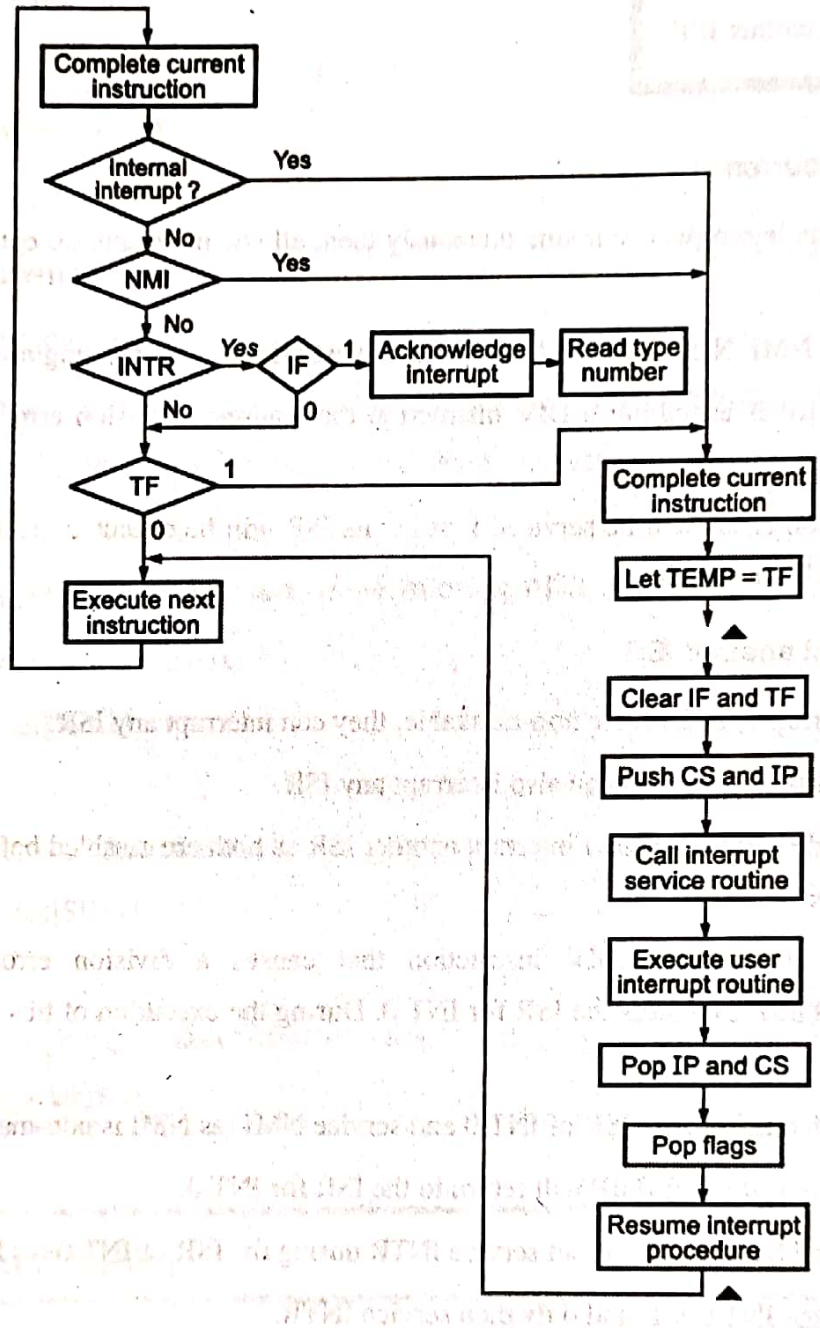
### ☞ 1. Simultaneous Occurrence

- 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.
- E.g. : 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.

### ☞ 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.

## 3.5 Interrupt Priority Flowchart




(1A25)**Fig. 3.5.1 : Flow chart for interrupt processing**

## 3.6 University Questions and Answers

**May 17**

**Q. 5(b)** Explain interrupt structure of 8086. *(Refer section 3.2)* **(10 Marks)**

**May 18**

**Q. 2(a)** Explain interrupt structure of 8086. *(Refer section 3.2)* **(10 Marks)**

**Dec. 18**

**Q. 5(b)** Explain interrupt structure of 8086. *(Refer section 3.2)* **(10 Marks)**

**May 19**

**Q. 6(b)(i)** Explain : Types of interrupts *(Refer section 3.2)* **(5 Marks)**

*Chapter ends...*

❑❑❑


# 8086 Circuit Configurations

✓ **Syllabus Topic : 8086 I Circuit Configurations**

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

## ➲ 4.1 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.

## ➲ 4.2 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.

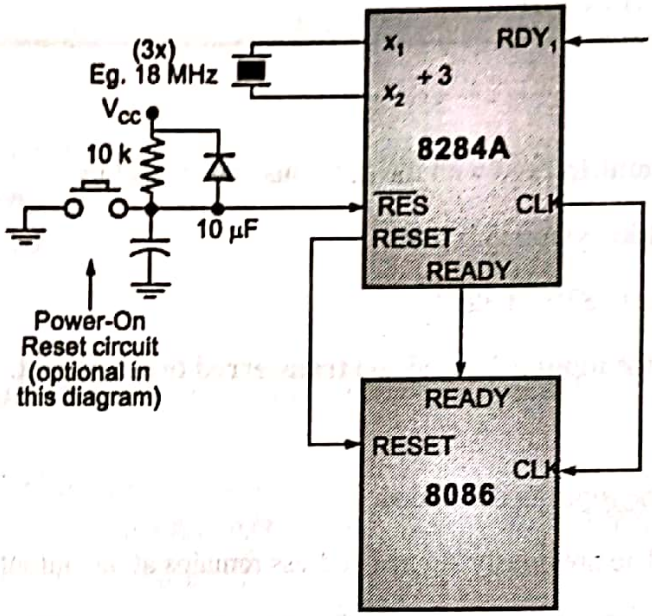
   If T = 1: data is transmitted.

   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.

✓ **Syllabus Topic : 8284 Clock Generator**

## 4.3 8284 – Clock Generator




(1A26)**Fig. 4.3.1 : 8284 Clock Generator**

1. 8284 is a Clock Generator IC.
2. It provides the C $\overline{\text{LOCK}}$ (CLK) signal, a train of pulses at a constant freq, to the entire circuit.
3. It synchronizes the READY ( $\overline{\text{RD}}$ 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 $X_1$, $X_2$** (Oscillator Clock Inputs)

      An **Oscillator** can be connected across the $X_1$, $X_2$ 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.
7. Clock Selection is done by the F/ $\overline{\text{C}}$ pin.

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

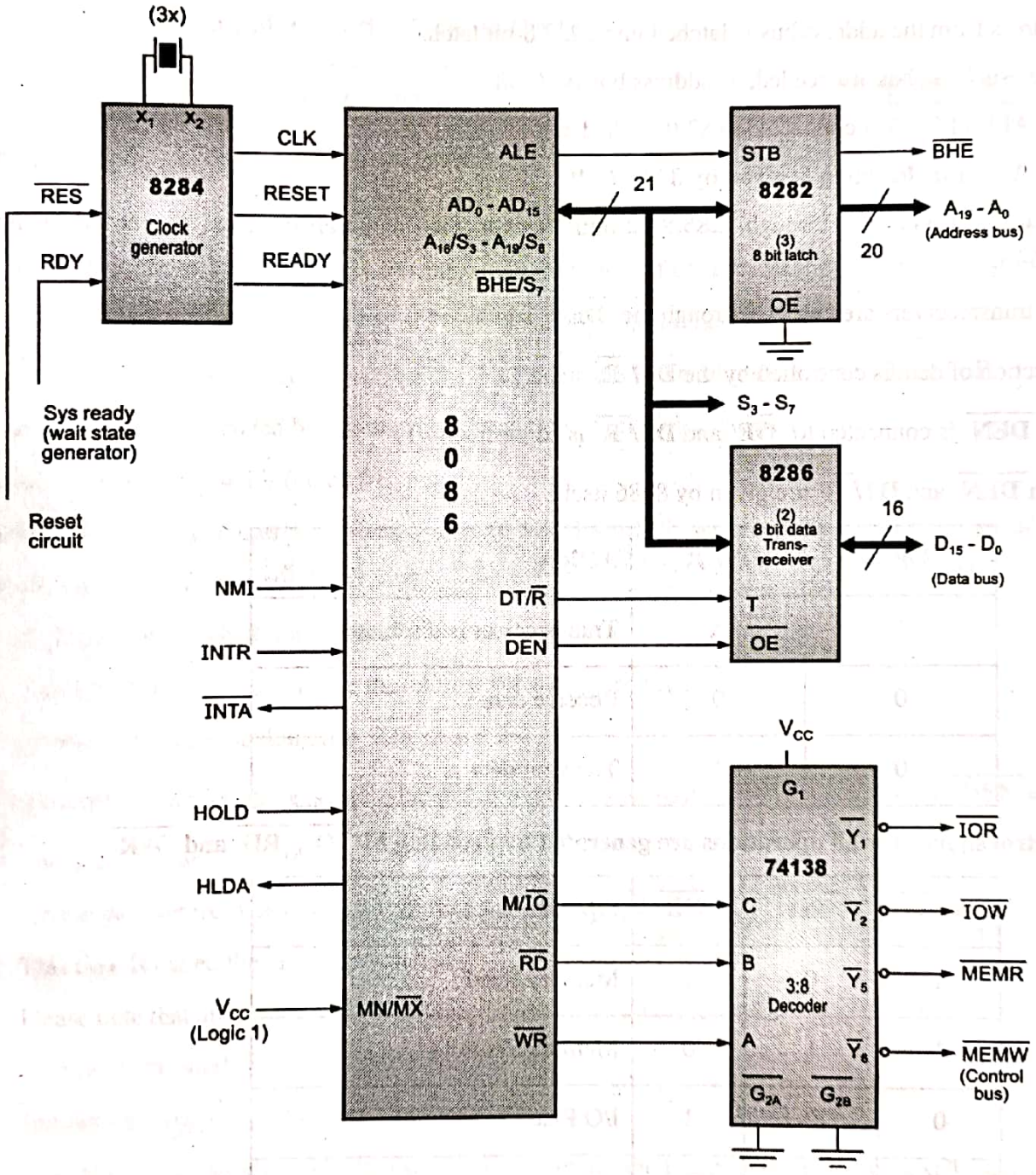
   F/ $\overline{\text{C}}$ = 0 → Input Clock given **through** Oscillator inputs $X_1$, $X_2$ pins.

- Generation of reset signal is done using a **power on reset circuit.**
- The purpose of this circuit is to activate the reset signal as soon as we supply power to 8086.
- This is done so that CS & IP acquire the above values and the system can initialize the monitor program (BIOS) as soon as 8086 is powered on. This is also called COLD starting the system.

✓ **Syllabus Topic : Functioning of 8086 in Minimum Mode, Timing Diagrams for Read and Write Operations in Minimum Mode**

## 4.4 8086 | Minimum Mode Configuration

**Q.** Explain minimum mode configuration of 8086 microprocessor. **(May 19, 10 Marks)**




(1A27)**Fig. 4.4.1 : 8086 Minimum Mode**

### 4.4.1 Explanation of Minimum Mode

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.**
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.

   Direction of data is controlled by the DT/ $\overline{R}$ signal.

   The $\overline{DEN}$ 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 $\overline{WR}$**

| M/ $\overline{IO}$ | $\overline{RD}$ | $\overline{WR}$ | Operation |
|---|---|---|---|
| 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.**

### 4.4.2 Timing Diagram of a "read" Machine Cycle

**Q.** Draw and discuss timing diagram for read operation in minimum mode of 8086. **(Dec. 17, 5 Marks)**

**Q.** Draw and explain memory read machine cycle timing diagram in minimum mode of 8086. **(Dec. 18, 5 Marks)**

- A machine cycle typically involves transferring a "single" value of data across the system.
- In a "read" machine cycle, data is transferred from memory or i/o to the µP.
- Memory read cycle transfers data from memory to µP, whereas I/O read transfers from I/O to µP.
- One machine cycle (also called bus cycle) involves 4 clock cycles also called T-states.
- 1 T-state = 1 clock cycle = 1 ÷ (clock frequency).
- Assume 8086 operating at 6 MHz, 1 T-state = 1 ÷ (6 MHz) = 0.167 microseconds or 167 nanoseconds.
- These are the activities performed in the 4 T-states

$T_1$ : ALE goes high as both multiplexed buses carry address.

This allows the latch to capture the address till ALE goes low.

$T_2$: µP asks for data by making the $\overline{RD}$ signal low.

As address is no longer present in the multiplexed bus, the transreceiver is enabled by $\overline{DEN} = 0$.

The data of course does reach the µP immediately.

Data is coming from memory hence it will take time to travel and reach the µP.

This time is called the "Propagation delay".

Please note that all these activities are only happening in nanoseconds.

$T_3$: Memory starts sending data on the data bus

Towards the end of $T_3$ µP checks Ready signal.

If Ready = 1 it means device is ready and µP completes the machine cycle in the next T-state (T4)

If Ready = 0 it means device is NOT ready

µP will then insert a wait state between $T_3$ and $T_4$

$T_4$: μP captures the data sent by the memory device

The multiplexed buses become tri-stated and will then open up in the next machine cycle

All control signals like $\overline{RD}$, $\overline{DEN}$ etc. are de-activated.

### 4.4.3 Timing Diagram of a "write" Machine Cycle

- A machine cycle typically involves transferring a "single" value of data across the system.
- In a "write" machine cycle, data is transferred from μP to memory or i/o.
- Memory write cycle transfers data from μP to memory, whereas I/O write transfers from μP to I/O.
- One machine cycle (also called bus cycle) involves 4 clock cycles also called T-states.

1 T-state = 1 clock cycle = 1 ÷ (clock frequency).

Assume 8086 operating at 6 MHz, 1 T-state = 1 ÷ (6 MHz) = 0.167 microseconds or 167 nanoseconds.

These are the activities performed in the 4 T-states

$T_1$: ALE goes high as both multiplexed buses carry address.

This allows the latch to capture the address till ALE goes low.

$T_2$: μP puts out the data on the data bus.

It informs the memory (or I/O device)that it wants to write (send) data.

This is done by making the $\overline{WR}$ signal low.

As address is no longer present in the multiplexed bus, the transreceiver is enabled by $\overline{DEN} = 0$.

$T_3$: Memory (or I/O) starts becoming ready and accept the data sent by μP

Towards the end of $T_3$ μP checks Ready signal.

If Ready = 1 it means device is ready and μP completes the machine cycle in the next T-state ($T_4$)

If Ready = 0 it means device is NOT ready

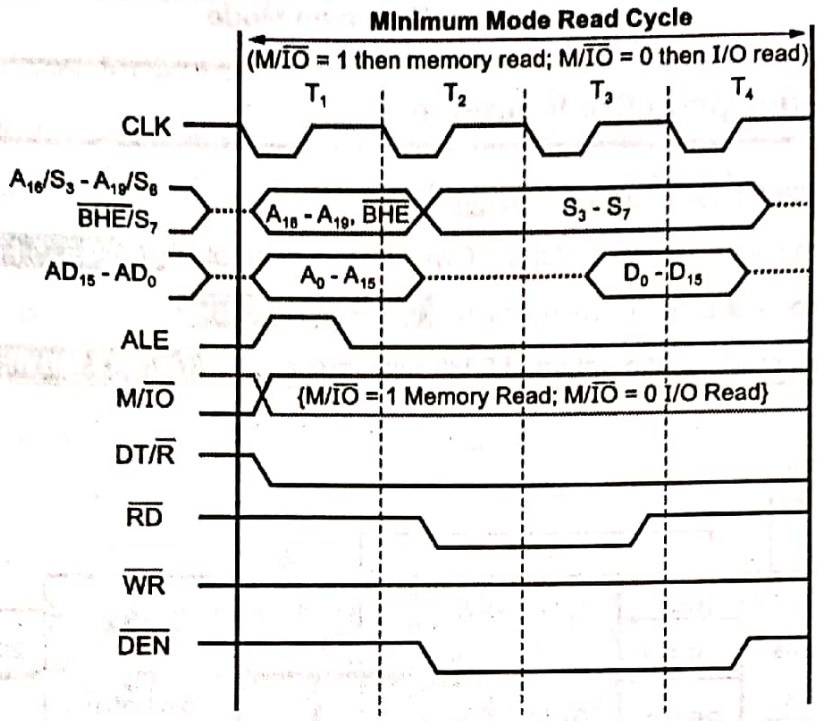
μP will then insert a wait state between $T_3$ and $T_4$

$T_4$: Data is captured by memory (or I/O device) and the operation is complete

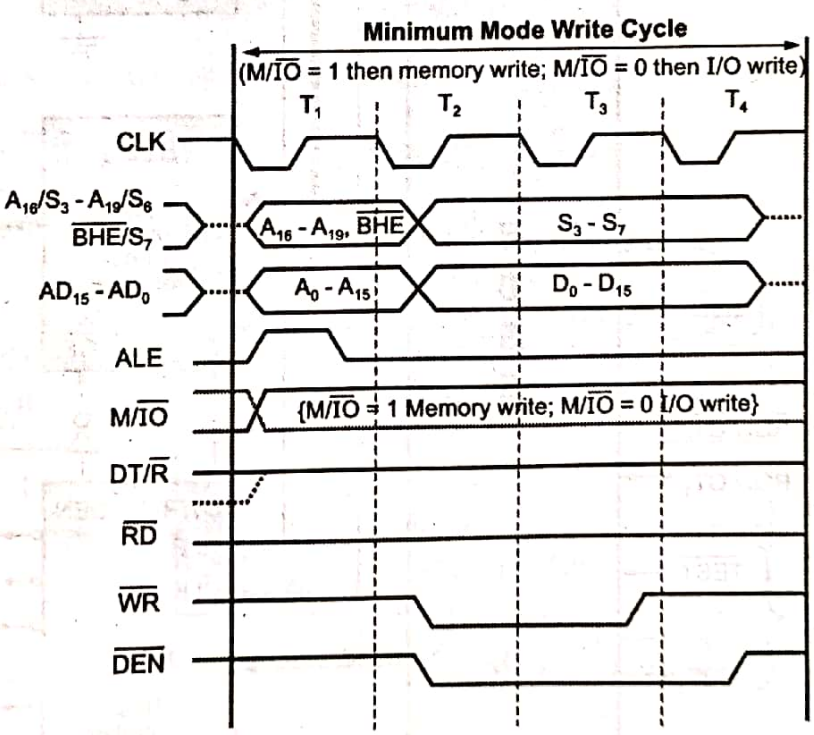
The multiplexed buses become tri-stated and will then open up in the next machine cycle

All control signals like $\overline{WR}$, $\overline{DEN}$ etc. are de-activated.

## Timing Diagrams




(1A28)**Fig. 4.4.2 : Minimum mode read cycle**




(1A29)**Fig. 4.4.3 : Minimum mode write cycle**

✓ **Syllabus Topic : Functioning of 8086 in Maximum Mode, Timing Diagrams for Read and Write Operations in Maximum Mode**

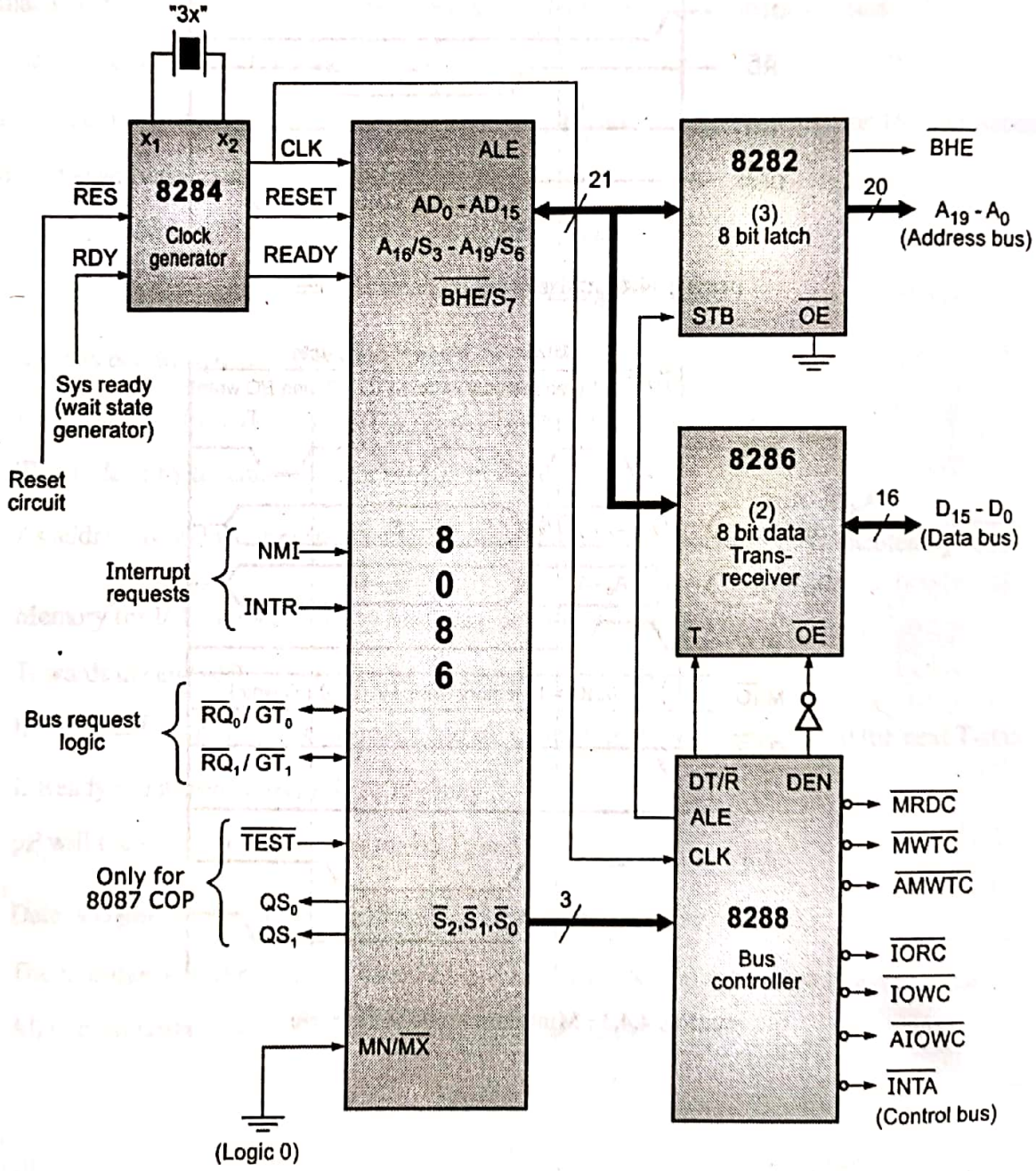
## 4.5 8086 | Maximum Mode Configuration

Q. Write short note on 8288 bus controller. **(Dec. 15, 5 Marks)**

Q. What is purpose of maximum mode of 8086 ? Give suitable example. **(Dec. 16, 5 Marks)**

Q. Explain maximum mode of 8086 microprocessor. **(Dec. 17, 10 Marks)**

Q. Explain the maximum mode configuration of 8086 microprocessor. **(Dec. 18, 10 Marks)**




(1A30) **Fig. 4.5.1 : 8086 maximum mode**

### 4.5.1 Explanation of Maximum Mode

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.

   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.**
8. **Roll of 8288 Bus controller in maximum mode.**

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

   The decoding of these signals are shown in the table below.

   8288 identifies the operation (machine cycle) that the active bus master needs to perform.

   Based on that, 8288 generates the appropriate control signal.

9. No control signal is needed in case of Idle and Halt states.

10. Instruction fetch cycle means fetching **"instruction"** from memory to μP.

    Memory read cycle means transferring **"data"** from memory to μP.

    Both essentially need the $\overline{\text{MRDC}}$ control signal.

11. For a write operation like memory write, 8288 generates both signals, the normal write and the advanced write i.e. $\overline{\text{MWTC}}$ **and** $\overline{\text{AMWTC}}$**.** 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".

| $\overline{S_2}$ | $\overline{S_1}$ | $\overline{S_0}$ | PROCESSOR STATE | 8288 ACTIVE OUTPUT |
|---|---|---|---|---|
| 0 | 0 | 0 | Int. Acknowledge | $\overline{\text{INTA}}$ |
| 0 | 0 | 1 | Read I/O Port | $\overline{\text{IORC}}$ |
| 0 | 1 | 0 | Write I/O Port | $\overline{\text{IOWC}}$ and $\overline{\text{AIOWC}}$ |
| 0 | 1 | 1 | Halt | None |
| 1 | 0 | 0 | Instruction Fetch | $\overline{\text{MRDC}}$ |
| 1 | 0 | 1 | Memory Read | $\overline{\text{MRDC}}$ |
| 1 | 1 | 0 | Memory Write | $\overline{\text{MWTC}}$ and $\overline{\text{AMWTC}}$ |
| 1 | 1 | 1 | Inactive | None |

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

    $RQ_0/GT_0$ has higher priority than $RQ_1/GT_1$.

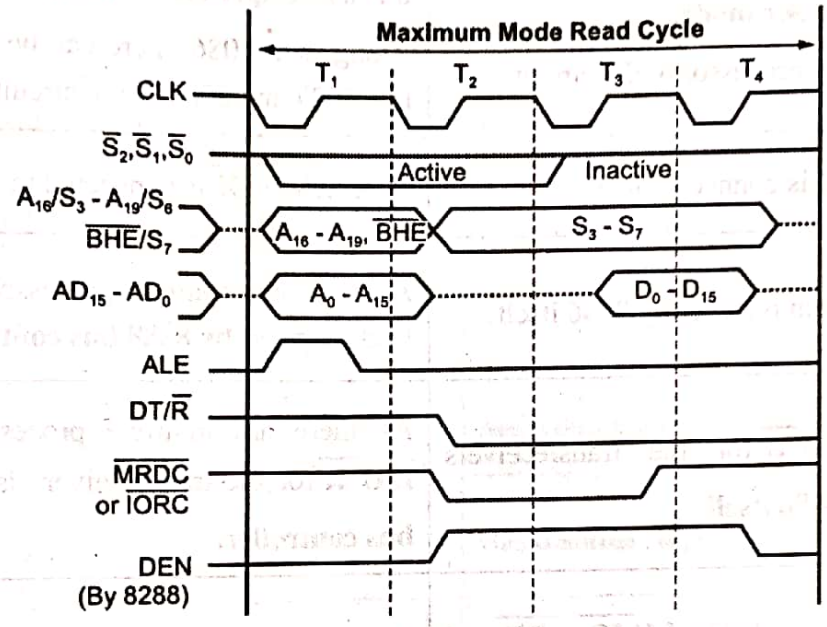
13. $\overline{\text{INTA}}$ **is given by 8288 Bus Controller, in response to an int. on INTR line of 8086.**

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

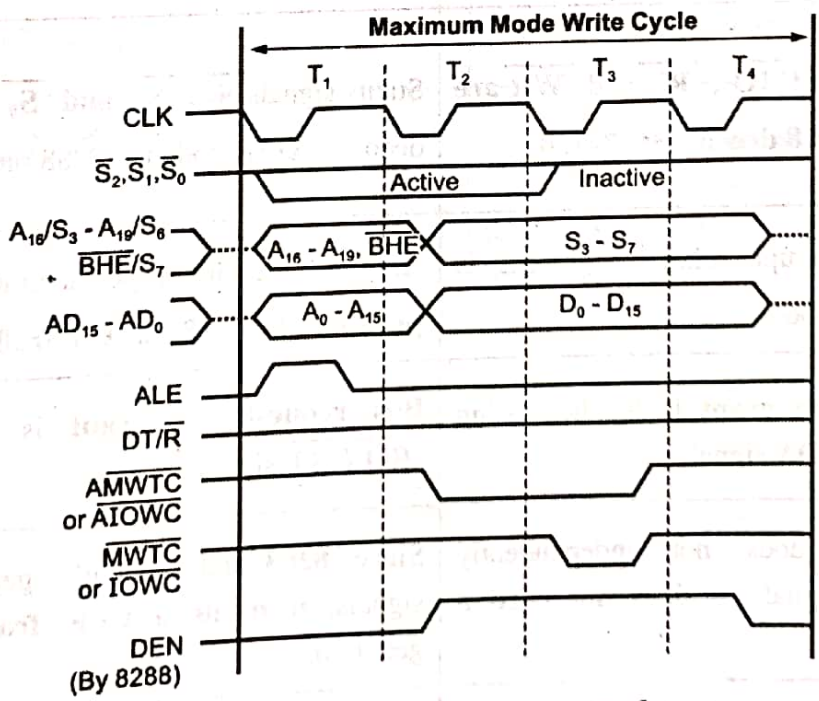
## 4.5.2 Timing Diagrams

Q. Draw and Explain write operation timing diagram for maximum mode. (May 18, 10 Marks)

Q. Draw and explain memory read and memory write machine cycle timing diagrams in maximum mode of 8086. (May 19, 10 Marks)




(1A31) **Fig. 4.5.2 : Maximum Mode Read Cycle**




(1A32) **Fig. 4.5.3 : Maximum Mode Write Cycle**

**Note :** For explanation refer theory of minimum mode timing diagrams.

## 4.5.3 Differentiate Between Min Mode and Max Mode

**Q. Differentiate between minimum mode and maximum mode in 8086. (Dec. 15, 5 Marks)**

| | MIN MODE | MAX MODE |
|---|---|---|
| 1 | It is a **uniprocessor mode.** 8086 is the only processor in the circuit. | It is a **multiprocessor mode.** Along with 8086, there can be other processors like 8087 and 8089 in the circuit. |
| 2 | Here **MN/ $\overline{MX}$** is connected to $V_{CC}$. | Here **MN/ $\overline{MX}$** is connected to **Ground.** |
| 3 | **ALE** for the latch is **given by 8086** itself. | As there are multiple processors, **ALE** for the latch is **given by 8288 bus controller.** |
| 4 | **$\overline{DEN}$** and **DT/ $\overline{R}$** for the transreceivers **are given by 8086 itself.** | As there are multiple processors, **DEN** and **DT/ $\overline{R}$** for the transreceivers is **given by 8288 bus controller.** |
| 5 | Direct control signals like **M/ $\overline{IO}$ , $\overline{RD}$** and **$\overline{WR}$ are produced by 8086 itself.** | Instead of control signals, all processors produce status signals **$\overline{S_2}$ , $\overline{S_1}$ and $\overline{S_0}$** . |
| 6 | Control signals **M/ $\overline{IO}$ , $\overline{RD}$** and **$\overline{WR}$ are decocded by a 3:8 decoder IC 74138.** | Status signals **$\overline{S_2}$ , $\overline{S_1}$ and $\overline{S_0}$** require special decoding are decoded by8288 bus controller. |
| 7 | **$\overline{INTA}$** for interrupt acknowledgement **is produced by 8086.** | **$\overline{INTA}$** for interrupt acknowledgement is **produced by 8288 Bus Controller.** |
| 8 | **Bus request are grant** is handled using **HOLD and HLDA** signals. | **Bus request are grant** is handled using **$\overline{RQ}$ / $\overline{GT}$** signals. |
| 9 | Since **74138** does not independently generate any signals, it **does not need a CLK.** | Since **8288** independently generates control signals, it **needs a CLK from 8284** clock generator. |
| 10 | The circuit is **simpler but does not support multiprocessing.** | The circuit is **more complex but supports multiprocessing.** |

### 4.5.4 Differentiate between 8085 and 8086

| | 8085 | 8086 |
|---|---|---|
| 1 | **8-bit processor** with:<br>**8-bit ALU** and<br>**8-bit data bus.** | **16-bit processor** with:<br>**16-bit ALU** and<br>**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.** |

### 4.5.5 Differentiate between 8088 and 8086

| | 8088 | 8086 |
|---|---|---|
| 1. | **16-bit processor** with:<br>**16-bit ALU** and<br>**8-bit data bus.** | **16-bit processor** with:<br>**16-bit ALU** and<br>**16-bit data bus.** |
| 2. | Memory **banking not needed.**<br>Hence **circuit is simpler.** | **Memory** is divided into **two banks.**<br>Hence **circuit is more complex.** |
| 3. | Since **data bus is 8-bits,** it can transfer **1 byte in 1 cycle.**<br>Hence is slower. | Since **data bus is 16-bits,** it can transfer **2 bytes in 1 cycle.**<br>Hence is **faster.** |
| 4. | $\overline{\text{BHE}}$ **is not needed.**<br>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/** $\overline{\text{M}}$ compatible with 8085. | Uses **M/** $\overline{\text{IO}}$ to differentiate between memory and I/O operations. |

## 4.6 University Questions and Answers

**Dec. 2015**

**Q. 1(a)** Write short note on 8288 bus controller. (*Refer section 4.5*) **(5 Marks)**

**Q. 3(c)** Differentiate between minimum mode and maximum mode in 8086. (*Refer section 4.5.3*) **(5 Marks)**

**Dec. 2016**

**Q. 1(a)** What is purpose of maximum mode of 8086 ? Give suitable example. (*Refer section 4.5*) **(5 Marks)**

**Dec. 2017**

**Q. 2(a)** Explain maximum mode of 8086 microprocessor. (*Refer section 4.5*) **(10 Marks)**

**Q. 6(b)** Draw and discuss timing diagram for read operation in minimum mode of 8086. (*Refer section 4.4.2*) **(5 Marks)**

➠ **May 2018**

**Q. 3(a)** Draw and Explain write operation timing diagram for maximum mode. (*Refer section 4.5.2*) **(10 Marks)**

➠ **Dec. 2018**

**Q. 1(a)** Draw and explain memory read machine cycle timing diagram in minimum mode of 8086. (*Refer section 4.4.2*) **(5 Marks)**

**Q. 2(a)** Explain the maximum mode configuration of 8086 microprocessor. (*Refer section 4.5*) **(10 Marks)**

➠ **May 2019**

**Q. 2(a)** Explain minimum mode configuration of 8086 microprocessor. (*Refer section 4.4*) **(10 Marks)**

**Q. 6(a)** Draw and explain memory read and memory write machine cycle timing diagrams in maximum mode of 8086. (*Refer section 4.5.2*) **(10 Marks)**

---

*Chapter ends...*

❑❑❑


# 8259 Programmable Interrupt Controller

## 5.1 8259 | Introduction

- 8259 is a Programmable Interrupt Controller.
- It is used to **increase the number of hardware interrupts** for the processor.
- In single 8259 can provide **8 interrupts**
- A cascaded configuration can provide up to **64 interrupts** using 1 Master 8259 and 8 Slave 8259s.
- Though 8259 was created during the era of 8085 μP, it was in service for various generations of the x86 family starting with the 8086 μP.
- It has several flexible properties such has **masking, edge/level triggering, priority modes, programmable vector numbers** and so on.

Before we start learning the bigger topics like architecture, working and programming of 8259, first lets clearly understand why we need to use 8259 in the circuit.

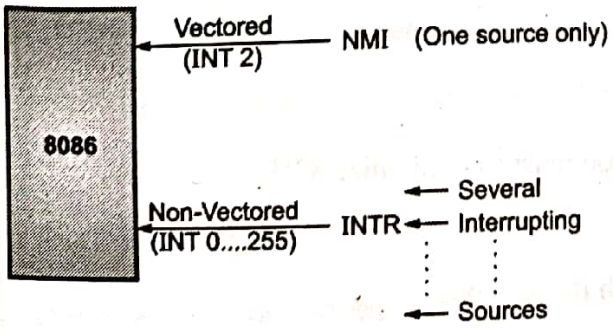
## 5.2 Need for 8259 PIC

**Q.** Write short note on : 8259-PIC **(May 16, 5 Marks)**

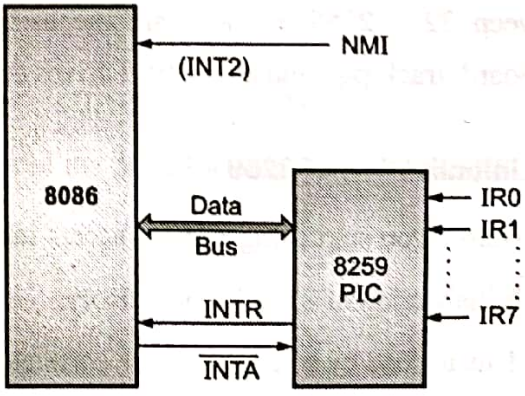
- You know, 8086 has only two Hardware Interrupts: NMI and INTR.
- Out of which, **NMI is vectored**. This means it has a fixed vector number in the IVT that is 2.
- So, whenever 8086 gets NMI interrupt, it will always execute the ISR of INT 2. This makes NMI "rigid".
- We can connect only a single device to interrupt 8086 through NMI because the ISR is fixed.

☞ **INTR on the other hand is non-vectored**

- It does not have a fixed vector number and hence 8086 can execute any ISR from INT 0… INT 255.
- This makes INTR a "flexible" interrupt line.
- We can combine several interrupting devices to give a common interrupt to 8086 on INTR.
- The idea makes sense, but we will need additional hardware.




(1B1)**Fig. 5.2.1 : Interrupt sources**




(1B2)**Fig. 5.2.2 : Interrupts via 8259**

- We cant just simply "combine" several interrupting sources (devices) into a common pin like INTR.
- This is because the processor would never really understand which of the devices has sent the interrupt.
- Moreover, the processor will also not be able to enforce any priority amongst them if several devices interrupt simultaneously.
- This is why we need 8259, as an Interface between the Interrupting devices and 8086.
- 8259 has 8 Interrupt input lines, IR0… IR7.
- We can connect 8 different interrupt sources (devices) on these lines. These are devices which wish to interrupt 8086.
- When 8259 gets an interrupt, it interrupts the μP on INTR pin.
- 8086 does not know the vector number as INTR is non-vectored.
- To obtain the vector number, 8086 issues $\overline{\text{INTA}}$ signal. There are 2 $\overline{\text{INTA}}$ pulses issued.
- When 8086 gives the 1st $\overline{\text{INTA}}$ signal, 8259 starts preparing the vector number.
- When 8086 gives the 2nd $\overline{\text{INTA}}$ signal, 8259 sends the 8 bit the vector number to 8086 through an 8 bit data bus. Although 8086 has a 16 bit data bus, 8259 is an 8 bit device and hence has an 8 bit data bus.
- Once 8086 gets the vector number, it multiplies the number by 4 and goes to the appropriate location in the IVT to obtain the ISR address and hence execute the ISR.
- Now that you have understood this basic concept, you will start getting some questions in your mind!
- How does 8259 know the vector numbers of the interrupting devices? How does priority get decided? What is the μP temporarily want to mask/ unmask some interrupts? How are edge and level triggered interrupts selected? The answers to all these questions lies in our next section: Initialization of 8259 PIC.

### ☞ #Just For Knowledge

In theory, the vector number can be anything from 0… 255, but in the real world it is generally a number between 32… 2555 as those are the user defined interrupts typically meant for user defined devices like keyboard, track-pad, mouse, optical drives etc.

### ☞ Initialization of 8259

- Before you start using 8259 to accept interrupts for 8086, you must first initialize 8259.
- Initialization of 8259 is compulsory.
- This is done by sending various **commands** to 8259 through the data bus.
- We form these commands and put them into a register like AL.

- Then using the **OUT instruction** we send these commands from 8086 to 8259 through the data bus.
- The commands are called ICWs and OCWs discussed in full detail later in this chapter.
- Using these commands we decide several properties such are Priority modes, Masking, Trigger modes
- (Edge or Level) and most importantly we inform the Vector numbers of all interrupts to 8259.
- You must at this point yourself realize that this initialization of 8259 is compulsory.
- If we do not inform 8259 the vector numbers beforehand, it will send random vector numbers to 8086 whenever it gets an interrupt. This will result in μP executing some random ISR and hence the whole interface will fail.
- You may argue that initializing vector numbers of all 8 interrupts will be time-consuming.
- This problem is overcome by a simple trick.
- We do not really give 8 vector numbers to 8259 during initialization.
- We give only one vector number (typically of IR0).
- The remaining vector numbers are taken by 8259 in a sequence.
- So if we give vector number of IR0 as 40H, 8259 will automatically establish the 8 vector numbers as:

  40H... 47H for IR0... IR7 respectively.
- Remember vector numbers can be anything from 0... 255 which means 00H... FFH.
- Does this mean 8259 has a secret ALU or an incrementer to derive vector numbers 41H... 47H from the original number 40H?
- No! 8259 simply "appends" the interrupt line number (IR0..7) with the lowest three bits of the original vector number. Lets do it with an example.
- Say vector number of IR0 is 40H.
- In binary it is: 0100 0000.
- Even better, it is: 0100 0 _ _ _
- For IR0: Append the lowest three bits as 000. Hence vector number is 0100 0 <u>000</u> = 40H
- For IR1: Append the lowest three bits as 001. Hence vector number is 0100 0 <u>001</u> = 41H
- For IR7: Append the lowest three bits as 111. Hence vector number is 0100 0 <u>111</u> = 47H
- Sometimes it is amazing to think how such simple ideas can avoid the need for complex hardware like an ALU or an incrementer, isn't it!

### ☞ Cascading of 8259

- This is the final piece of "introduction" you need to be familiar with. The need for cascading.

- Yes, you might have guessed it... To further increase the number of interrupts.
- On single 8259 can provide 8 interrupts.
- Now imagine if we connect another 8259 it one of the 8 interrupts of the first 8259 to form a cascade.
- This will in turn provide 8 more interrupts.
- This is called a "Cascaded configuration" of 8259s, also called a "Master - Slave" configuration.
- The 8259 that directly interrupts 8086 on INTR pin is called the Master.
- The 8259s that interrupt the master are called the Slaves.
- We could connect a Slave to each of the interrupt lines of the master from IR0... IR7.
- This would give us 9 8259s: 1 – Master and 8 – Slaves.
- Such a configuration will provide a total of 64 interrupts.
- To be honest, such a large number of hardware interrupts is never really required in a regular computer.
- There cannot be a third level of cascading.
- This is because the 8259s at the mid level would have to work as both: Master and Slave.
- By the time you understand cascading in full detail you will agree that this is just not possible.
- The initialization and behavior of the master is different from that of the salve.
- It is important to note, in a cascaded configuration, every 8259 has to be individually initialized.
- The Master as well as all the slaves have to each be initialized.

### ☞ 8259 | 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.

(6) In 8259 interrupts can be **masked** individually.

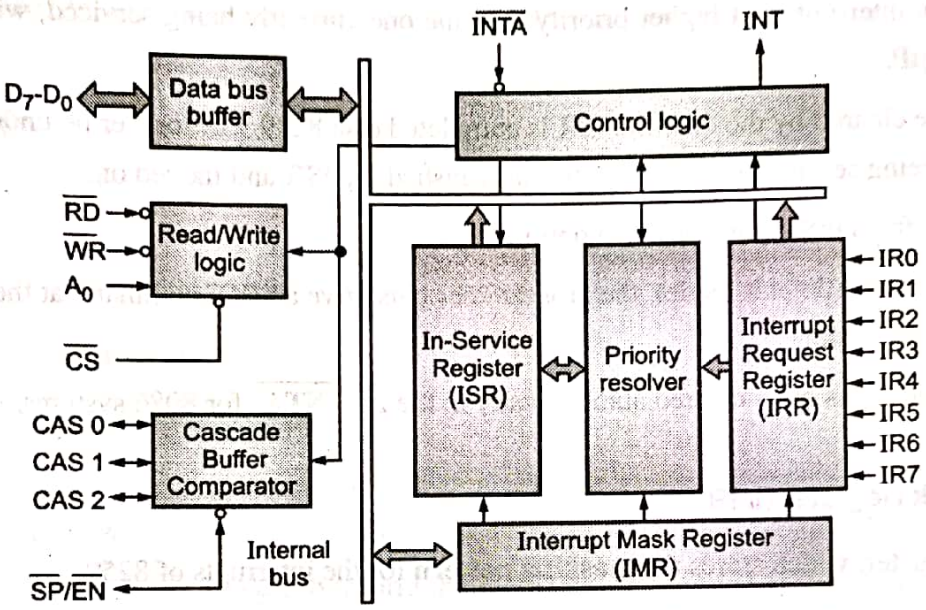
(7) The **Vector address** of the interrupts is **programmable**.

(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.

✓ **Syllabus Topic : Programmable Interrupt Controller 8259 - Block Diagram**

## 5.3 8259 | Architecture | Block Diagram




(1B3)**Fig. 5.3.1 : 8259 Architecture**

- The main components of the architecture are as follows,

### ☞ Interrupt Request Register (IRR)

- 8259 has **8 interrupt** input lines **IR7... IR0**.
- These lines are active high. By default, when no interrupt occurs, these lines are at logic 0.
- When an interrupt occurs the corresponding line becomes logic 1.
- 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** (becomes 1)in the Interrupt Request Register (**IRR**).
- This bit reminds the processor that the interrupt is pending.
- It means, the interrupt has occurred but not yet serviced.
- The bit will only become 0 when the µP acknowledges this interrupt and responds by sending 1st $\overline{\text{INTA}}$ .

### ☞ In-Service Register (InSR)

- It is an **8-bit** register. It **stores** the **level** of the Interrupt Request, **currently** being **serviced**.
- When the µP send the 1st $\overline{\text{INTA}}$ the corresponding bit becomes 1 in this register, indicating that from now on, this particular interrupt is being serviced.

- This bit must stay 1 till the end of the ISR.
- During this same ISR if any other interrupt occurs, the priority resolver compares its level (in IRR) with the interrupt which is currently being serviced (in InSR).
- Only if the new interrupt is of higher priority than the one currently being serviced, will 8259 send the new interrupt to the μP.
- This bit must be cleared by the time the ISR is completed else 8259 will forever be under the impression that the ISR is still being serviced even though μP has finished the ISR and moved on.
- To clear this bit from InSR, there are two options.
- In Normal EOI mode (default mode), the programmer must give an EOI command at the end of the ISR. EOI stand for End of Interrupt.
- In Auto EOI mode, this bit is cleared automatically in the 2nd $\overline{\text{INTA}}$ for 8086 systems.

### ☞ 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.**
- The interrupts can be masked or unmasked by the programmer as many times as they may wish to.
- This is done by the OCW1 command.
- To mask a bit, the corresponding bit must be made 1.
- Now even if this interrupt occurs, it will not be sent to the μP, irrespective of its priority.

### ☞ Priority Resolver

- It **examines** the **IRR**, **InSR**, and **IMR** and determines which interrupt is of **highest priority** and should be sent to the μP.
- It checks IRR to know which interrupts have occurred, IMR to know which interrupts are masked and InSR to know which interrupt is in service.
- If the interrupt that has occurred is of the highest priority amongst those that have occurred, is unmasked and is higher priority than the one currently being serviced, only then the interrupt will be sent to the μP.

### ☞ 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 by sending internal control signals.

### ☞ Data Bus Buffer

- It is a bi-directional buffer used to **interface** the internal **data bus** of 8259 with the external (system) data bus.
- The data bus is used to transfer commands from μP to 8259 and also give the vector number from 8259 to the μP.

### ☞ Read/Write Logic

- It is used to accept the $\overline{RD}$, $\overline{WR}$, $A_0$ and CS signal.
- It is also used to hold some of the Initialization Command Words (ICW's) and the Operational Command Words (OCW's).

### ☞ Cascade Buffer / Comparator

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

**1) CAS 2, CAS 1, CAS 0 lines**

- CAS lines are used by the master, to inform the slave, that it has been selected, between the 1st and the 2nd $\overline{INTA}$, so that the slave can give the correct vector number to the μP.
- There are 3 CAS lines.
- They can carry any number between 000… 111.
- This means there can be a maximum of 8 slaves connected to the master with the Slave Ids ranging from 000… 111.

**2) $\overline{SP}$ / $\overline{EN}$ (Slave Program/Master Enable):**

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

✓**Syllabus Topic : Operating Modes**

## ➲ 5.4 8259 I Operating Modes

Q. Explain Operating Modes of PIC 8259. **(May 18, 10 Marks)**

### ☞ Priority Modes

### 5.4.1 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.

### 5.4.2 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.

### 5.4.3 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.**
- **Eg:** If $IR_2$ is has just been serviced, it will get the lowest priority.

#### Specific Rotation Mode

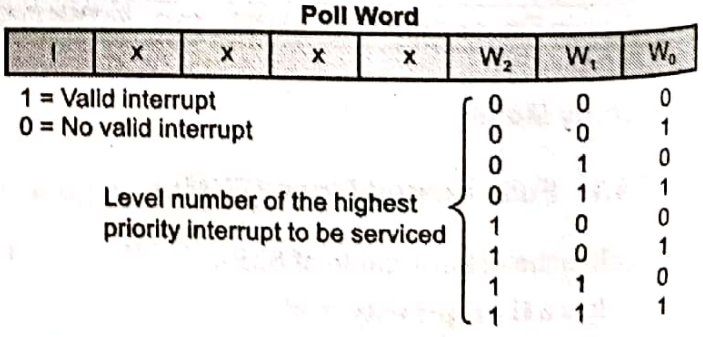
It is **also** a **rotating** priority mode, **but here** the **user can select** any **IR level for lowest priority,** and thus fix all other priorities.

### 5.4.4 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

### 5.4.5 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.
- Thereafter the µP services the interrupt.




(1B4)**Fig. 5.4.1 : Poll Word**

**Advantage**

The µP's programs 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.

### 5.4.6 Buffered Mode

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

### 5.4.7 EOI Modes

- EOI means End of Interrupt.
- EOI command is given by the programmer using OCW command word of 8259.
- Normally the programmer gives an EOI command an the end of the ISR.
- This is used to indicate to 8259 that the ISR is over and hence 8259 must clear the corresponding bit in the In Service register.
- Based on whether the EOI command must be given by the programmer or it must be performed automatically, there are two EOI modes, Normal EOI mode and Auto EOI mode.

#### 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 and Specific EOI Command

#### 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

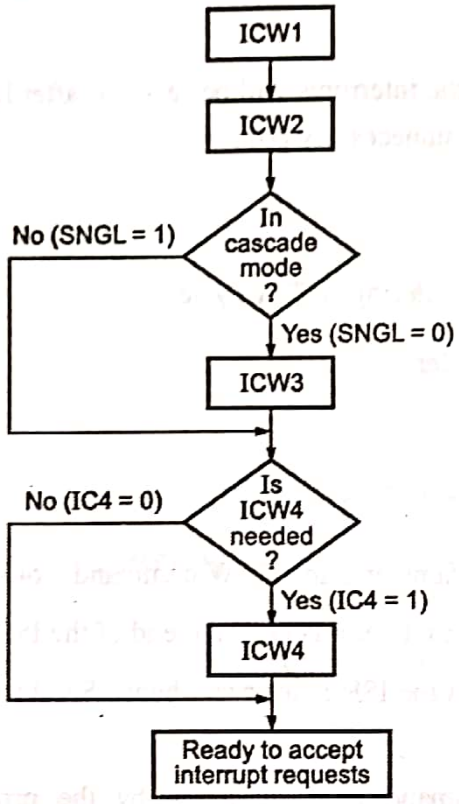
Here the programmer specifies the Bit number to be cleared from In Service Register.

#### 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 $\overline{INTA}$ pulse.

## 5.5 Initialization Sequence of 8259

**Q.** Give formats of initialisation command words(ICW's) of 8259 PIC. **(Dec. 18, 5 Marks)**



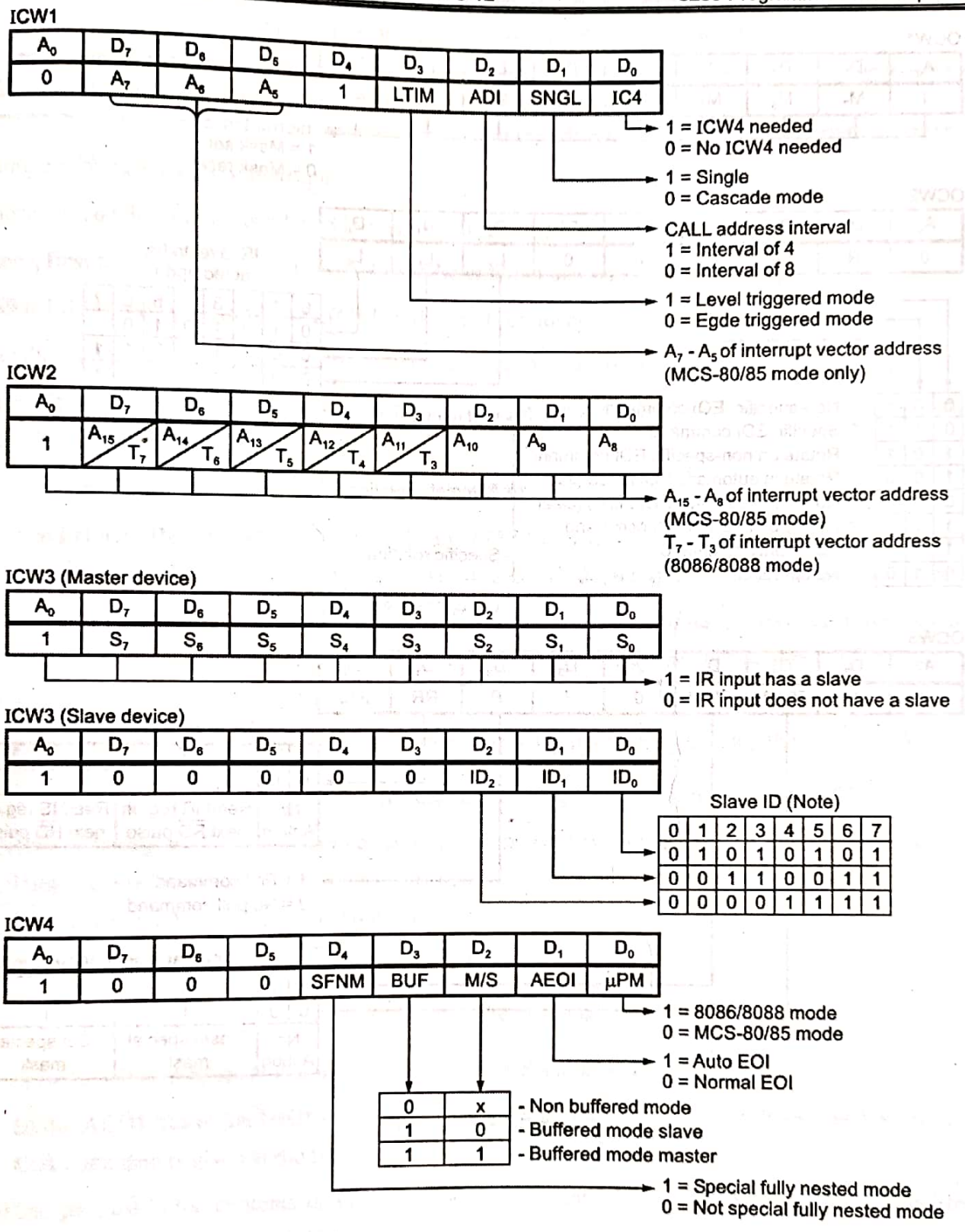

(185)**Fig. 5.5.1 : Initialization Sequence of 8259**

### ICWs

- **ICWs** have to be **given during** the **initialization** of 8259 (i.e. **before** the µP can start **using 8259**).
- **ICW1** and **ICW2** are **compulsory.**
- ICW1 indicates if the system is Single or Cascaded.
- **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 **given.**
- It is **important** that the ICWs are **given 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.




(1B6)**Fig. 5.5.2 : ICWs**

OCW1

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

Interrupt mask
1 = Mask set
0 = Mask reset

OCW2

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

IR level to be acted upon

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

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

† $L_0$ - $L_2$ are used

OCW3

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

Read register command

| | | | |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| No Action | Read IR reg on next $\overline{RD}$ pulse | | Read IS reg on next $\overline{RD}$ 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 |

(1B7)Fig. 5.5.3 : OCWs

✓ **Syllabus Topic: Interfacing the 8259 in Single Mode**

## 5.6 8259 I 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, ICW2** 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) and **InSR** (for the current level serviced) and thus **determines** the **highest priority** interrupt.

(3) If the interrupt that has occurred is of the highest priority amongst those that have occurred, is unmasked and is higher priority than the one currently being serviced, only then the interrupt will be sent to the µP. It **sends** the **INT** signal **to** the **µP.**

(4) µP finishes the current instruction and acknowledges the interrupt by sending the first $\overline{\text{INTA}}$.

(5) 1st $\overline{\text{INTA}}$ is a confirmation by the µP that the interrupt will be serviced. Hence 8259 will make the corresponding bit "1" in In Service register and the corresponding bit "0"in IRR as this interrupt is no longer a pending interrupt. Instead it is the one currently being serviced. **8259** now **prepares** to send the Vector number **N** to the µP on the data bus.

(6) The µP sends the second $\overline{\text{INTA}}$ pulse to 8259.

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

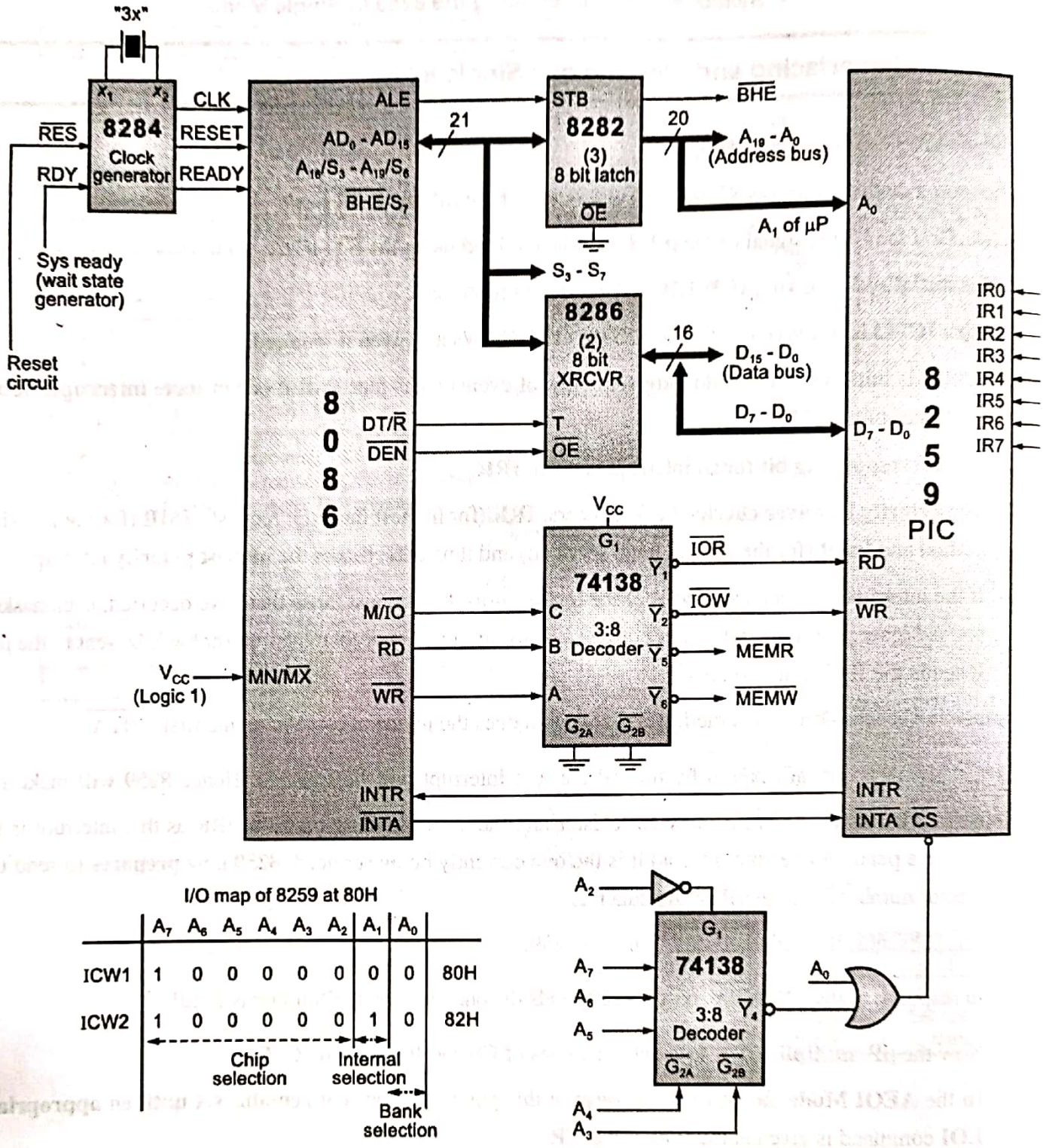
(8) Now the **µP multiplies N × 4**, to get the values of CS and IP **from the IVT.**

(9) **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.

(10) 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.

(11) The ISR thus begins.

(12) At the end of the ISR, the programmer gives an EOI command (assuming the Normal EOI mode). This makes the corresponding bit "0" in In Service Register of 8259.




(1B8)**Fig. 5.6.1 : Interfacing of Single 8259**

✓ **Syllabus Topic : Interfacing the 8259 In Cascaded Mode**

## 5.7 8259 I Interfacing and Working of a Cascaded 8259

Q. Interface three 8259s with 8086 in minimum mode and explain its functionality in fully nested mode. **(Dec. 17, 10 Marks)**

Q.. Explain the operation of three 8259 PIC in cascaded mode. **(May 19, 10 Marks)**

- 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 $\overline{SP}$ / $\overline{EN}$ = + **5V** and the **slave** has **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 $\overline{INTA}$ pulses.

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

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

The Master sets the corresponding bit in its InSR.

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

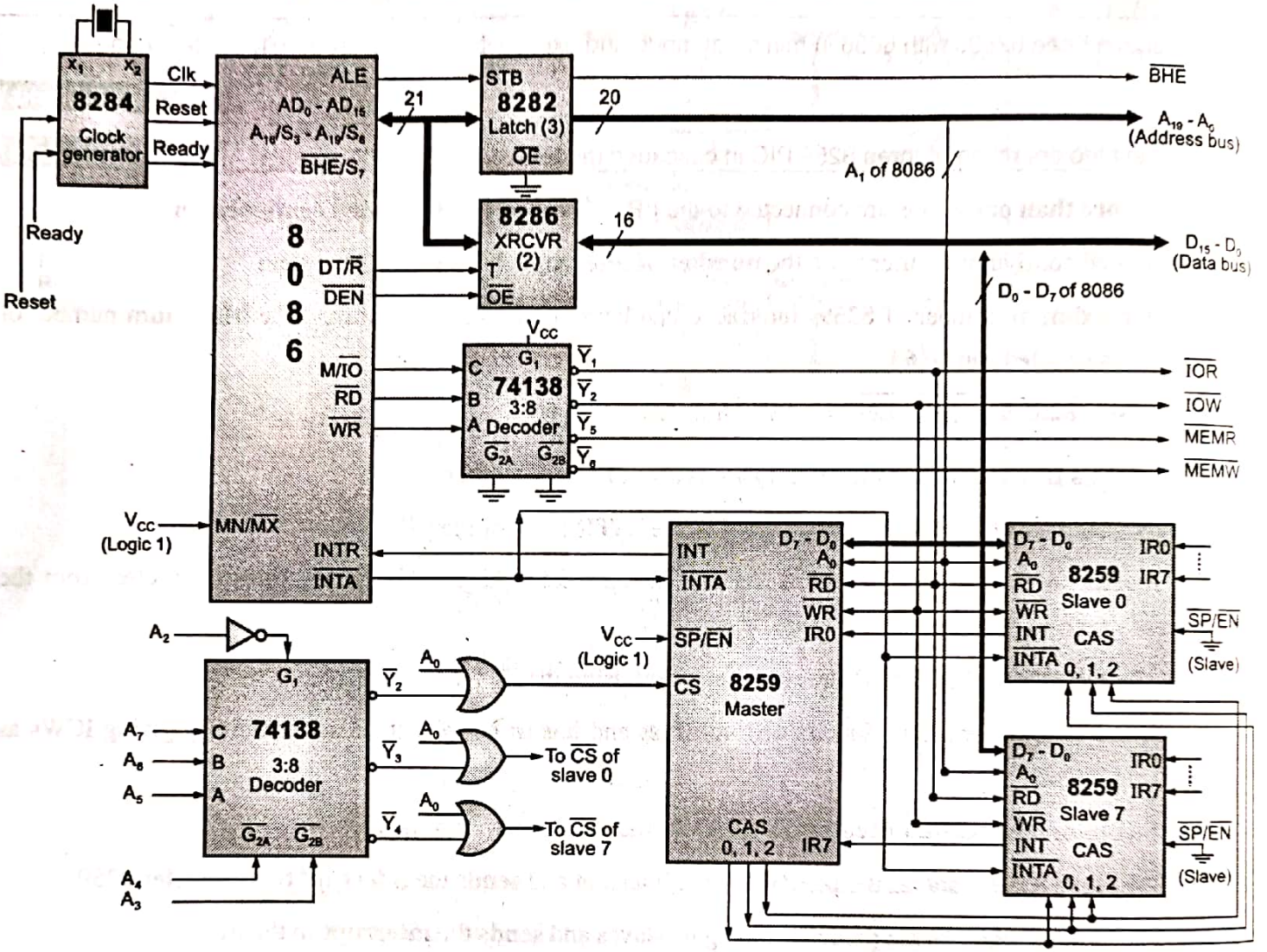
(5) In response to the second $\overline{INTA}$ pulse the slave places Vector Number N on the data bus.

(6) During the 2nd $\overline{INTA}$ 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.

(8) The ISR thus begins.

(9) At the End of the ISR, EOI commands are given to the master and slaves to make the corresponding bit "0" in the In Service Registers.




(1B9)**Fig. 5.7.1 : Interfacing of Cascaded 8259**

| I/O map | | $A_7$ | $A_6$ | $A_5$ | $A_4$ | $A_3$ | $A_2$ | $A_1$ | $A_0$ | I/O Address |
|---|---|---|---|---|---|---|---|---|---|---|
| 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 |

## 5.8 8259 I Programming

- Programming question of this type has been asked in several MU papers.
- Based on their given specifications, you need to form the commands and then send them to 8259 using OUT instruction. Normally in the exam you are asked to initialize a **single 8259** for some given specifications.
- 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.

✓ **Syllabus Topic : Programs for 8259 using ICWs and OCWs**

### 5.8.1 Program to initialize Single 8259

**Program 5.8.1 :** Write a program 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
```

**Program 5.8.2 :** Write a program 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.

**Soln. :**

```
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
```

## 5.9 University Questions and Answers

**May 2016**

**Q. 6(d)** Write short note on : 8259-PIC. (*Refer section 5.2*) **(5 Marks)**

**Dec. 2017**

**Q. 2(b)** Interface three 8259s with 8086 in minimum mode and explain its functionality in fully nested mode. (*Refer section 5.7*) **(10 Marks)**

**May 2018**

**Q. 3.(b)** Explain Operating Modes of PIC 8259. (*Refer section 5.4*) **(10 Marks)**

**Dec. 2018**

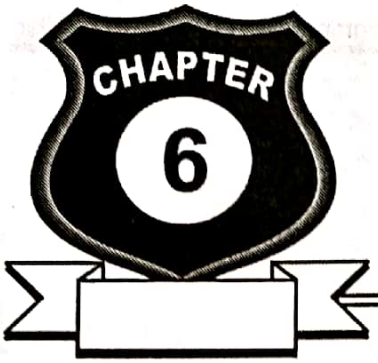
**Q. 1(d)** Give formats of initialisation command words(ICW's) of 8259 PIC. (*Refer section 5.4*) **(5 Marks)**

**May 2019**

**Q. 5(b)** Explain the operation of three 8259 PIC in cascaded mode. (*Refer section 5.7*) **(10 Marks)**

*Chapter ends...*

❑❑❑




# 8255 Programmable Peripheral Interface

## 6.1 8255 | Introduction

1. 8255 is a Programmable Peripheral Interface.
2. It is used to interface (connect) μP with I/O devices such as a printer, keyboard etc. 8255 has three 8 bit bidirectional I/O ports called Port A, Port B and Port C.
3. They can be used as input ports or as output ports.
4. These ports can operate in three different modes of data transfer.
5. Mode 0 is called Simple I/O. Here the three ports perform basic 8 bit data transfers.
6. Mode 1 is called Handshake I/O. Here port A and Port B transfer data using an advanced technique called handshaking. This prevents any loss of data and hence makes the transfer more reliable. To carry out handshaking, lines of Port C are used up by Port A and Port B.
7. Mode 2 is bidirectional handshaking in which as the name suggests, the port can perform input as well as output handshaking.
8. The modes are selected by the programmer by giving commands to 8255. Depending upon the system requirement, the appropriate command is formed and is first stored in a register like AL. Then using an instruction like OUT, the command is sent to 8255 through the data bus.
9. Additionally, 8255 has an excellent feature called BSR command. It stands for Bit Set Reset. Using BSR command, the programmer can individually set or reset (send 1 or 0) to any line of Port C without affecting other lines. This basically transforms Port C from being one 8 bit port to becoming eight 1 bit ports that can be individually controlled by the programmer. This feature is very useful in complex interfaces like ADC, LCD controller etc.

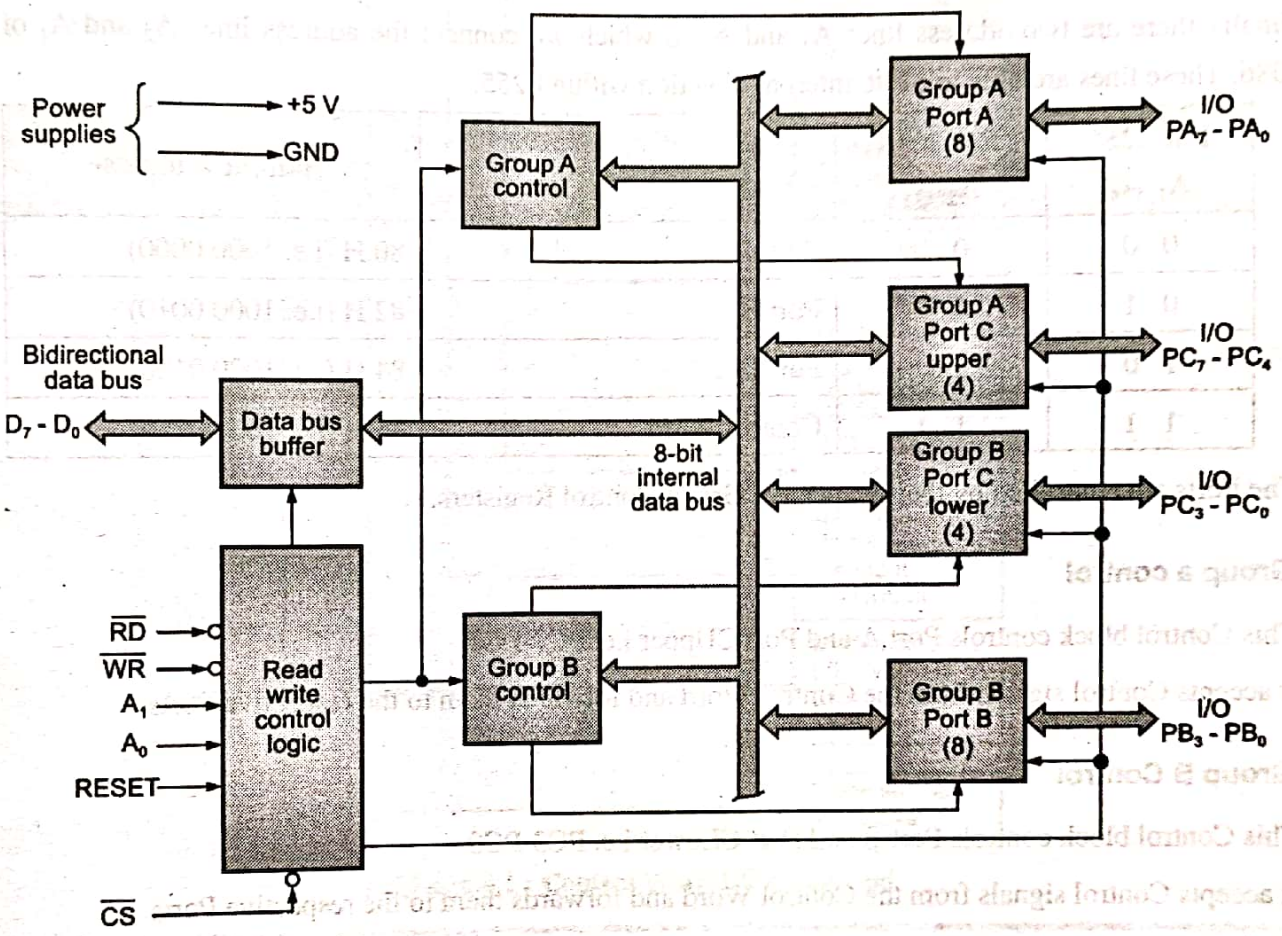
## 6.2 8255 | Salient Features

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.

✓ **Syllabus Topic : 8255-PPI – Block Diagram**

## 6.3 8255 I Architecture I Internal Block Diagram

**Q.** Explain with block diagram working of 8255 PPI. **(Dec. 15, 10 Marks)**
**Q.** Explain PPI 8255 with block diagram. **(May 18, 10 Marks)**
**Q.** Draw and explain the block diagram of 8255 Programmable Peripheral Interface (PPI) with control word formats. **(May 19, 10 Marks)**




(1C1)**Fig. 6.3.1 : Architecture of 8255**

The architecture of 8255 has the following main components…

### ☞ 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 the data bus via this buffer.
- Moreover, the commands given to 8255 (I/O command and BSR command) are also given through the same data bus.

### ☞ 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.
- During a read, data will be transferred from 8255 to the μP.
- During a write, data will be transferred from μP to 8255.
- There is a chip selection signal that selects 8255 on the basis of its address.
- The reset signal is to reset 8255 and stop all current transfers.
- Finally there are two address lines $A_1$ and $A_0$ to which we connect the address lines $A_2$ and $A_1$ of the μP 8086. These lines are uses to make internal selection within 8255.

| 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 0000) |
| 0 1 | 0 1 | Port B | 82 H (i.e. 1000 0010) |
| 1 0 | 1 0 | Port C | 84 H (i.e. 1000 0100) |
| 1 1 | 1 1 | Control Word | 86 H (i.e. 1000 0110) |

The Ports are controlled by their respective Group Control Registers.

### ☞ Group a control

- This Control block controls Port A and Port CUpper i.e. PC7-PC4.
- It accepts Control signals from the Control Word and forwards them to the respective Ports.

### ☞ Group B Control

- This Control block controls Port B and Port CLower i.e. PC3-PC0.
- It accepts Control signals from the Control Word and forwards them to the respective Ports.

### ☞ 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 :

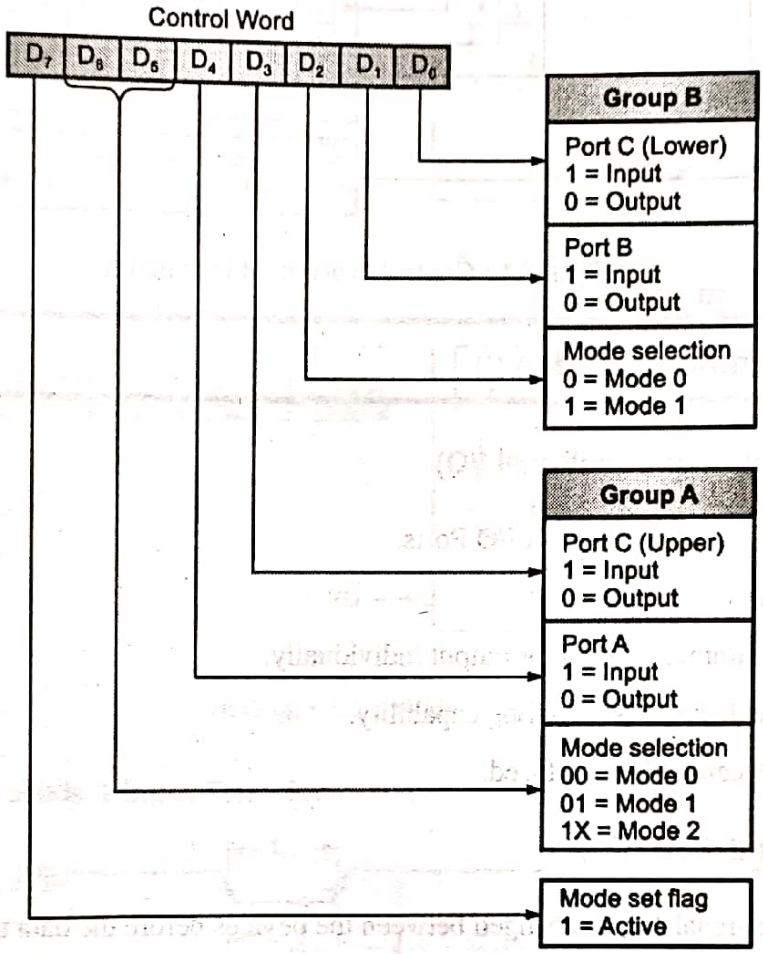
| Port | Mode 0 | Mode 1 | Mode 2 |
|---|---|---|---|
| Port A | Yes | Yes | Yes |
| Port B | Yes | Yes | No(Mode 0 or Mode 1) |
| Port C | Yes | No(Handshake signals) | No (Handshake signals) |

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

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

**Q.** Explain the I/O mode control word format of 8255 PPI. **(Dec. 18, 5 marks)**

- 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 :

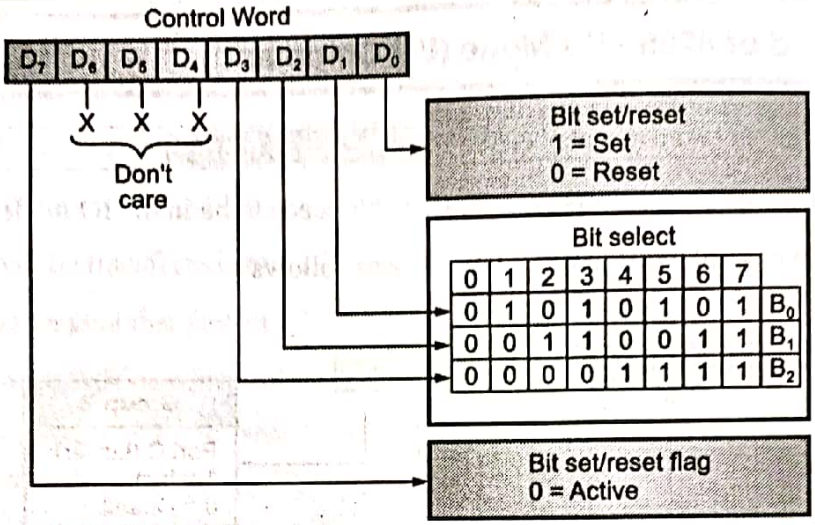



(1C2)**Fig. 6.4.1 : Control Word I/O Command**

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

**Q.** Discuss control word format for Bit Set Reset (BSR) mode of 8255 PPI. **(Dec. 17, 5 Marks)**

- 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.




(1C3)**Fig. 6.5.1 : Control Word BSR Command**

## 6.6 8255 I Data Transfer Modes

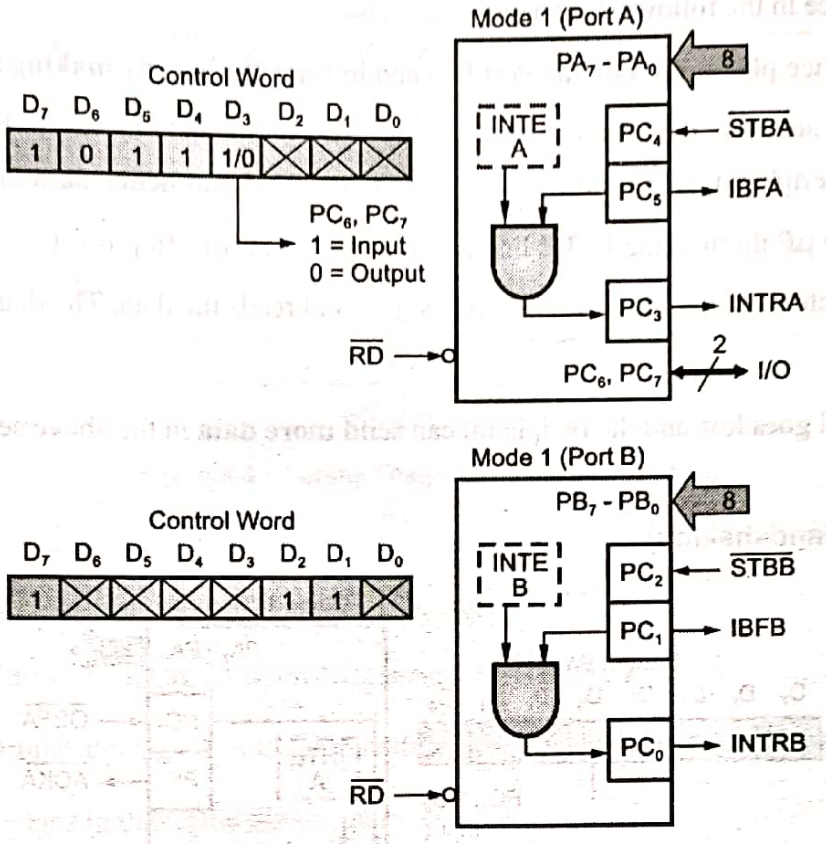
### 6.6.1 Mode 0 (Simple Bi-directional I/O)

- 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.

### 6.6.2 Mode 1 (Handshake I/O)

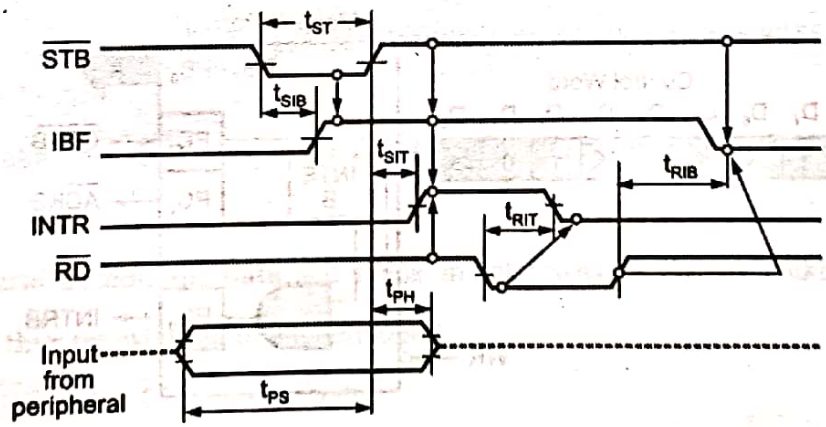
- In Mode 1, handshake signals are exchanged between the devices before the data transfer takes place.
- Port A and Port B used as 2 8-bit I/O Ports that can programmed in Input OR in output mode.
- Each Port uses 3 lines from Port C for handshake. The remaining lines of Port C can be used for simple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

## ☞ Mode 1 (Input Handshaking)




(1C4)**Fig. 6.6.1 : Mode 1 Input**

## ☞ Timing Diagram for Mode 1 Input Transfer




(1C5)**Fig. 6.6.2 : Timing Diagram for Mode 1 Input**

### 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.**

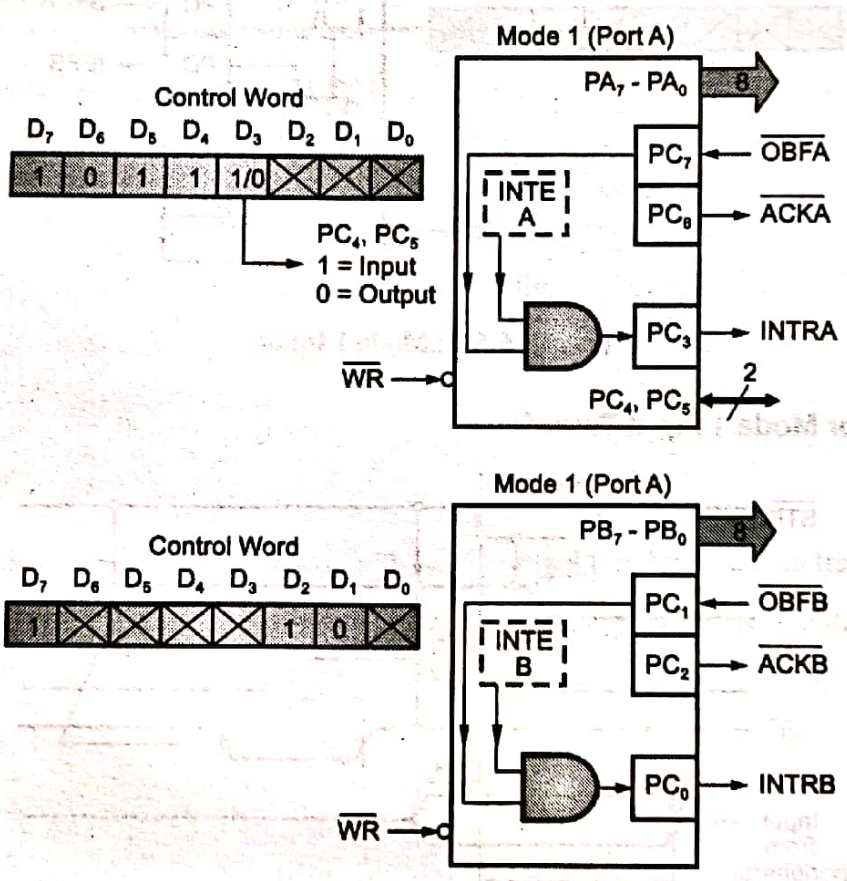
(2) The **input Port accepts** the **data** and informs the peripheral to wait by making **IBF high.** This **prevents** the peripheral from **sending more data** to the 8255 and **hence data loss** is prevented.

(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.

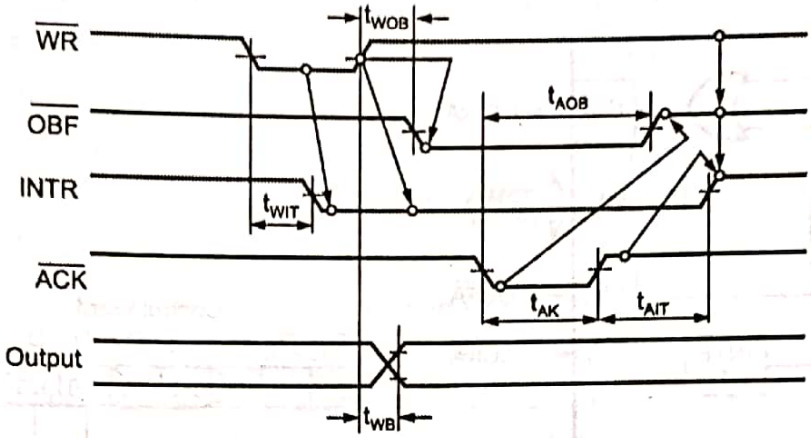
(5) Now, the **IBF** signal **goes low** and the peripheral can **send more data** in the above sequence.

### ☞ Mode 1 (Output Handshaking)




(1C8)**Fig. 6.6.3 : Mode 1 Output**

## ☞ Timing Diagram for Mode 1 Output Transfer




(1C7)**Fig. 6.6.4 : Timing Diagram for Mode 1 Output**

**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.

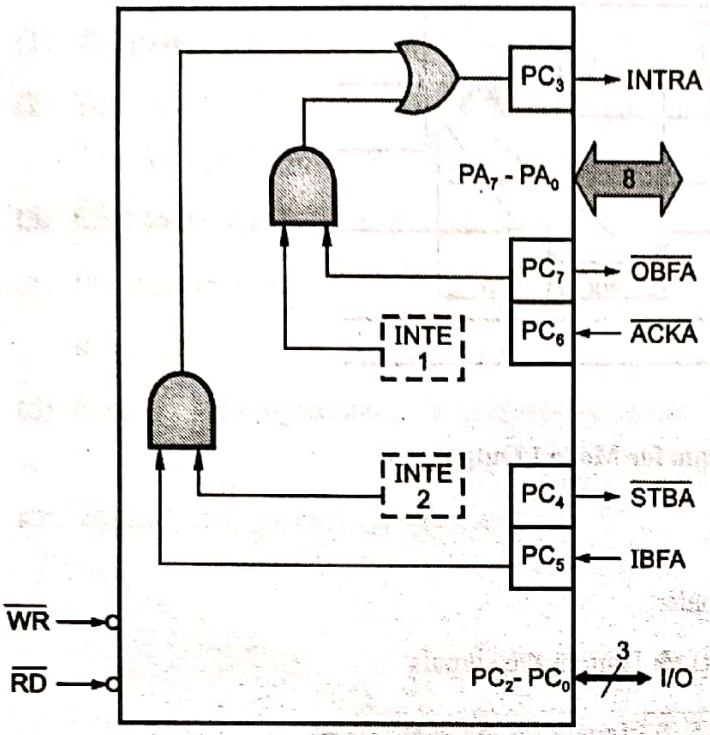
(4) The **peripheral accepts** the **data** and sends an acknowledgement by making the $\overline{\textbf{ACK}}$ **low**. The **data byte** is **thus transferred** to the peripheral.

(5) Now, the $\overline{\text{OBF}}$ and $\overline{\text{ACK}}$ lines go high.

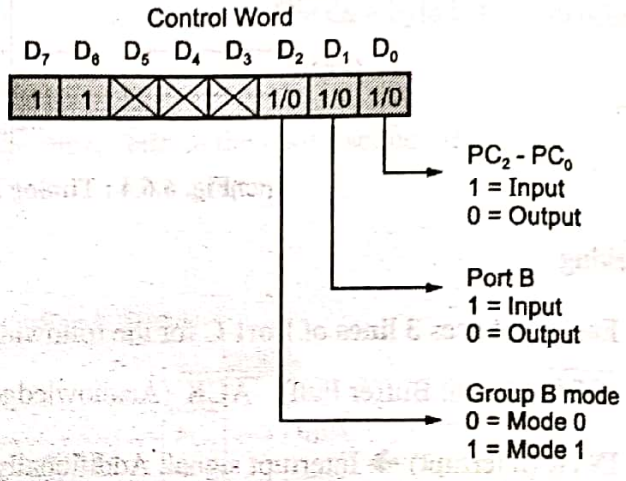
(6) The **INTR** line **becomes high** to **inform** the **µP** that **another byte** can be **sent.** i.e. the output port is empty.

This process is repeated for further bytes.

## 6.6.3 Mode 2 (Bi-directional Handshake I/O)

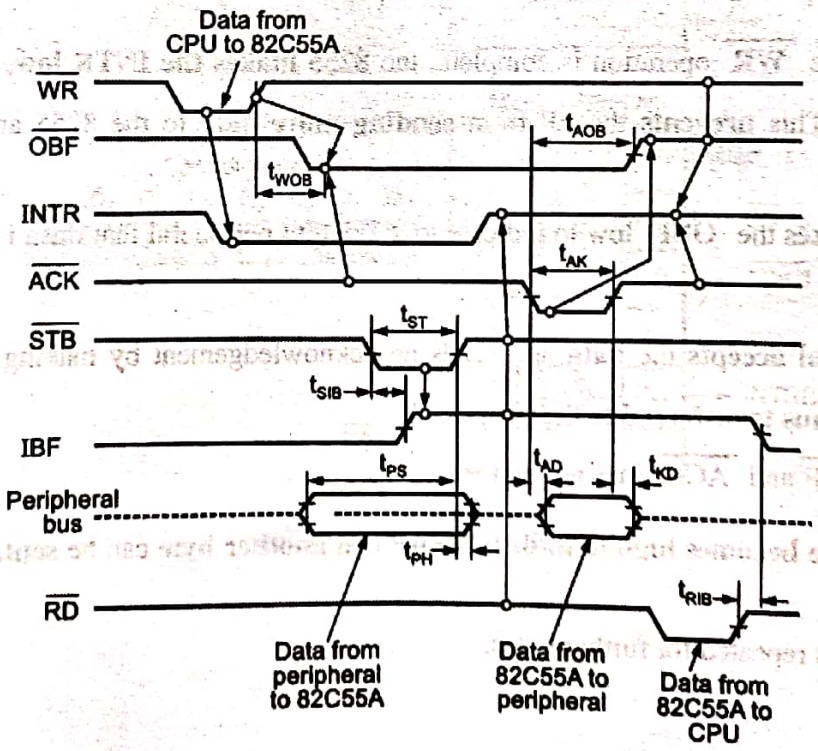



(1C8)Fig. 6.6.5 : Mode 2




(1C9)Fig. 6.6.6 : Control Word Mode 2

### Timing Diagram for Mode 2 Bi-Directional Transfer




(1C10)Fig. 6.6.7 : Timing Diagram for Mode 2

**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 two **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).

(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:**

$\overline{\text{OBF}}$ and $\overline{\text{ACK}}$ ➔ handshaking signals, INTR ➔ Interrupt signal.

Thus the 5 signals used from Port C are:

**STB, IBF, INTR, $\overline{\text{OBF}}$** and **$\overline{\text{ACK}}$** .

✓ **Syllabus Topic : Interfacing with 8086**

## 6.7 8255 | Interfacing 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.

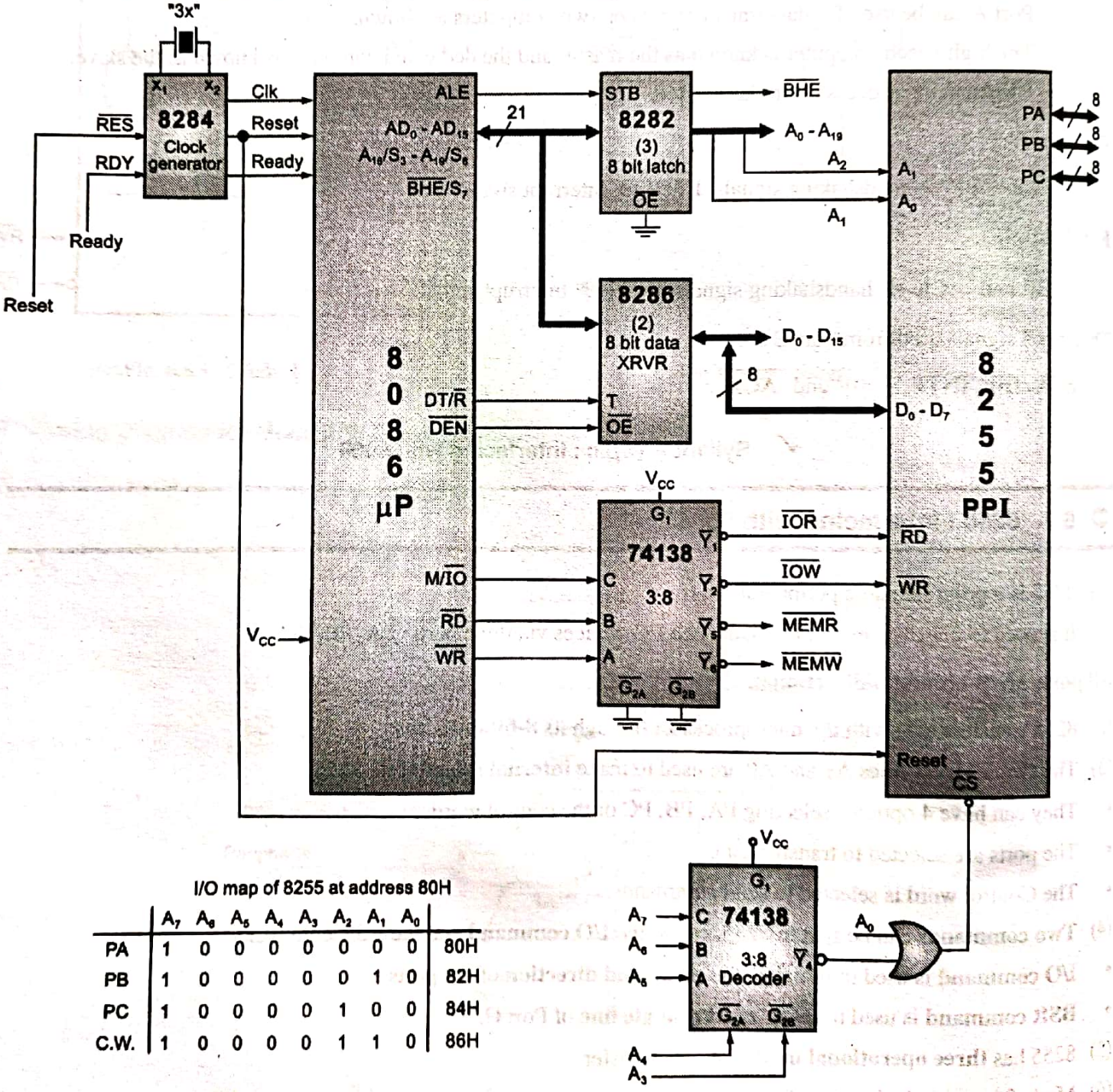
- It does not perform handshaking but all three ports are available for data transfer.

(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 address 80H

| | $A_7$ | $A_6$ | $A_5$ | $A_4$ | $A_3$ | $A_2$ | $A_1$ | $A_0$ | |
|---|---|---|---|---|---|---|---|---|---|
| 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 |

(1C11)**Fig. 6.7.1 : 8255 interfacing with 8086**

## 6.8 8255 I Programming

### 6.8.1 Program to Initialize 8255

**Q.** 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
```

### 6.8.2 Program to Generate Square Wave using 8255

**Q.** Generate a square wave on a display device connected to Port$C_3$ by BSR command.

**Soln. :**

```
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
```

### 6.8.3 Program to Generate Positive Spikes

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

**Soln. :**

```
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
```

### 6.8.4 Program to Generate Rectangular Wave

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

**Soln. :**

```
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
```

### 6.8.5 Program to Generate Staircase Waveform

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

**Soln. :**

```
        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
```

### 6.8.6 Program to Generate Ramp Waveform

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

**Soln. :**

```
        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.

### 6.8.7 Program to Generate Saw-tooth Waveform

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

**Soln. :**

```
        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.

### 6.8.8 Program to Generate Triangular Waveform

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

**Soln. :**

```
        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.

## 6.9 University Questions and Answers

**Dec. 2015**

**Q. 3(a)** Explain with block diagram working of 8255 PPI. *(Refer section 6.3)* **(10 Marks)**

**Dec. 2017**

**Q. 1(d)** Discuss control word format for Bit Set Reset (BSR) mode of 8255 PPI. *(Refer section 6.5)* **(5 Marks)**

**May 2018**

**Q. 2(b)** Explain PPI 8255 with block diagram. *(Refer section 6.3)* **(10 Marks)**

**Dec. 2018**

**Q. 4(b)(i)** Explain the I/O mode control word format of 8255 PPI. *(Refer section 6.4)* **(5 Marks)**

**May 2019**

**Q. 3(b)** Draw and explain the block diagram of 8255 Programmable Peripheral Interface (PPI) with control word formats. *(Refer section 6.3)* **(10 Marks)**

*Chapter ends...*

❑❑❑


# 8253 Programmable Interval Timer

## 7.1 8253 | Salient Features

**Q.** List features of 8253. **(Dec. 17, 5 Marks)**

(1) IC **8253/8254** 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.**

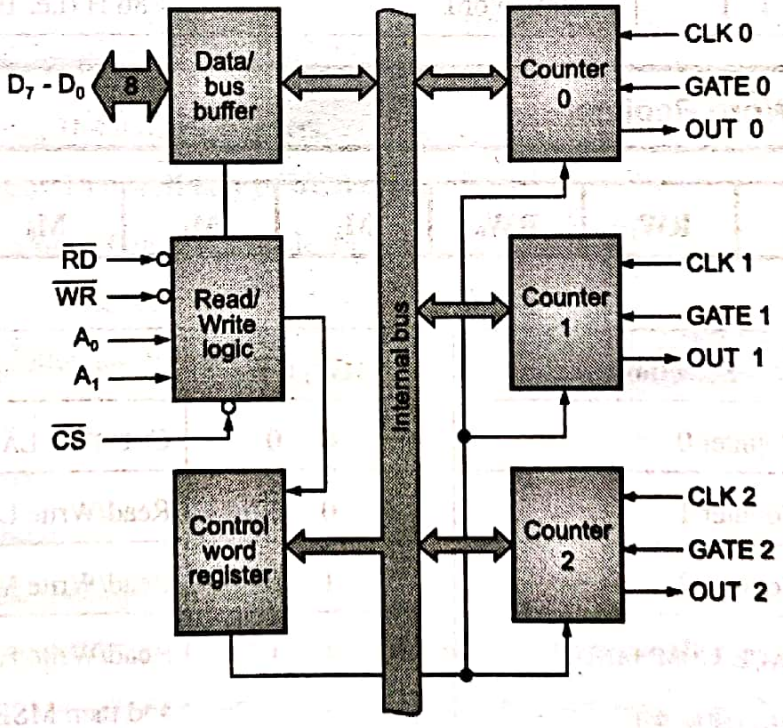
(6) Once the Counter finishes counting (required delay is produced), 8254 interrupts μP.

(7) 8253 and 8254 are both Timer ICs. The difference is 8254 additionally supports a READ BACK Command which is not available in 8253. Rest all is the same.

## 7.2 8253 | Architecture

**Q.** Draw and explain block diagram of 8253. **(Dec. 16, May 17, 10 Marks)**

**Q.** Explain with block diagram PIT 8253 **(May 18, 10 marks)**




(1C12)**Fig. 7.2.1 : 8253 Architecture**

### ☞ 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.

### ☞ 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 0000) |
| 0 1 | 0 1 | Counter 1 | 82 H (i.e. 1000 0010) |
| 1 0 | 1 0 | Counter 2 | 84 H (i.e. 1000 0100) |
| 1 1 | 1 1 | Control Word | 86 H (i.e. 1000 0110) |

## ➲ 7.3 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

### ☞ 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.**
- 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.**

## 7.4 8253 | Timer Modes

Q. Modes of 8253 Programmable Interval timer. **(May 19, 5 Marks)**

### 7.4.1 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

### 7.4.2 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.

The **GATE pin** can be used as a **Trigger.**

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.

### 7.4.3 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.

### 7.4.4 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.**

### 7.4.5 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

### 7.4.6 Mode 5 I 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.**

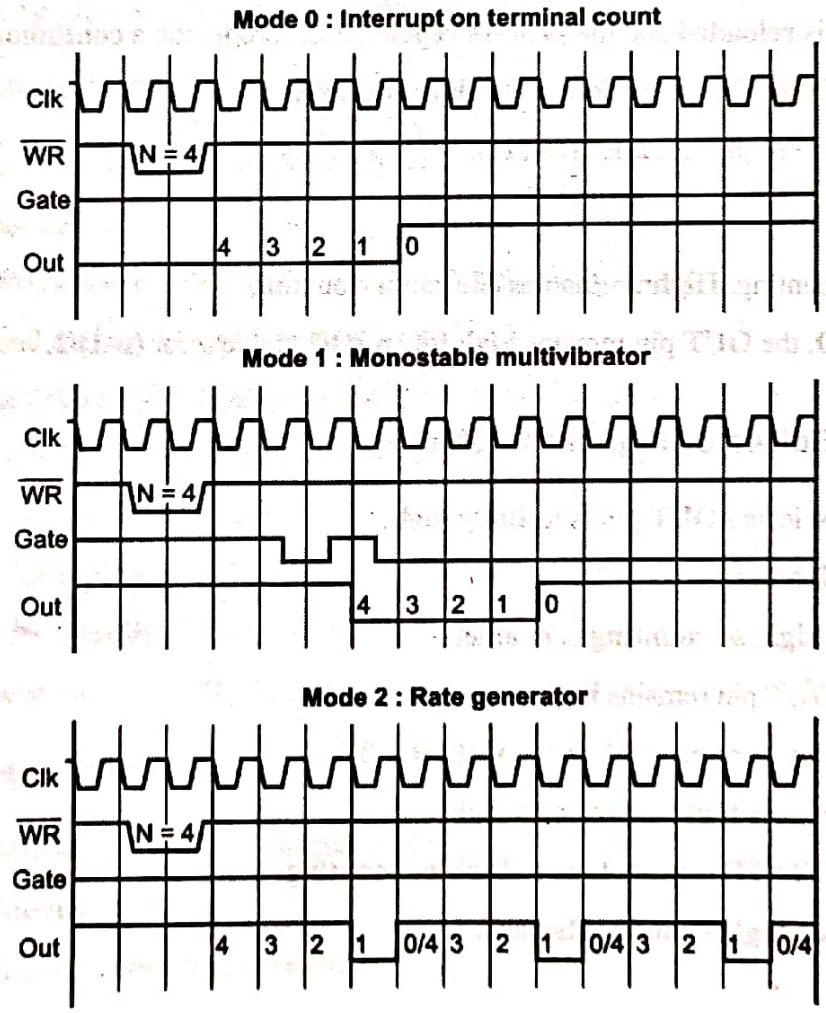
(6) The **OUT** pin goes **low** for **one** clock **cycle, just after TC.**

**(7) After that OUT** pin goes **high** and remains high.

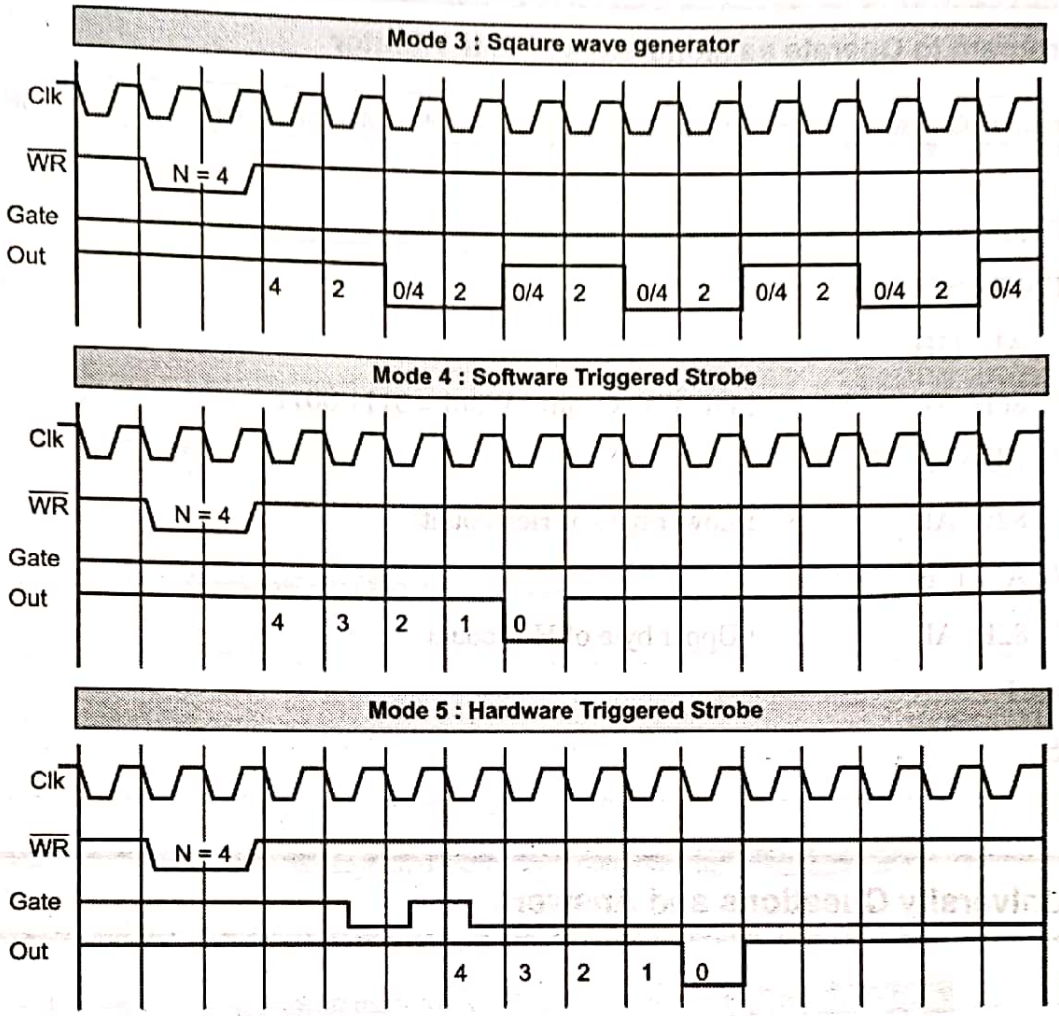
**(8) Thus GATE is used as a Trigger** i.e.it has to be triggered to start counting.

**Effect of Gate:**

**Low➔** No Effect; **High(Trigger)➔** Starts Counting, can also re-rtigger it.




(1C13)**Fig. 7.4.1 : Times modes : 0, 1, 2**




(1C14)**Fig. 7.4.2 : Times modes : 3, 4, 5**

## 7.5 8253 I Programming

### 7.5.1 Program to Generate Square Wave using 8253

**Q.** WAP to initialize 8253 for producing a square wave of 1 KHz from an input frequency of 1.625 MHz using Counter-0.

**Soln. :**

```
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
```

### 7.5.2 Program to Operate as Mono-stable Multi-Vibrator

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

**Soln. :**

```
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
```

## 7.6 University Questions and Answers

**Dec. 2016**

**Q. 2(b)** Draw and explain block diagram of 8253. *(Refer section 7.2)* **(10 Marks)**

**May 2017**

**Q. 2(b)** Draw and explain block diagram of 8253. *(Refer section 7.2)* **(10 Marks)**

**Dec. 2017**

**Q. 6(e)** List features of 8253. *(Refer section 7.1)* **(5 Marks)**

**May 2018**

**Q. 6(b)** Explain with block diagram PIT 8253. *(Refer section 7.2)* **(10 Marks)**

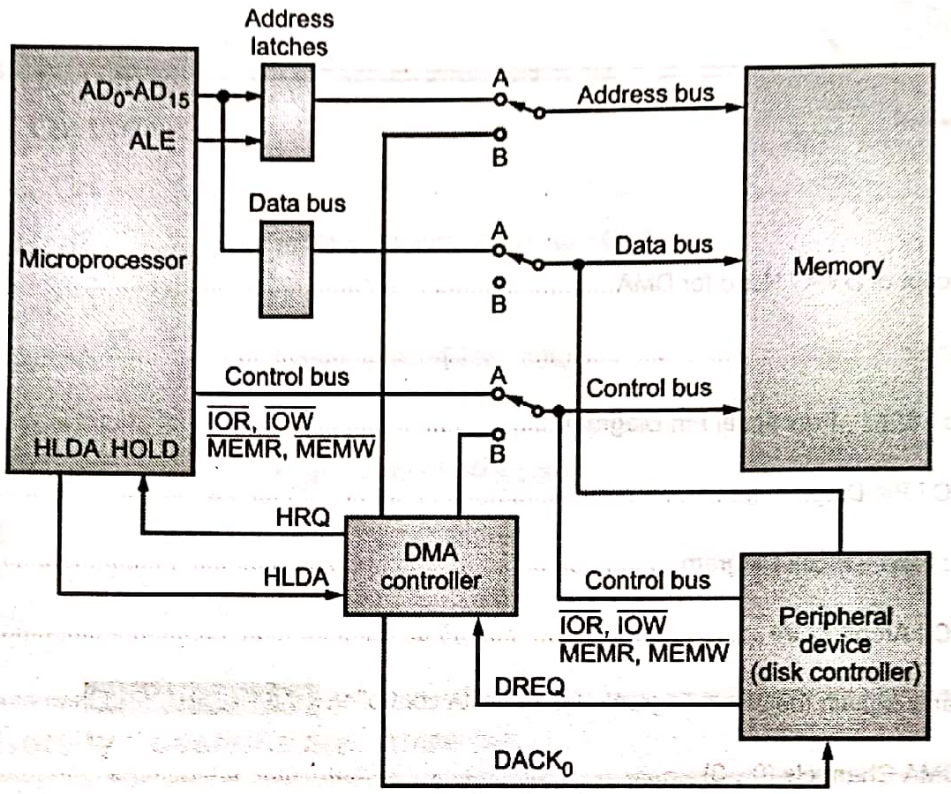
**May 2019**

**Q. 6(b)(II)** Write short note on Modes of 8253 Programmable Interval timer *(Refer section 7.4)* **(5 Marks)**

*Chapter ends...*


# 8257 DMA Controller

## 8.1 8257 I Concept of DMA / Need for DMA




(1D1)**Fig. 8.1.1 : Concept of DMA**

- DMA stands for Direct Memory Access.
- It means transferring data directly between memory and I/O without the involvement of µP.

### ☞ Steps for performing a DMA transfers

Although the transfer is mainly carried out by the DMAC, the initialization is first done by the µP.

The following steps are required for the complete DMA transfer.

**1. µP initializes the DMAC**

This is done by giving the starting address and the number of bytes to be transferred.

**2. I/O device requests the DMAC**

I/O device makes the DREQ signal = 1.

**3. DMAC requests the µP for control of the system bus**

DMAC makes HOLD = 1

**4. µP releases control of the system bus**

- µP finishes the current machine cycle and releases control of the system bus.
- µP informs the DMAC that the bus is released by making HLDA = 1.
- Now µP enters HOLD state.

**5. DMAC becomes the bus master.**

- On getting HLDA from µP, DMAC becomes the bus master.
- It informs the I/O device that DMA transfer is about to begin by activating the DACK signal.

**6. DMA Transfer begins**

- DMAC transfers data between memory and I/O, one byte in one cycle.
- After every cycle, the Address Register is incremented and the Count Register is decremented.
- This continues till the Count reaches zero (Terminal Count).
- Now the DMA transfer is completed.

**7. DMA Transfer ends**

- DMAC releases control of the system bus.
- It makes HOLD = 0.
- This makes µP come out of Hold state and once again become the bus master.
- µP takes control of the system bus and continues its operation.

### ☞ Advantage of DMA

DMA transfers are **extremely fast** due to two reasons

**1. Hardware based**

- Normally data transfers are performed by the µP.
- They are based on software and hence are also called Programmed I/O.
- These transfers are very slow as they involve fetching and decoding of instructions in a loop throughout the transfer process.
- In fact in most cases, more time is wasted in fetching and decoding of instructions than actually transferring the data.
- This is where DMA based transfers have a massive speed advantage.

- DMA transfers are performed by a dedicated hardware called a DMAC (IC 8257/ 8237).
- It is "hard-wired" to perform DMA transfers and hence doesn't need instructions to transfer every byte.
- This saves a lot of time which would have been otherwise wasted in fetching and decoding instructions.

**2. Direct transfer**

- Normally data transfers "via" the μP.
- This means if a byte has to be transferred from memory to I/O, it must first go from memory to μP and then from μP to I/O.
- This involves double the time as two machine cycles are needed: memory read and I/O write.
- In DMA transfers the byte is transferred "directly" from memory to I/O and vice versa.
- It does not need to travel via the μP and hence does not take "double" the time.

## 8.2 DMA Channels

- The DMAC does DMA transfer through its channels.
- DMAC has four channels: Channel 0 … Channel 3
- Each channel has four components :

**1. Address register**

- The address register holds the starting address for the transfer.
- μP gives this value to the DMAC.

**2. Count register**

- The count register holds the number of bytes to be transferred.
- μP gives this value to the DMAC.

**3. DREQ**

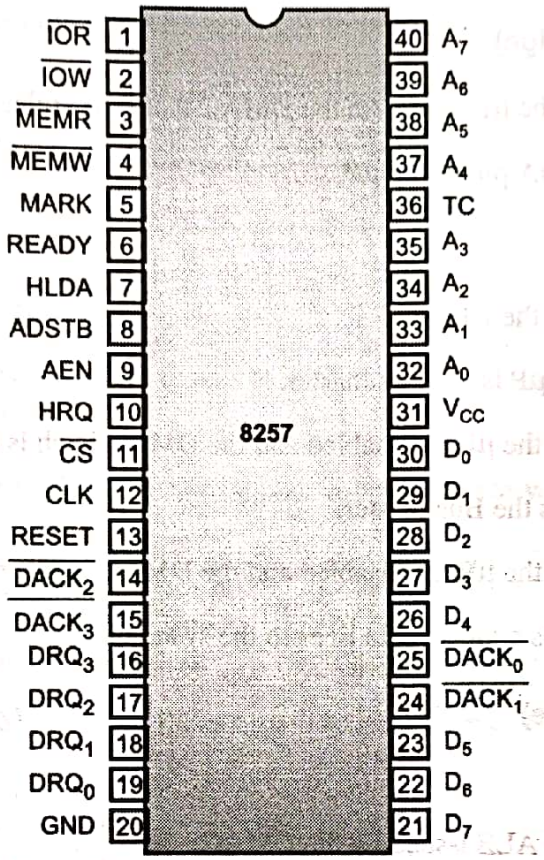
- This signal is used by the I/O device to request for a data transfer.
- I/O device gives this signal to the DMAC

**4. DACK**

- This signal is used to indicate that the DMA Transfer is about to begin.
- DMAC gives this signal to the I/O device.

✓ Syllabus Topic : 8257 - Functional Pin Diagram

## 8.3 8257 DMAC I Pin Diagram




(1D2)Fig. 8.3.1 : Pin diagram of 8257

### 1. $DREQ_3$ ... $DREQ_0$ (Data Request lines)

- 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 simultaneous requests, priorities are used.$DREQ_0$ has the highest priority while $DREQ_3$ has the lowest priority. (Fixed priority mode)

### 2. $\overline{DACK_3}$ ... $\overline{DACK_0}$ (Data Acknowledgement)

- These output pins are used by the DMAC to inform the I/O device that it is performing a data transfer.
- This pin becomes low just before a DMA data transfer.
- 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 has 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 AEN = 0 (default), µP is the Bus master.
- At that time the latches of the µP are enabled and the DMAC latch is disabled.
- When AEN = 1, DMAC is the Bus master.
- At that time the latches of the µP are disabled and the DMAC latch enabled.
- This connects the DMAC's address-data lines to the system bus.

### 6. ADSTB (Address Strobe)

- It is an output signal.
- To put it simply this is the ALE issued by the DMAC.
- When ADSTB = 1, the multiplexed bus ($DB_0$-$DB_7$) carries address ($A_8$-$A_{15}$).
- When ADSTB = 0, the multiplexed bus ($DB_0$-$DB_7$) carries data ($D_0$-$D_7$).

### 7. $DB_7 - DB_0$ (Data Bus)

- These are 8 bi-directional data lines used to connect the internal data bus of 8257 with the external (system) data bus.
- When µP is the bus master, this bus is used by the µP to read/write from the DMAC.
- In active cycle (which means DMAC is the bus master), 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 8257, 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 8257, 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 8257, these are output lines.

### 10. $\overline{IOR}$ , $\overline{IOW}$

- These are bi-directional control lines.
- During idle state, the µP issues these signals to read from or write into the 8257 (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. $\overline{MEMR}$ , $\overline{MEMW}$

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

### 12. Ready

- Similar function as Ready pin of the µP.
- It is used to synch DMAC with slower devices when DMAC is the Bus Master.
- DMAC checks this signal during every DMA transfer cycle.
- If Ready = 1, it means I/O device is ready.
- Hence DMAC will continue with the transfer.
- If Ready = 0, it means I/O device is not ready.
- Hence DMAC will insert Wait states during the transfer to allow the device to become ready.

### 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. TC

- This is an output signal from the DMAC.
- It becomes 1 to indicate that terminal count is reached, that means the count has become 0 and hence the transfer has been completed.

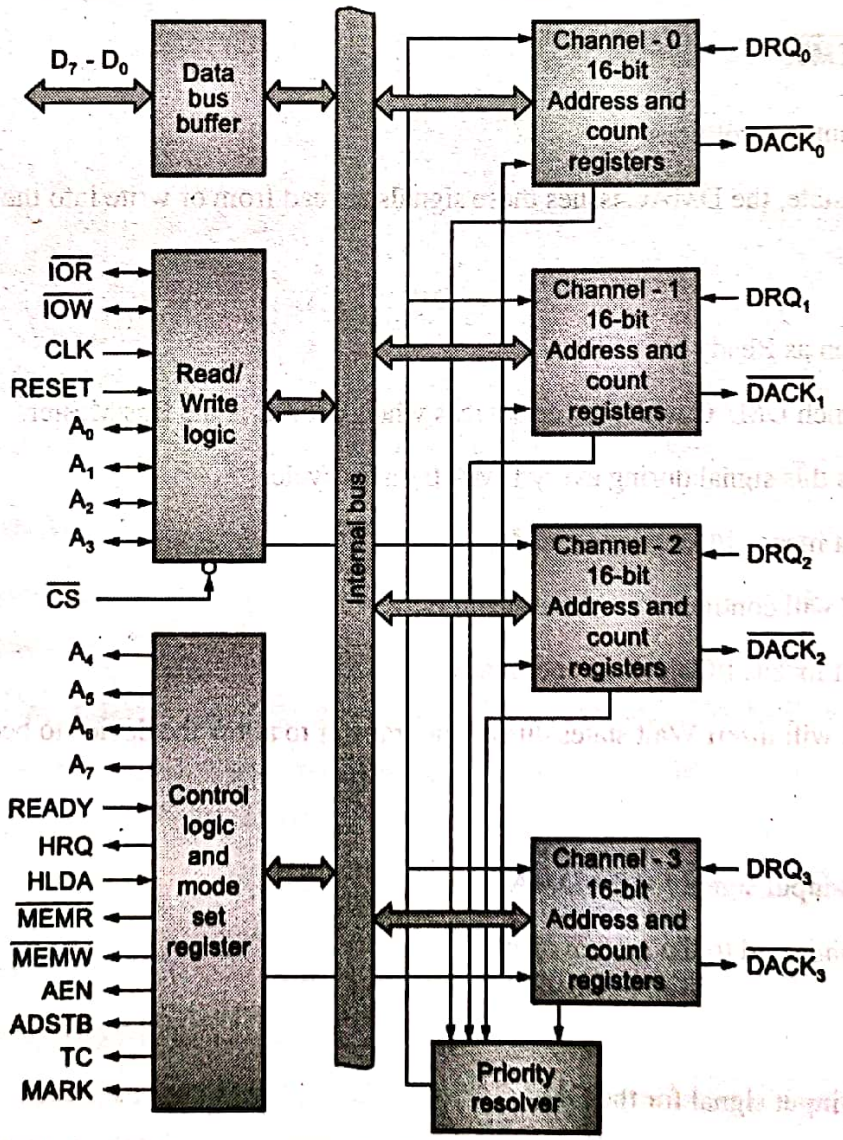
### 16. Mark

- This is an output signal from the DMAC.
- It is a modulo 128 mark output.
- This means it becomes 1 on every 128th cycle of the data transfer as a marker to indicate that a batch of 128 cycles has been completed.

✓ **Syllabus Topic : 8257 - Block Diagram**

## 8.4 8257 DMAC I Architecture

**Q.** Draw and explain the block diagram of 8257 DMA controller. **(Dec. 18, 10 Marks)**




(103)**Fig. 8.4.1 : Architecture of 8257 DMAC**

The internal architecture of 8257 has the following components

1. DMA Channels (0…3)
2. Priority Resolver
3. Data Bus Buffer
4. Read Write Logic
5. Control Logic and Mode Set Register

## 8.4.1 DMA Channels (0...3)

Q. Explain different data transfer modes of 8237 DMA controller. (Dec. 15, 10 Marks)

Q. Explain DMA data transfer modes in brief. (May 17, 10 Marks)

- A single 8257 DMAC has 4 DMA Channels (0…3)
- Four I/O devices are connected on these DMA Channels, one on each.
- In the default priority mode (Fixed priority), Channel 0 is the highest and 3 is the lowest priority.
- Each Channel has 4 components :

**Address Register, Terminal Count Register, DREQ and $\overline{\text{DACK}}$ .**

**a) Address Register (16 bit)**

- It is used to store the16 bit memory address for the DMA Transfer.
- µP initializes this register with the starting address of the DMA Transfer.
- Thereafter, as each byte is transferred the address gets incremented (or decremented, depending upon the mode selected by the programmer)

**b) Terminal Count Register (16-bit)**

- It is used to store the 14bit count of the DMA Transfer.
- The remaining higher two bits are used to decide the mode of DMA operation.
- µP initializes this register with the 14 bit count (N−1) of the DMA Transfer.
- Thereafter, as each byte is transferred the count gets decremented.
- This repeats till the count becomes 0 also called Terminal Count (TC).
- The higher two bits are used to select the mode.
- They can be used to give 3 different modes : DMA Verify, DMA Read, DMA Write

**DMA Read**

- In this mode, when DMAC becomes the bus master, it transfers data from
- Memory to I/O. Hence in every transfer, the signals produced are $\overline{\text{MEMR}}$ and $\overline{\text{IOW}}$

☞ **Count Register : 16 bit ($D_{15}$...$D_0$)**

| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| ← Low 8 bits of the 14 bit count → | | | | | | | |

| $D_{15}$ | $D_{14}$ | $D_{13}$ | $D_{12}$ | $D_{11}$ | $D_{10}$ | $D_9$ | $D_8$ |
|---|---|---|---|---|---|---|---|
| ← Mode bits (2) → | | | ← Higher 6 bits of the 14 bit count → | | | | |

| Mode | Bits | Mode Selected |
|---|---|---|
| 0 | 0 | DMA verify cycle |
| 0 | 1 | DMA Write Cycle (I/O to Memory) |
| 1 | 0 | DMA Read Cycle (Memory to I/O) |
| 1 | 1 | No action |

☞ **DMA Write**

In this mode, when DMAC becomes the bus master, it transfers data from I/O to Memory. Hence in every transfer, the signals produced are $\overline{\text{IOR}}$ and $\overline{\text{MEMW}}$

☞ **DMA Verify**

- In this mode, when DMAC becomes the bus master, it doesn't really transfer any data.
- This mode is just used to verify the DMA process.
- The DMAC will issue a HOLD, will become bus master, issue the acknowledgement to the I/O device and so on. It just will not produce any read or write signals to perform any data transfer.

**c) DREQ**

I/O device gives this signal to the DMAC to request a DMA transfer

**d) $\overline{\text{DACK}}$**

It is given by DMAC to the I/O device, indicating that a DMA transfer is being performed.

## 8.4.2 Priority Resolver

- Priority is needed when several DMA channels get request (DREQ) from I/O devices "simultaneously" for data transfer.
- Priority resolver decides which channel will be "serviced" first, and which one will become "pending". There are two priority schemes: Fixed Priority and Rotating Priority.

**Fixed Priority**

- This is the Default Mode.
- Channel 0 is the highest priority and Channel 3 is the lowest.
- Fixed priority causes Domination.
- If Channel 0 and 1 keep requesting all the time the Channel 2 and 3 will starve and never get a chance. This is called Domination.
- To avoid this, we can use Rotating Priority.

**Rotating Priority**

- Here, once a channel is serviced it becomes the lowest priority.
- All channels below it rise up by one position in the priority order.
- As priorities move in a circular manner, it is called Rotating Priority.
- It gives every channel a fair chance of being high priority and hence prevents Domination.

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 |

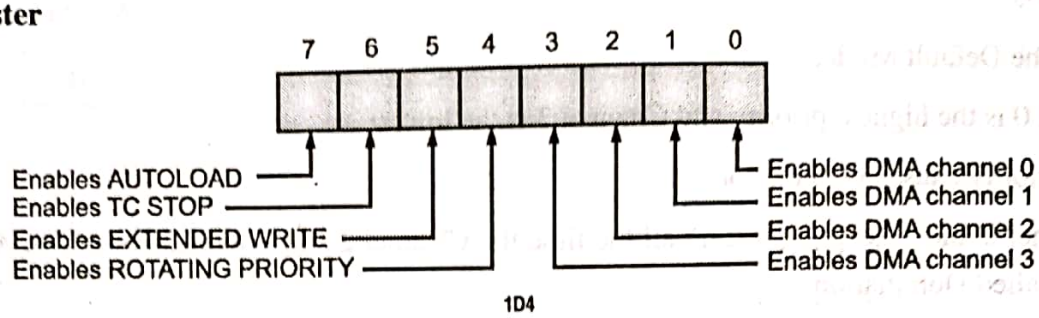
## 8.4.3 Read Write Logic

- It mainly provides the read and write signals as well as the chip select signal.
- The read and write signals are connected to $\overline{IOR}$, $\overline{IOW}$ of the µP.
- It also has address lines $A_3 \ldots A_0$ used for internal selection of components as explained earlier in the pin descriptions.

## 8.4.4 Control Logic and Mode Set Register

- It generates the internal control signals for the DMAC.
- It also provides external signals such as HRQ (Hold), HLDA, AEN, ADSTB, TC, MARK etc.
- as explained earlier.
- Additionally, it has the Mode Set register.

**Mode Set Register**




**Bit 0...3 : Channel Enables**

1 : Enable the respective DMA channel

0 : Disable

**Bit 4 : Rotating Priority**

1 : Rotating Priority

0 : Fixed Priority

**Bit 5 : Extended Write**

1 : Extended Write

0 : Normal Write

**Extended Write Mode**

- Here the Write control signal gets activated one T-State in advance.
- This is similar to the Advanced write signals of 8086 Maximum Mode.
- Once the Write signal gets activated (goes low), the I/O device has to respond by Making Ready Signal=1 to indicate that it is ready for the transfer.
- A slow device may require additional time to get ready and hence this will force the DMAC to insert Wait states in between every cycle, thus making the whole operation slow.
- To avoid this, we use the extended write mode.
- Here, the Write signal goes low one T-state in advance hence the I/O gets a little extra time to become ready. This reduces the chances of inserting Wait states and hence prevents a slightly slow I/O device from making the whole DMA operation slow.

**Bit 6 : TC Stop**

1 : Enable TC Stop mode

0 : Disable

**TC Stop Mode**

- In this mode, the DMA operation stops once terminal count is reached.
- The Respective DMA channel enable bit automatically becomes 0.

**Bit 7 : Auto load**

1 : Enable Auto Load Mode

0 : Disable

**Auto Load Mode**

- This is a continuous self-reloading mode.
- It is applicable only for Channel 2.
- When this mode is selected the original count and address register values
- of Channel 2 are stored as a back up in Channel 3 registers.
- After every byte is transferred, Channel 2 registers keep changing but Channel 3 registers maintain the original values.
- When Channel 2 reaches TC, There is an Automatic reload of address and count information from Channel 3 registers to Channel 2 registers and the DMA transfer restarts. This mode is basically used to perform repetitive DMA transfers.

### 8.4.5 Data Bus Buffer

It connects the external data bus of the system with the internal data bus of the DMAC, when the DMAC Chip is selected.

✓ **Syllabus Topic : DMA Operations**

## 8.5 8257 DMAC I Interfacing 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, microprocessor 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. $A_0$-$A_7$ are not multiplexed so they are directly connected to address bus.
18. $A_8$-$A_{15}$ and $D_0$-$D_7$ are multiplexed as $DB_0$-$DB_7$.

    They are given to the lower latch.
19. The ADSTB (Address Strobe, same as ALE), helps the latch to capture address that's $A_8 - A_{15}$.

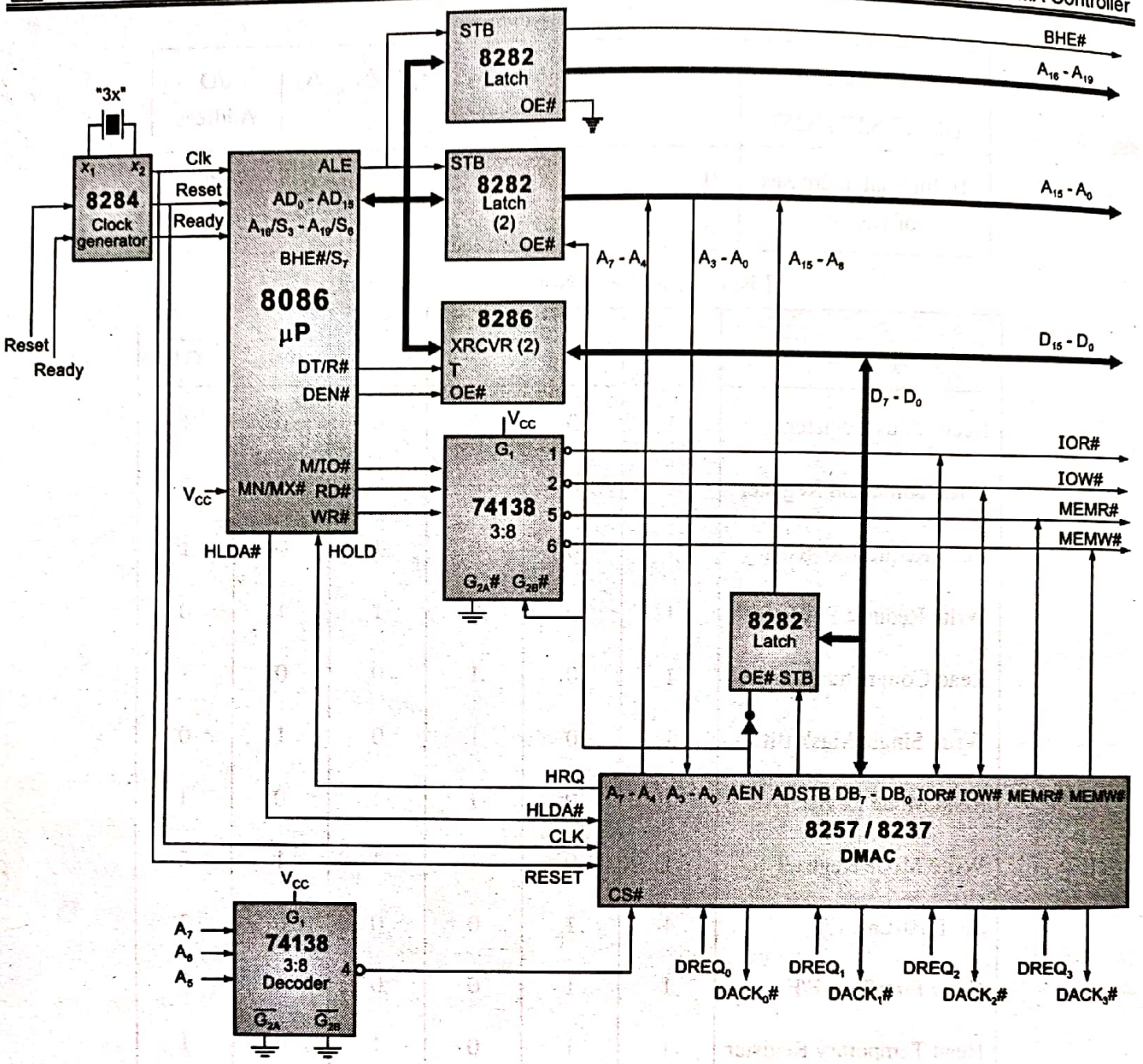
    This connects to the address bus.
20. The part not captured by the latch, carries data D0-D7.

    This is connected to the data bus.
21. $A_0$-$A_3$ 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 | $A_7$ | $A_6$ | $A_5$ | $A_4$ | $A_3$ | $A_2$ | $A_1$ | $A_0$ | 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 | $A_3$ | $A_2$ | $A_1$ | $A_0$ | $\overline{IOR}$ | $\overline{IOW}$ |
|---|---|---|---|---|---|---|
| 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 |




(1D5)**Fig. 8.5.1 : 8257 DMAC Interfacing with 8086**

## 8.6 University Questions and Answers

**Dec. 2015**

**Q. 5(a)** Explain different data transfer modes of 8237 DMA controller. *(Refer sections 8.4.1, 8.4.2 and 8.4.4)* **(10 Marks)**

**May 2017**

**Q. 3(a)** Explain DMA data transfer modes in brief. *(Refer sections 8.4.1, 8.4.2 and 8.4.4)* **(10 Marks)**

**Dec. 2018**

**Q. 3(b)** Draw and explain the block diagram of 8257 DMA controller. *(Refer section 8.4)* **(10 Marks)**

*Chapter ends...*


# 8086 Designing

## 9.1 8086 System Designing

**Q.** Design 8086 based system with following specifications : **(Dec. 15, 10 Marks)**

(i) 8086 in minimum mode working at 8MHz.

(ii) 32KB EPROM using 16KB devices.

(iii) 64KB SRAM using 32KB devices.

**Q.** Design 8086 based minimum mode system for following requirements : **(May 16, 12 Marks)**

(I) 256 KB of RAM using 64 KB x 8-bit device

(II) 128 KB of RAM using 64 KB x 8-bit device

(III) Three 8-bit parallel ports using 8255

(IV) Support for 8 interrupts

**Q.** Design 8086 based system for following requirements : **(Dec. 16, May 17, 10 Marks)**

(i) Clock frequency 5 MHz

(ii) 512 KB RAM using 32 KB × 8

(iii) 256 KB ROM using 32 KB × 8

**Q.** Design 8086 based system with following specifications. **(Dec. 17, 10 Marks)**

(i) 8086 is working in minimum mode at 10 MHz.

(ii) 8KB EPROM using 2KB chips.

(iii) 16KB SRAM using 8KB chips.

Discuss system with memory address map.

**Q.** Design 8086 based system for following specifications : **(Q. 2(b), Dec. 18, 10 Marks)**

(i) 8086 in minimum mode with clock frequency 5MHz.

(ii) 64KB EPROM using 16KB X 8 chips

(iii) 16KB RAM using 8KB X 8 chips

**Q.** Design 8086 based system for following specifications : **(May 19, 10 Marks)**

(i) 8086 in minimum mode with clock frequency 5MHz.

(ii) 128 KB EPROM using 32KB*8 chips

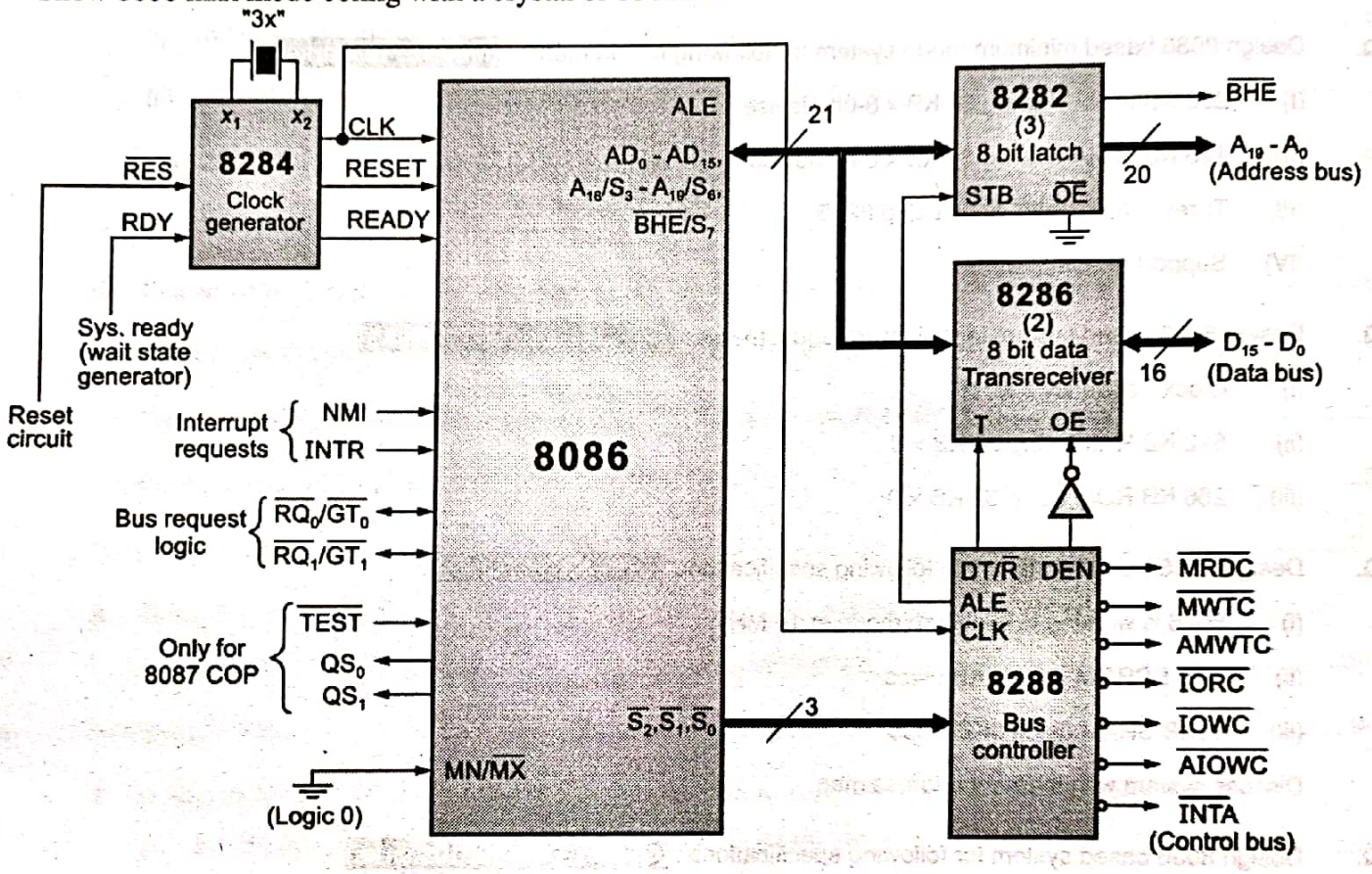
(iii) 32 KB RAM using 16KB*8 chips

## 9.1.1 Designing Problem 1

**Q.** 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 **(20 Marks)**

**Soln. :**

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




(1E1)**Fig. 9.1.1 : Maximum mode**

### ☞ Memory Calculations

**EPROM**

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)

FFFFFH
– 7FFFH
F8000H

Size of a single EPROM chip = 16 KB

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

= 14 address lines

= (A14... 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

$= 32 \times 1\ KB = 2^5 \times 2^{10} = 2^{15}$

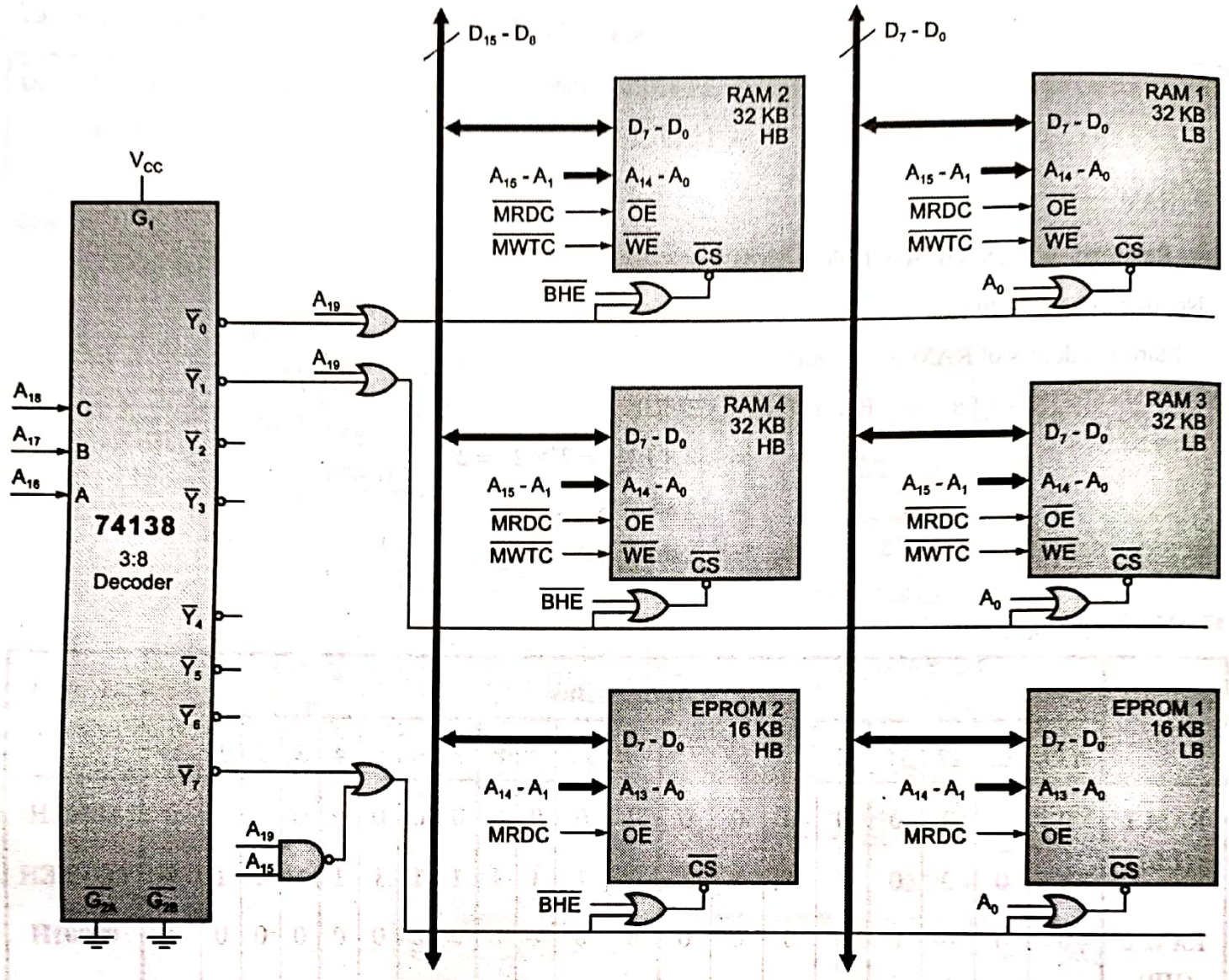
= 15 address lines

= **(A15 ... A1)**

☞ **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 (LB) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 00000H |
| | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | OFFFEH |
| RAM 2 (HB) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 00001H |
| | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | OFFFFH |
| RAM 3 (HB) | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 10000H |
| | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1FFFEH |
| RAM 4 (HB) | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 10001H |
| | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1FFFFH |
| EPORM 1 (LB) | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | F8000H |
| | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | FFFFEH |
| EPORM 2 (HB) | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | F8001H |
| | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | FFFFFH |

(1E2)

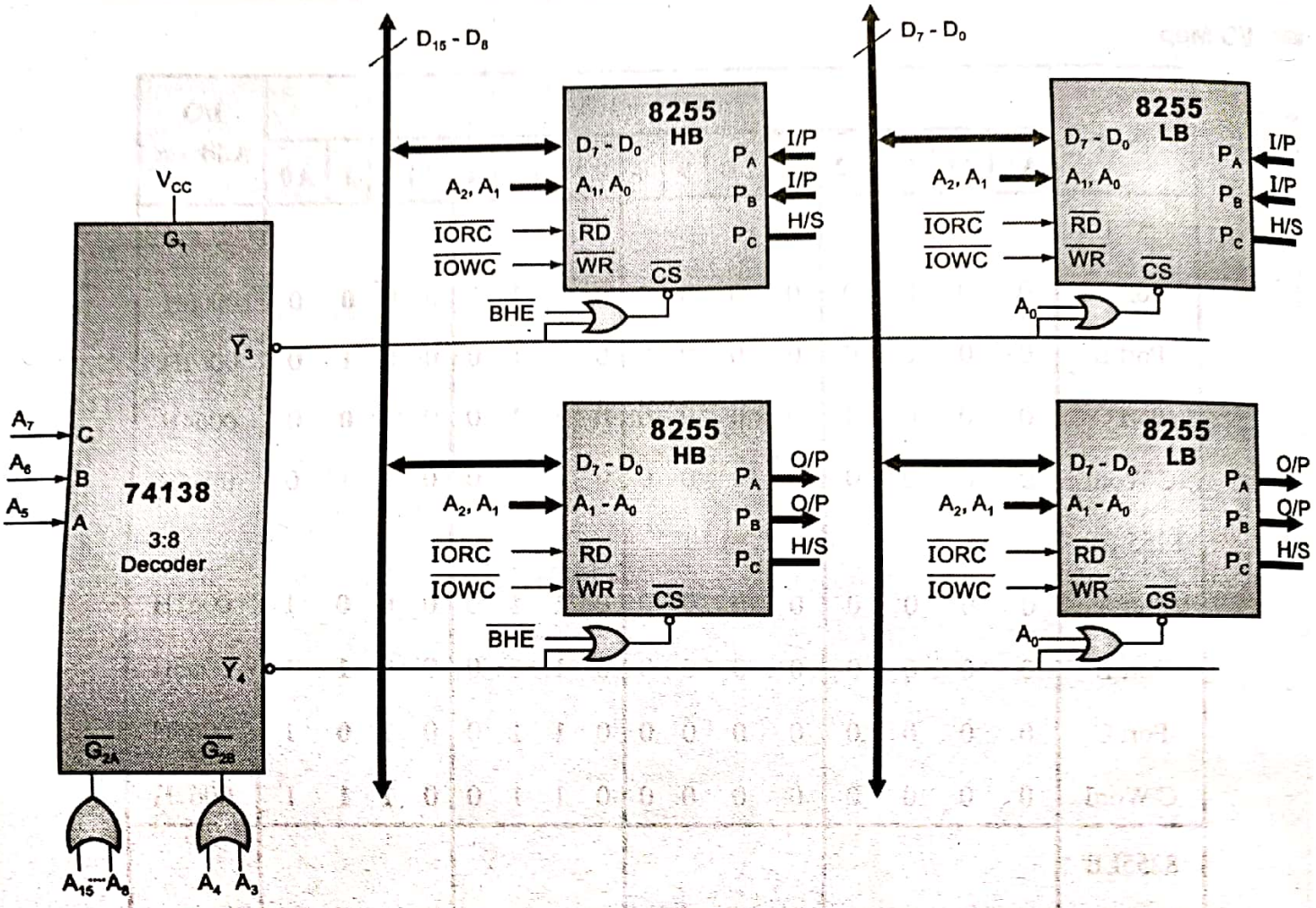



(1E3)**Fig. 9.1.2 : Memory diagram**

## I/O Map

| I/O Prot | 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 |
| 8255LB | | | | | | | | | | | | | | | | | |
| 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 |

(1E4)




(1E5)**Fig. 9.1.3 : I/O diagram**

☞ **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

## 9.2 Compare I/O MAPPED I/O and MEMORY MAPPED I/O

| Sr. No. | 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.** |

## 9.3 Designing Problem 2

**Q.** 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. :**

☞ **EPROM**

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; = ___ × 1KB;

= 2— × $2^{10}$ ; = 2— ;= _____ address lines = (A_____ ... A1)

☞ **RAM**

Required = _____ KB

Available = _____ KB

No. of chips = _____ chips.

Size of a single RAM chip = ____KB; = ___ × 1KB;

= 2— × $2^{10}$ ; = 2— ;= _____ address lines = (A_____ ... A1)

## Memory Map

| Memory CHIP | Address Bus 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | A0 | Memory 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 | | | | | | | | | | | | | | | | | | | | | |
| 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 |

## 9.4 Designing Problem 3

**Q.** 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. :**

### ☞ EPROM

Required = _____ KB

Available = _____ KB

No. of chips = _____ chips.

Starting address of EPROM is calculated as:

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

F FFFF H

- _____ H

_____ H

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

= 2— $\times 2^{10}$ ; = 2—;= _____ address lines = (A_____ ... A1)

### ☞ RAM

Required = _____ KB

Available = _____ KB

No. of chips = _____ chips.

Size of a single RAM chip = ____KB; = ___ $\times$ 1KB;

= 2— $\times 2^{10}$ ; = 2—;= _____ address lines = (A_____ ... A1)

## Memory Map

| Memory | Address Bus | | | | | | | | | | | | | | | | | | | Memory Address |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CHIP | 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 | 0000H |
| 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 2 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 |

## 9.5 Designing Problem 4

**Q.** 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. :**

### ☞ EPROM

Required = 16 KB

Available = 4 KB

No. of chips = 4 chips.

Starting address of EPROM is calculated as:

FFFFFH – (Space required by total EPROM of 16 KB)

F F F F F H

\- 3 F F F H

F C 0 0 0 H

Size of a single EPROM chip = 4KB; = 4 × 1KB;

= $2^2 \times 2^{10}$ ; = $2^{12}$;= 12 address lines = **(A12... A1)**

### ☞ RAM

Required = 64 KB

Available = 8 KB

No. of chips = 8 chips.

Size of a single RAM chip = 8KB; = 8 x 1KB;

= $2^3 \times 2^{10}$ ; = $2^{13}$; = 13 address lines = **(A13... A1)**

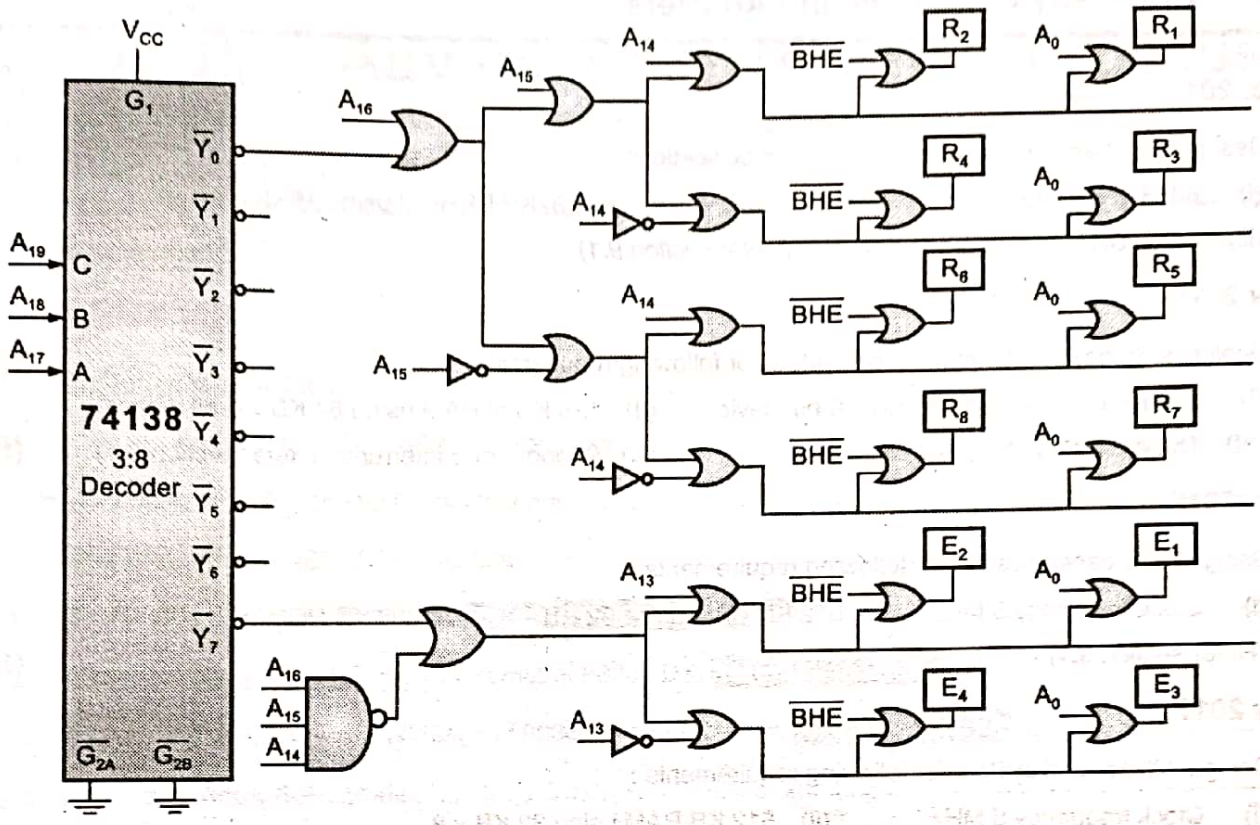
## Memory Map

| Memory CHIP | Address Bus | | | | | | | | | | | | | | | | | | | Memory 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 | 1 | 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 |

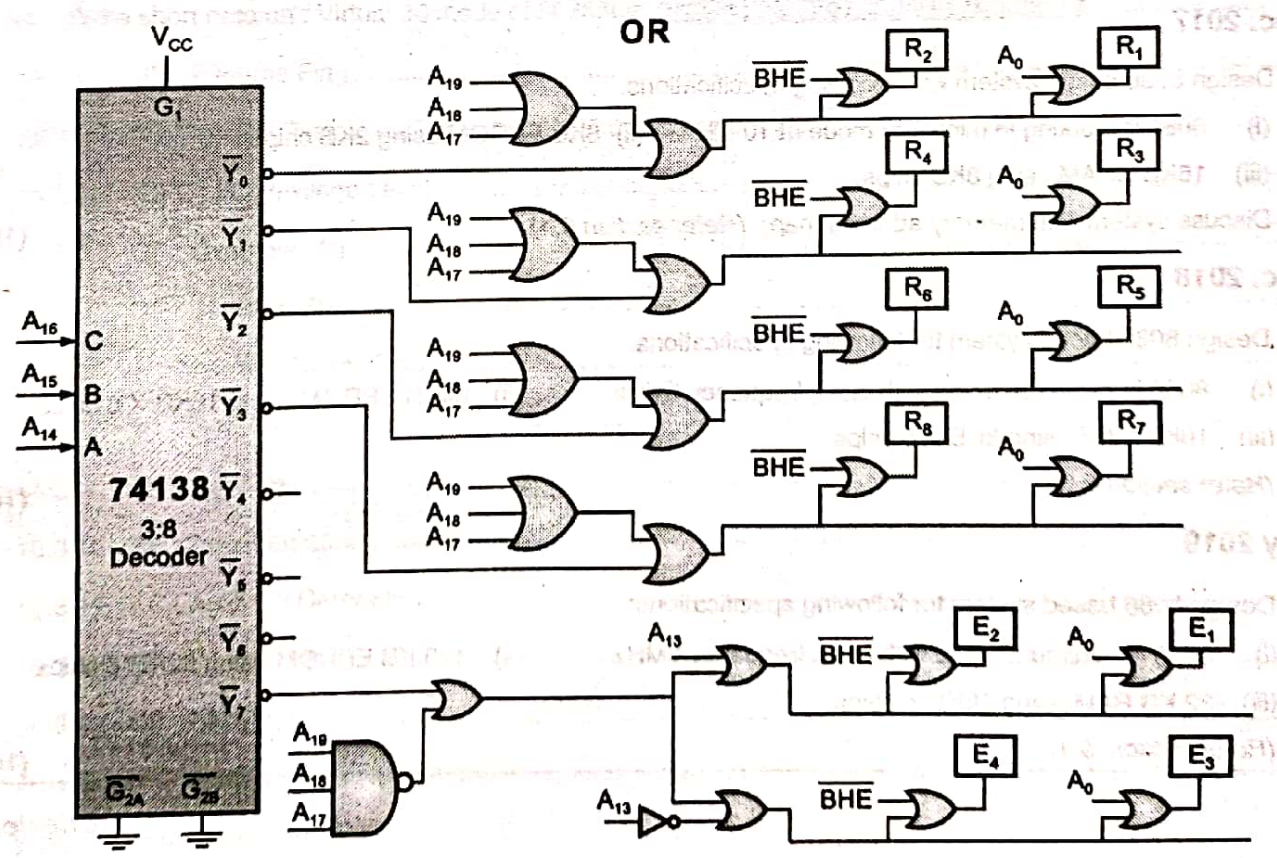
| Memory CHIP | Address Bus | | | | | | | | | | | | | | | | | | | | Memory Address |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | A0 | |
| RAM 7 Begin | 0 | 0 | 0 | 0 | 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 |

(1E8)

## Two Equally Correct Decoding Solutions




(1E9)**Fig. 9.5.1 : Solution 1**




(1E10)**Fig. 9.5.2 : Solution 2**

## 9.6 University Questions and Answers

### Dec. 2015

**Q. 2(b)** Design 8086 based system with following specifications :

(i) 8086 in minimum mode working at 8MHz. (ii) 32KB EPROM using 16KB devices.

(iii) 64KB SRAM using 32KB devices. *(Refer section 9.1)* **(10 Marks)**

### May 2016

**Q. 3(a)** Design 8086 based minimum mode system for following requirements:

(I) 256 KB of RAM using 64 KB x 8-bit device (II) 128 KB of RAM using 64 KB x 8-bit device

(III) Three 8-bit parallel ports using 8255 (iv) Support for 8 interrupts (Refer section 9.1) **(12 Marks)**

### Dec. 2016

**Q. 2(a)** Design 8086 based system for following requirements :

(i) Clock frequency 5 MHz (ii) 512 KB RAM using 32 KB × 8 (iii) 256 KB ROM using 32 KB × 8

*(Refer section 9.1)* **(10 Marks)**

### May 2017

**Q. 2(a)** Design 8086 based system for following requirements :

(i) Clock frequency 5 MHz (ii) 512 KB RAM using 32 KB × 8

(iii) 256 KB ROM using 32 KB × 8 *(Refer section 9.1)* **(10 Marks)**

### Dec. 2017

**Q. 5(a)** Design 8086 based system with following specifications.

(i) 8086 is working in minimum mode at 10 MHz. (ii) 8KB EPROM using 2KB chips.

(iii) 16KB SRAM using 8KB chips.

Discuss system with memory address map. *(Refer section 9.1)* **(10 Marks)**

### Dec. 2018

**Q. 2(b)** Design 8086 based system for following specifications:

(i) 8086 in minimum mode with clock frequency 5MHz. (ii) 64KB EPROM using 16KB X 8 chips

(iii) 16KB RAM using 8KB X 8 chips

*(Refer section 9.1)* **(10 Marks)**

### May 2019

**Q. 4(b)** Design 8086 based system for following specifications:

(i) 8086 in minimum mode with clock frequency 5MHz. (ii) 128 KB EPROM using 32KB*8 chips

(iii) 32 KB RAM using 16KB*8 chips

*(Refer section 9.1)* **(10 Marks)**

*Chapter ends...*

❑❑❑


# Advanced 32-Bit Microprocessor 80386

## 10.1 Salient Features of 80386

### ☞ Address Bus

**80386 has a "32 bit" address bus.**

- This means it can access a total of $2^{32}$ = 4GB of physical memory.
- The memory has an address range of 0000 0000H … FFFF FFFFH.

| Memory Address | Data |
|---|---|
| 0000 0000 h | 8-bit |
| 0000 0001 h | 8-bit |
| 0000 0002 h | 8-bit |
| 0000 0003 h | 8-bit |
| --- | --- |
| FFFF FFFF h | 8-bit |

- Though the total address bus is of 32 bits, only the higher 30 bits from $A_{31} - A_2$ are released by the µP.
- The lower 2 lines $A_1$ and $A_0$ are used internally by the µP to produce the four bank-enable signals $\overline{BE_3}$ … $\overline{BE_0}$

### ☞ Data Bus

- **80386 has a "32-bit" data bus.** This means 80386 can transfer 32-bit data at a time.
- It also has a 32-bit ALU, which means 80386 can operate on 32-bit numbers in one cycle.
- **Hence 80386 is called a "32-bit µP".**
- 32-bit data is stored in 4 consecutive locations.
- To transfer 32-bit data in one operation 80386 memory is divided into 4 banks of 1 GB each. The banks are enabled by 4 bank-enable signals: $\overline{BE_3}$ … $\overline{BE_0}$ produced by the µP.

### ☞ Address Pipelining

- 80386 performs address pipelining, by putting address of the next machine cycle on the address bus, during T2 state of the current machine cycle.
- This makes the decoder delay transparent and is especially useful for interfacing slower devices as it reduces the number of wait states.

### ☞ Virtual Memory

- 80386 supports Virtual Memory which is implemented using Segmentation and Paging.
- It can access a total Virtual Memory of 64 TB ($2^{46}$).

### ☞ Protection

80386 use a protected model for accessing both memory and I/O. It uses 4 Privilege Levels.

### ☞ Multitasking

- 80386 allows multitasking using timesharing.
- Here several tasks can execute simultaneously by taking a small time slice of the μP. this gives higher system performance.

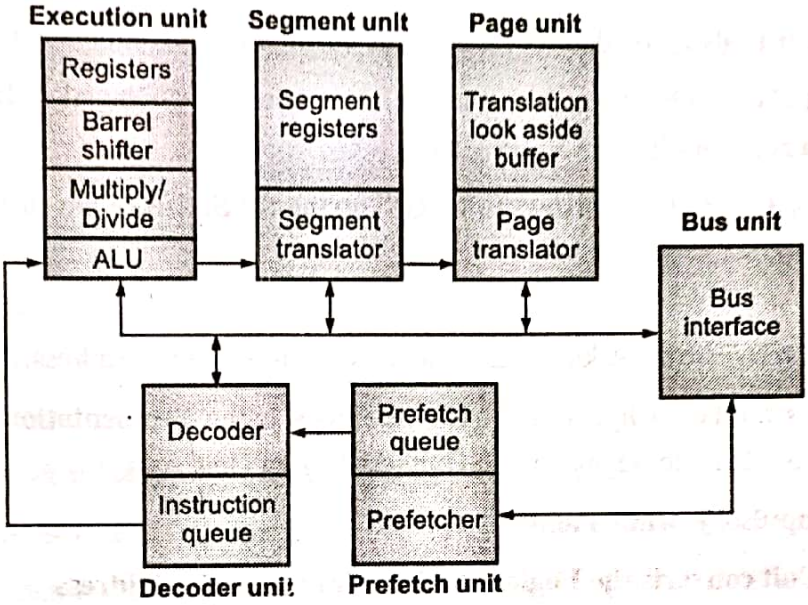
### ☞ I/O Addressing

80386 uses a 16-bit I/O address and hence can access up to $2^{16}$ i.e. 65536 I/O devices with address 0000 h … FFFF h.

| Sr. No. | 80386 DX | 80386 SX |
|---|---|---|
| 1. | 80386 DX has a 32 bit data bus. | 80386 SX has a 16 bit data bus. |
| 2. | Due to 32 bit data bus, the execution speed is higher.<br>Hence the name: DX – Double Execution speed. | Due to 16 bit data bus, the execution speed is lower.<br>Hence the name: SX – Single Execution speed. |
| 3. | 32-bit transfers require 4 Memory Banks. | 16-bit transfers require 2 Memory Banks. |
| 4. | Has 4 Bank enable signals:<br>$\overline{BE_3}$ , $\overline{BE_2}$ , $\overline{BE_1}$ , $\overline{BE_0}$ . | Has only 2 Bank enable signals:<br>$\overline{BHE}$ and $\overline{BLE}$ . |
| 5. | 4 Bytes are fetched at once in the pipelining queue. | 2 Bytes are fetched at a time in the pipelining queue. |
| 6. | Has dynamic data bus sizing of<br>16-bit and 32-bit data bus, using $\overline{BS16}$ signal. | No such option available as the data bus is only of 16-bits.<br>Hence $\overline{BS16}$ signal not useful. |
| 7. | Used for high performance. | Used for low cost memory and I/O system design. |
| 8. | Comes in a 132-pin ceramic PGA (Pin Grid Array) package for higher performance. | Comes in 100 lead plastic quad flat packages (PQFP) to permit lower cost. |

✓ Syllabus Topic : Architecture of 80386 Microprocessor

## 10.2 80386 Architecture / Functional Block Diagram




(1F1) **Fig. 10.2.1 : Functional Block Diagram of 80386**

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

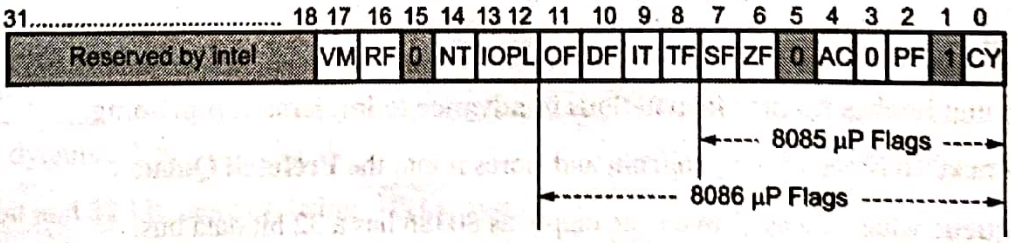
1. The Memory unit converts Virtual Address (Logical address) to Physical Address.
2. 80386 µP implements **64 Terra bytes of Virtual memory using Segmentation and Paging.** Hence the Memory Unit is sub-divided into Segmentation Unit and Paging Unit.
3. **Segmentation is compulsory, while Paging is optional.**
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.

## ➲ 10.3 Flag Register of 80386 (EFLAGS – 32 bits)

**Q.** Explain flag register of 80386DX. (Dec. 16, Dec. 18, 5 Marks)

**Q.** Draw and explain EFLAG register format of 80386 DX. (Dec. 17, 10 Marks)

**Q..** Explain VM, RF, IOPL and NT flags of 80386 microprocessor. (May 19, 5 Marks)




(1F2)**Fig. 10.3.1 : E Flag Register of 80386**

- As seen from the above diagram, the lower 12 bits (11 … 0) of EFLAGS are same as those in 8086. These are the only flags available when µP is in Real Mode.
- The additional 5 flags are only available once µP enters Protected mode by making PE bit = 1, in CR0 register.

### 10.3.1 VM : Virtual 8086 Mode

Q. Explain V86 mode of 80386DX. (May 16, 5 Marks)

Q. Write short note on : Virtual 86 mode of 80386DX (Dec. 16, May 17, May 18, 5 Marks)

This flag is used to make 80386 operate in Virtual 8086 Mode (V86).

**If VM = 1, enter Virtual 8086 Mode.**

V86 mode is basically used to run 8086 programs in a faster environment of 80386 using Multitasking and Protection.

**V86 mode can only be entered if µP is working in Protected Mode.**

Once in Virtual 8086 Mode, we can return back to Protected Mode by making VM ← 0.

A special program called Virtual 8086 Monitor is responsible for switching back and forth between Protected Mode and Real Mode.

### 10.3.2 RF : Resume Flag

**Resume flag is useful during debugging.**

**If RF = 1, then any debug fault in the next instruction will be ignored. RF is automatically reset after the next instruction.**

In 80386 µP, some fault handlers (ISRs) return back to the same instruction that caused the fault instead of returning back to the next instruction. By keeping RF = 1, we ensure that the program resumes after such a fault instead of repeatedly generating breakpoint faults on the same instruction.

### 10.3.3 NT : Nested Task

**NT flag is used to indicate that the current task is nested i.e. it is invoked by another task.**

**If NT =1 then, the current task is Nested and its TSS (Task State Segment) has a valid "back link" to the TSS of the previous task.**

This bit is automatically set whenever a Nested task is initiated, and thereafter can only be reset by software.

NT flag is checked by the IRET instruction to know whether it should perform an "Intra-task" return or an "Inter-task" return.

### 10.3.4 IOPL : I/O Privilege Level

**These bits are used to assign I/O Privilege Level.**

80386 µP has four privilege levels used for protection mechanism.

Privilege Level = 0 is the highest Privilege Level and 3 is the lowest.

**IOPL bits define the numerically maximum Privilege Level (logically lowest) at which a task must be running to access I/O devices.**

If IOPL bits = 00 then only highest privileged tasks running at PL=0 can perform I/O instructions.

If IOPL bits =11 then all tasks at any Privilege Level can perform I/O instructions.

### 10.3.5 OF : Overflow Flag

**It is used to indicate an overflow during a signed operation.**

OF is determined as an Ex-Or between the last and the second last carry.

For an 8-bit operation, OF = $C_7$ Ex-Or $C_6$.

### 10.3.6 DF : Direction Flag

**It is used to give the direction during String operations.**

If DF = 1 then, auto decrement. If DF = 0 then, auto increment.

### 10.3.7 IF : Interrupt Enable Flag

**It is used to enable the hardware interrupt INTR.**

When IF = 1, INTR is enabled. When IF = 0, INTR is disabled.

### 10.3.8 TF : Trap Flag

**It is used to perform Single Stepping.**

If TF = 1 then perform Single Stepping.

If TF = 1 then stop/ do not perform Single Stepping.

### 10.3.9 SF : Sign Flag

**It is used to indicate the MSB of the result.**

If SF = 1 then, MSB of the result is 1, so the number is treated as –ve.

If SF = 0 then, MSB of the result is 0, so the number is treated as +ve.

### 10.3.10 ZF : Zero Flag

**It is used indicate if the result has become zero.**

If ZF = 1 then the result is zero.

If ZF = 0 then, the result is non-zero.

### 10.3.11 AC : Auxiliary Carry Flag

**It indicates a Carry from the lower nibble to the higher nibble.**

If AC = 1 then, there was a carry from Lower Nibble to Higher Nibble.

If AC = 0 then, there was no carry from Lower Nibble to Higher Nibble.

Even during a 16-bit or a 32-bit operation AC only indicated the carry from the lowest nibble to the next nibble.

### 10.3.12 PF : Parity Flag

**Indicates whether the result has even or odd parity.**

If PF = 1 then, result has even parity.

If PF = 0 then, result has odd parity.

### 10.3.13 CF : Carry Flag

**It indicates if a carry was produced beyond the MSB of the result.**

If CF = 1 then, Carry produced beyond the MSB of the result.

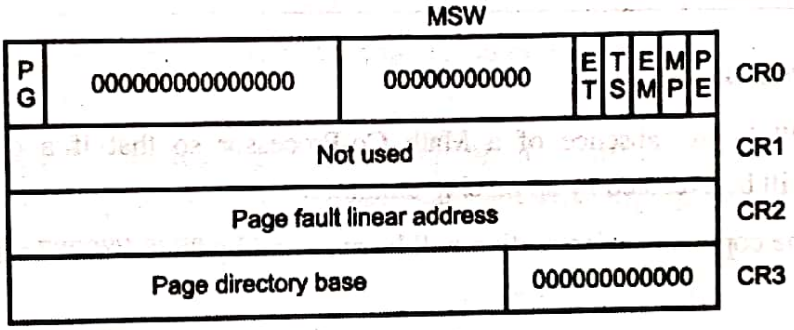
If CF = 0 then, Carry not produced beyond the MSB of the result.

✓**Syllabus Topic : 80386 Control Registers**

## 10.4 80386 I Control Registers

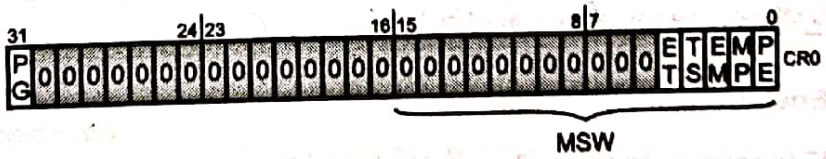
**Q.** Write short note on :Control registers of 80386DX (Dec. 16, May 17, 5 Marks)

There are four; 32-bit Control registers in 80386.




(1F3)**Fig. 10.4.1 : Control registers**

### 10.4.1 Control Register 0




(1F4)**Fig. 10.4.2 : CR0**

The various bits of CR0 are explained as follows:

#### PG : Paging Enable

- This bit is made "1" to enable paging mode.
- 80386 μP implements Virtual Memory using the techniques of segmentation and paging. Though segmentation is compulsory, paging is optional. Paging is enabled using the PG bit of CR0. The default value of PG bit is "0".

### ☞ ET : Extension Type

This bit is used to indicate the type of Math Co-Processor used with 80386.

If ET = 1, then 80387 Math Co-Processor is used.

If ET = 0, then 80287 Math Co-Processor is used.

**Note :** This bit was reserved (R) in the initial versions of 80386 and was introduced later once 80387 was developed.

### ☞ TS : Task Switched

- This bit is made "1" to indicate if a Task Switch is performed.
- 80386 µP implements "Multitasking", and thus it switches between various tasks, giving the programmer the impression that all tasks are running concurrently.
- This significantly improves the overall system performance.
- If TS = 1, it means a task switch is performed. Now the TSS of the current task has a back-link to the previous task.

**Note :** If TS bit = 1, then a type 7 interrupt called Math Co-Processor not available is caused, whenever a coprocessor instruction is encountered.

### ☞ EM : Emulate Coprocessor

- This bit is made "1" in the absence of a Math Co-Processor so that if a coprocessor instruction is encountered, then it will be executed by an on chip emulator.
- If this bit is 0, then the coprocessor instruction will be executed by 80387/80287 whichever is present in the system.

### ☞ MP : Math Co-processor Present

This bit is made "1" to indicate that a Math Co-Processor like 80387 or 80287 is present.

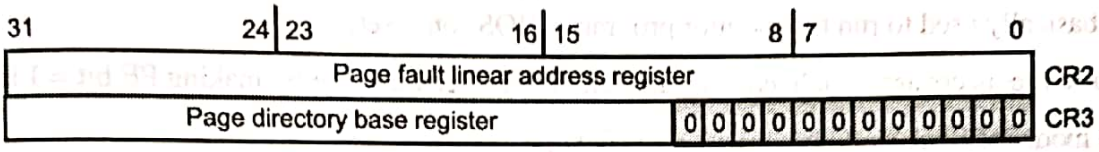
**Note :** Out of EM bit and MP bit, only one of them must be "1".

### ☞ PE : Protection Enable

- This bit is made "1" to enter protected mode.
- On reset, by default this bit is "0". It is the only bit of CR0 which is also available in Real Mode.

### 10.4.2 Control Register 1 : Not Used (Reserved by Intel)

### 10.4.3 Control Registers 2 and 3 : (Used Only For Paging)




(1F4)**Fig. 10.4.3 : CR2 & CR3**

### 10.4.4 Control Register 2 : (Page Fault Linear Address)

This is a 32-bit register giving the Linear Address that caused the last Page fault. A page Fault occurs when the desired page is not present in the Physical Memory.

### 10.4.5 Control Register 3 : (Page Directory Base Register - PDBR)

- 80386 µP implements 2-level Page translation mechanism (explained in later sections). Information about the various pages is stored in various page tables.
- Addresses of these page tables are stored in the page directory. CR3 gives the base address (starting address) of the Page Directory.
- Note that PDBR is only 20 bits. It just gives the upper 20-bits of the starting address of the page directory. That's because the page directory is of 4KB and is stored at a 4KB aligned location, so the last 12 bits of the starting address are assumed to be "0".

### Operating modes of 80386

80386 can operate in three different modes.

1. Real Mode
2. Protected Mode
3. Virtual 8086 Mode

✓**Syllabus Topic : Real mode**

## 10.5 80386 | Real Mode

**Q.** Explain the modes of operation of 80386 microprocessor (Dec. 18, 10 Marks)

**Q.** Differentiate Real Mode, Protected Mode and virtual 8086 mode of 80386 microprocessor. (May 19, 10 Marks)

- It is the **default mode selected when 80386 is reset.** In this mode, 80386 µP simply behaves as a **fast 8086 machine.**
- All registers are just like 8086. Even the **memory used is only 1 MB,** just like in 8086. Physical address calculation is also like in 8086:

**Physical Address = Base Address × 10H + Offset Address**

Eg: Segment address (Base address) = 5142H and offset address
= 0006H then the Physical address will be = 51426H

This mode is basically used to run the monitor program (BIOS) on reset.

Once the required registers are initialized, we can switch to Protected mode by making PE bit = 1 in CR0.

The Software model of Real Mode is shown on the next page.

**Segment Registers**

- There are 6, 16-bit segment registers containing the base addresses of their respective segments.
- The registers are CS, SS, DS, ES, FS and GS.

**Offset Registers**

- There are 5, 16-bit offset registers containing the offset addresses for various segments.
- The registers are IP, SP, BP SI and DI.
- The 32-bit extended form (ESI, EDI etc) of these registers is not used in Real Mode.

**Data Registers**

- There are 4, 16-bit data registers used as General Purpose Registers during programming.
- The registers are AX, BX CX and DX. They can be also used as 8, 8-bit registers AL, AH, BL, BH, CL, CH, DL and DH.
- The 32-bit extended form (EAX, EBX etc) of these registers is not used in Real Mode.

**Flags**

- Only the lower 12-bits of the Flag Register are used in the Real Mode.
- The 32-bit extended form of Flag Register i.e. EFLAGS is available only in Protected Mode.

**Control Registers**

- Only the LSB of CR0 is available in Real Mode.
- The remaining bits of CR0 and the remaining Control Registers are available only in Protected Mode.
- The LSB of CR0 is "PE" bit which should be made = "1" to begin Protected Mode.

**Debug Registers and Test Registers :** These registers are not available in Real Mode.

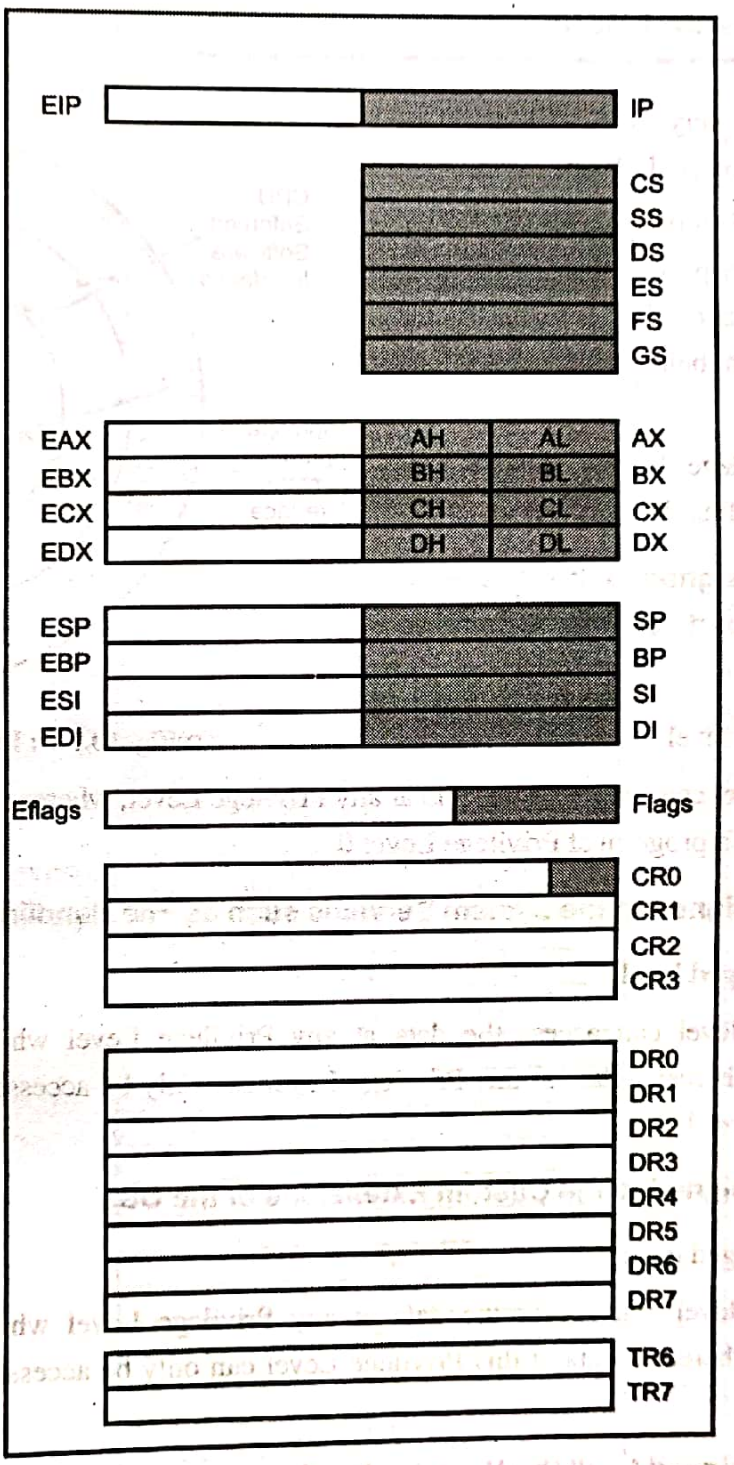
**Memory Range**

The size of memory available in real mode is 1 MB and has address range from 00000H ... FFFFFH, just like in 8086.

**I/O Range**

A total of 64K I/O addresses are available having a range from 0000H ... FFFFH, just like in 8086.

## 10.6 Software Model / Register Model of 80386 in Real Mode

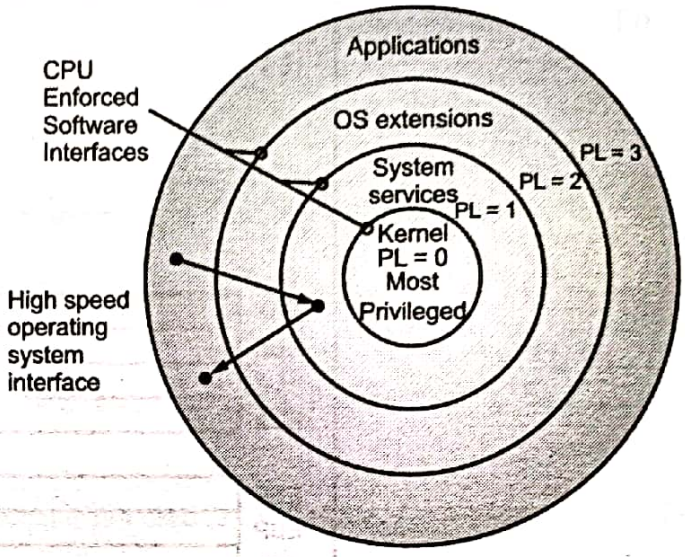



(1F5)**Fig. 10..6.1 : Software Model in Real Mode**

✓Syllabus Topic : Protected Mode

## 10.7 80386 | Protected Mode

- 80386 µP provides a very advanced mode of operations called the Protected Mode.
- In Protected Mode, 80386 µP provides dedicated hardware to prevent user programs from affecting other user programs and also safeguards the Operating System from being affected by user programs.
- There are Four Privilege Levels, assigned to programs and data to define their privileges.




(1F6)Fig. 10.7.1 : Privilege levels

**Level 0 : This level is assigned to the Operating System Kernel (Main part of the Operating System).**

- It is the most privileged level.
- Any program at this level can access all the data at any Privilege Level, whereas a data at this Privilege Level can only be accessed by a program at Privilege Level 0.

**Level 1 : This level is assigned to the System Services such as File Handling, Device Drivers.**

- It is the 2nd most privileged level.
- Any program at this level can access the data at any Privilege Level which is lower than this level (numerically higher), whereas a data at this Privilege Level can only be accessed by a program at Privilege Level 0 or Privilege Level 1.

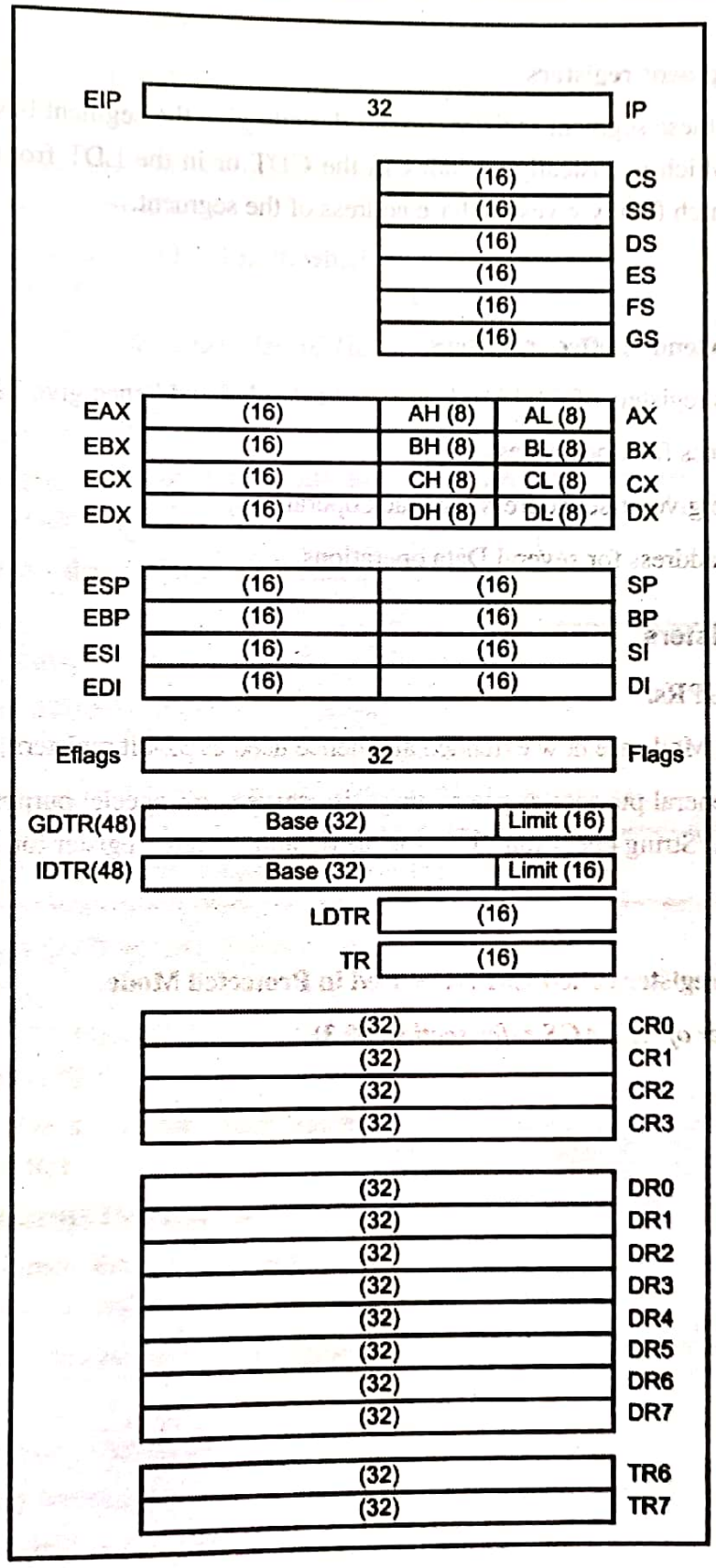
**Level 2 : This level is assigned to the Custom Extensions of the OS.**

- It is the 3rd most privileged level.
- Any program at this level can access the data at any Privilege Level which is lower than this level (numerically higher), whereas a data at this Privilege Level can only be accessed by a program at Privilege Level 0, 1 or 2.

**Level 3 : This level is assigned to all the User Application and Programs.**

- It is the least privileged level.
- Any program at this level can normally access the data at Privilege Level 3, whereas a data at this Privilege Level can be accessed by a program at any Privilege Level 0...3.

## 10.8 Protected Mode Register Model / Software Model / Programming Model




(1F7)**Fig. 10.8.1 : Software Model in Protected Mode**

The Software Model of 80386 is explained as follows :

### ☞ Segment Registers

**There are six, 16-bit segment registers.**

Unlike Real mode, now these segment registers do not directly give the segment base address. Instead, these registers give a "Selector" which is basically an index in the GDT or in the LDT from which the descriptor is loaded. It is the descriptor which finally gives the base address of the segment.

### ☞ Offset Registers

- **There are five, 32-bit extended offset registers.**
- The 16-bit offset address registers of Real Mode are now extended and hence give 32 bit offset addresses.
- EIP gives the offset address for Code access.
- ESP and EBP are used to give offset address for Stack operations.
- ESI and EDI give offset address for several Data operations.

### ☞ General Purpose Registers

- **There are four, 32-bit GPRs.**
- The 16-bit GPRs of Real Mode are now extended and hence used as 32-bit registers.
- Besides being used as general purpose registers, they also serve some special purposes like, AX – acts as an accumulator in MUL/DIV/String operations, CX – is the default "count" register for several instructions etc.

### ☞ Flag Register

**80386 has a 32-bit flag register called EFLAGS used in Protected Mode.**

***(For diagram and answer of EFLAGS refer section 10.3)***

## 10.9 GDT & GDTR (Global Descriptor Table Register)

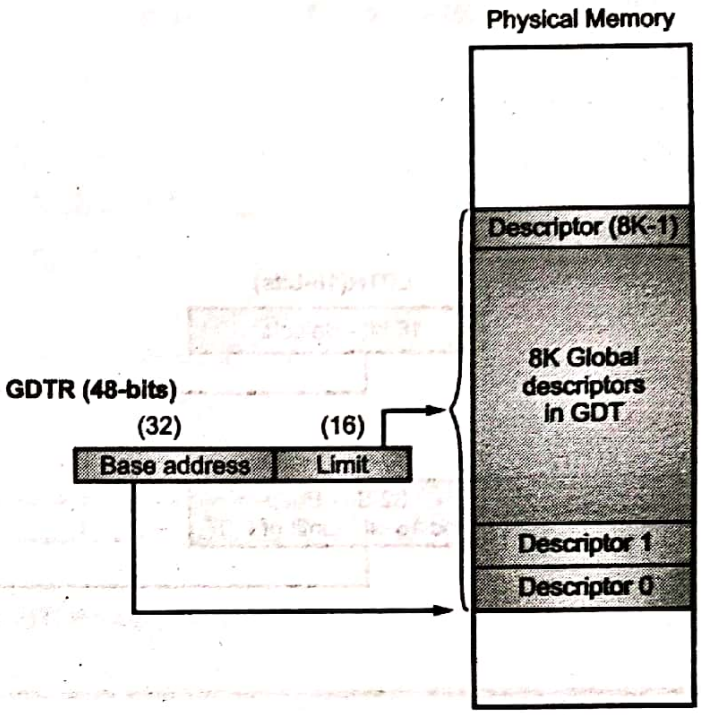
**Q.** What is GDT? Explain structure of GDT. **(May 17, 5 Marks)**

1. GDTR is a **48 bit register. It defines the GDT in the Physical Memory.**
2. The upper 4 bytes (32 bits) give the starting address of the GDT also called the base address.
3. The lower 2 bytes give the 16-bit Limit which decides the size of the GDT.
4. As the Limit field is 16-bits, the **max size of the GDT can be 64 KB.**

   The size of GDT is always Limit + 1.

   If Limit is FFFFH then size of GDT will be 65535+1=65536 i.e. 64KB.
5. **The GDT contains descriptors of Global Segments.**
6. Each descriptor is of 8 bytes.

   It gives the Starting Address, Limit and Access Rights of the segment
7. **The GDT can contain max 64KB/8 = 8K descriptors (8192 Descriptors).**

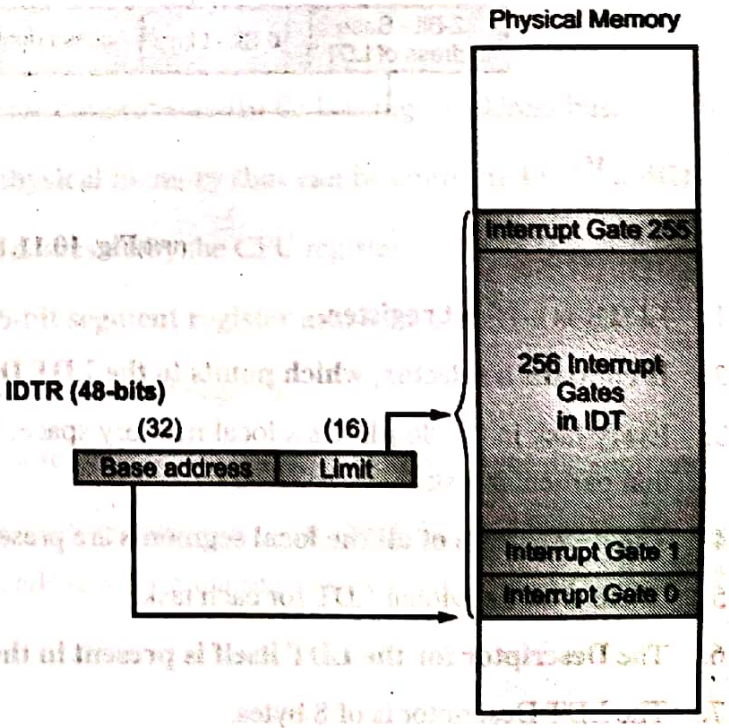



(1F8)**Fig. 10.9.1 : GDT and GDTR**

## 10.10 IDT & IDTR (Interrupt Descriptor Table Register)

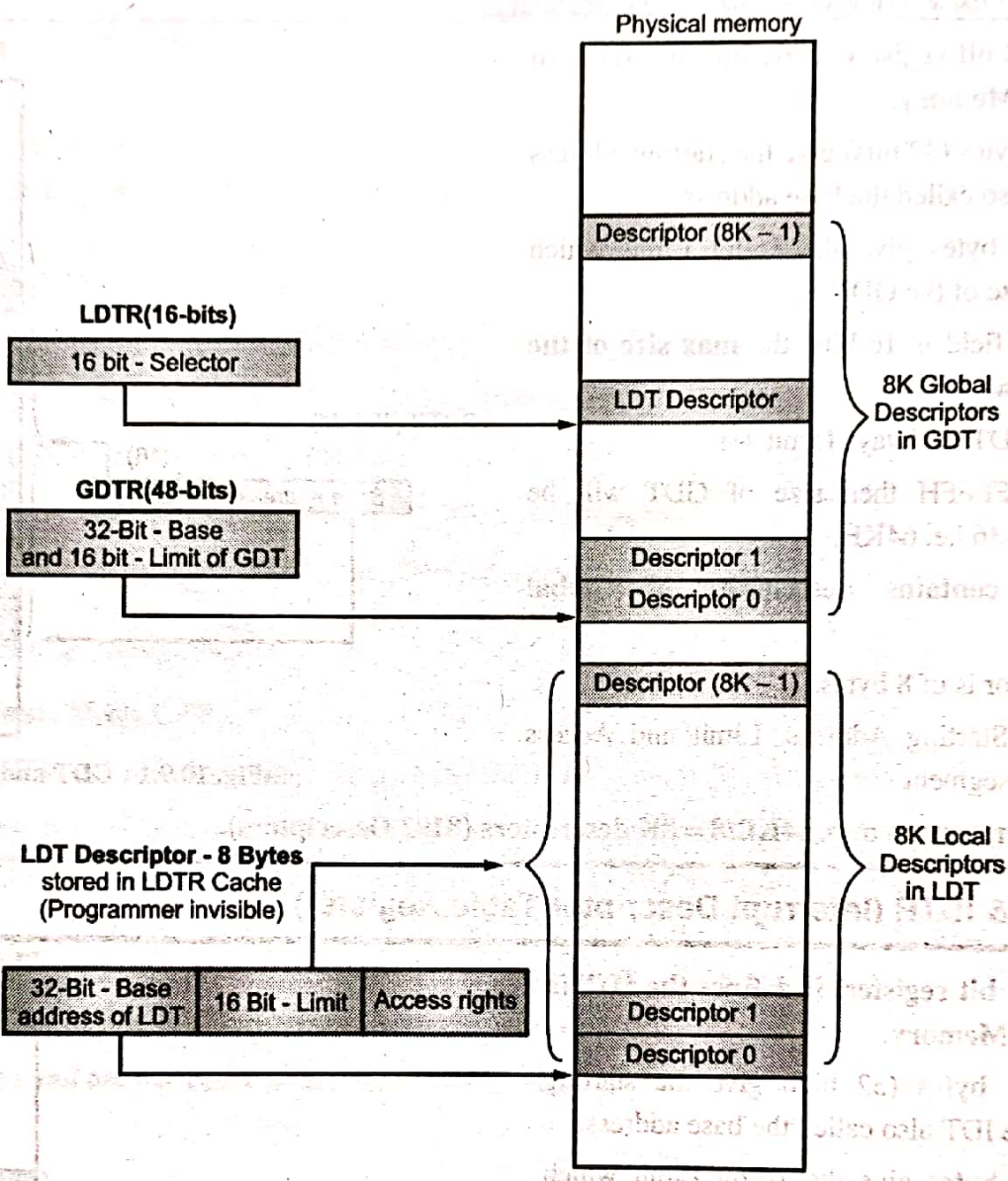
1. IDTR is a **48 bit register. It defines the IDT in the Physical Memory.**
2. The upper **4 bytes** (32 bits) give the starting address of the IDT also called the base address.
3. The lower **2 bytes** give the 16-bit Limit which decides the size of the IDT.
4. The IDT contains **Interrupt Descriptors.**
5. These Descriptors direct the µP towards the respective ISR whenever an interrupt occurs.

   Hence the Descriptors are also called **"Interrupt Gates".**
6. Each Interrupt Descriptor is of **8 bytes.**
7. 80386 µP supports **256 Interrupts.**
8. Hence **max size of the IDT is 256 x 8 = 2KB.**
9. The size of IDT is Limit + 1.
10. Hence **max value of Limit must be 07FFH.**




(1F9)**Fig. 10.10.1 : IDT & IDTR**

## 10.11 LDT & LDTR (Local Descriptor Table Register)



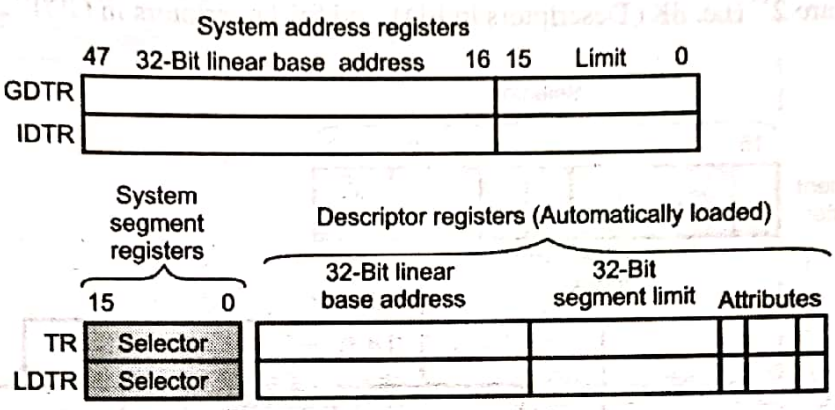

(1F10)**Fig. 10.11.1 : LDT & LDTR**

1. LDTR is a **16 bit register.**
2. **It contains a selector, which points to the LDT Descriptor in the GDT.**
3. Every task in 80386 µP has a local memory space. Here it has its local segments, which are only available to that particular task.
4. **The descriptors of all the local segments** are present in the LDT of that task.
5. There is an individual LDT for each task.
6. The **Descriptor for the LDT itself is present in the GDT.**
7. The LDT Descriptor is of 8 bytes.
8. It contains a 32 bit base address of the LDT, 16 bit Limit and Access rights information.

9. Once we load a new selector in the LDTR, the corresponding LDT Descriptor is fetched from the GDT and loaded into a LDT Descriptor Cache and thus it provides the address and limit of the LDT.
10. The structure of LDT is similar to that of GDT.

    It also has max 8K Descriptors each of size 8 Bytes and hence **max size of LDT is also 64KB.**

### ☞ Summary




(1F11)**Fig. 10.11.2 : GDTR, IDTR and LTDR**

## ➲ 10.12 80386 I Virtual Memory Management

**Q.** Explain memory management in details in 80386DX processor. (Dec. 15, 10 Marks)

**Q.** Draw format of selector and explain its field. (Dec. 17, 5 Marks)

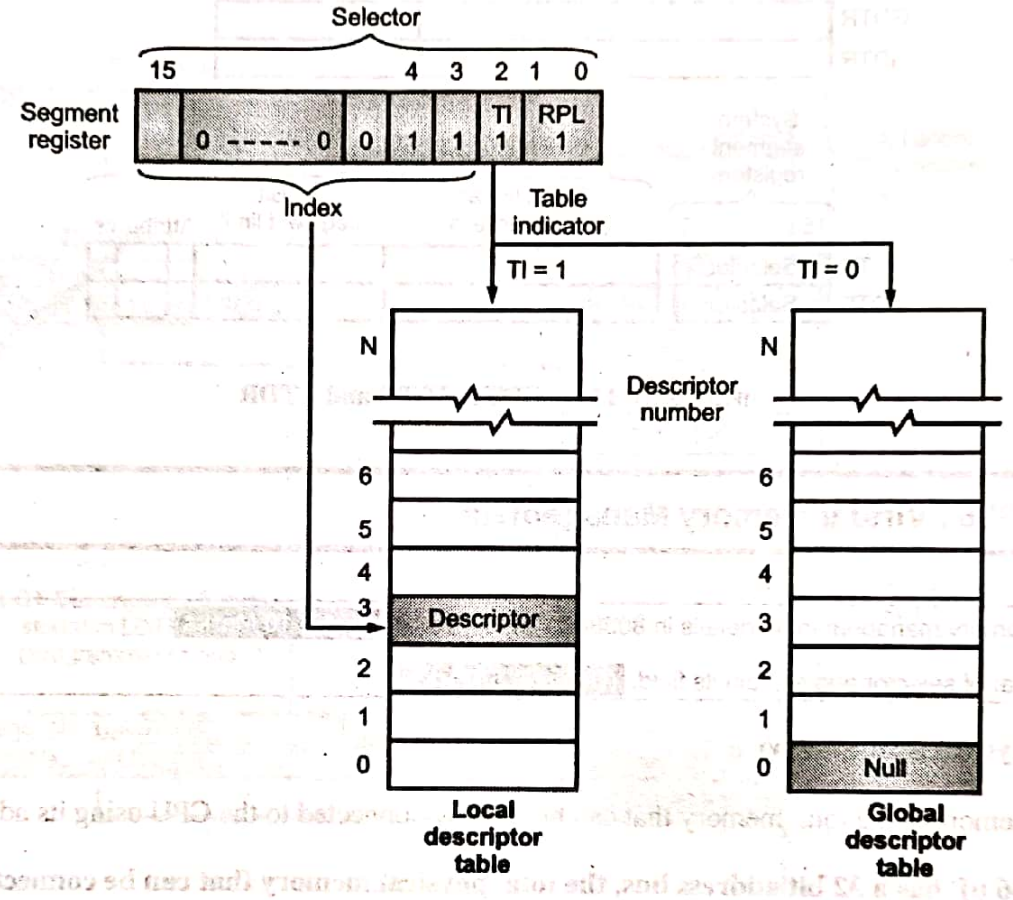
### Virtual and Physical Address Space

1. Physical Memory is the total memory that can be directly connected to the CPU using its address bus.
2. **Since 80386 µP has a 32 bit address bus, the total physical memory that can be connected is $2^{32}$ = 4GB.**
3. Virtual Memory is the total memory space that can be addressed by the CPU registers.
4. **Virtual address is basically the combination of a 16-bit segment register and a 32 bit offset register.**
5. In Real Mode, the segment register gives the starting address of the segment.
6. But in Protected Mode, the segment register just gives a selector which selects a Descriptor for the segment.
7. Though the selector is of 16-bits, only 14 bits are used as two bits give the Privilege Level used for protection as seen above. Each selector value corresponds to a different segment. Hence there can be max $2^{14}$ segments.

8. The locations within the segment are identified by their offset addresses. Since offset addresses are 32 bit, each segment can be max $2^{32}$ = 4GB. Hence the max total Virtual memory that can be accessed = Max number of segments x Max size of each segment = $2^{14} \times 2^{32} = 2^{46} = 2^6 \times 2^{40} = 64 \times 1TB = 64$ **Terra Bytes.**

9. Out of 14 bits of the selector, one bit is used as a table identifier (TI)if TI = 1 then use LDT, if TI = 0 then use GDT.

10. So basically there are $2^{13}$ (i.e. 8K) Descriptors in LDT and 8K Descriptors in GDT.




(1F12)**Fig. 10.12.1 : Working of Segment Selector**

11. **This means the total 64TB Virtual space is divided into 32TB of Global space and 32 TB of Local Space.**

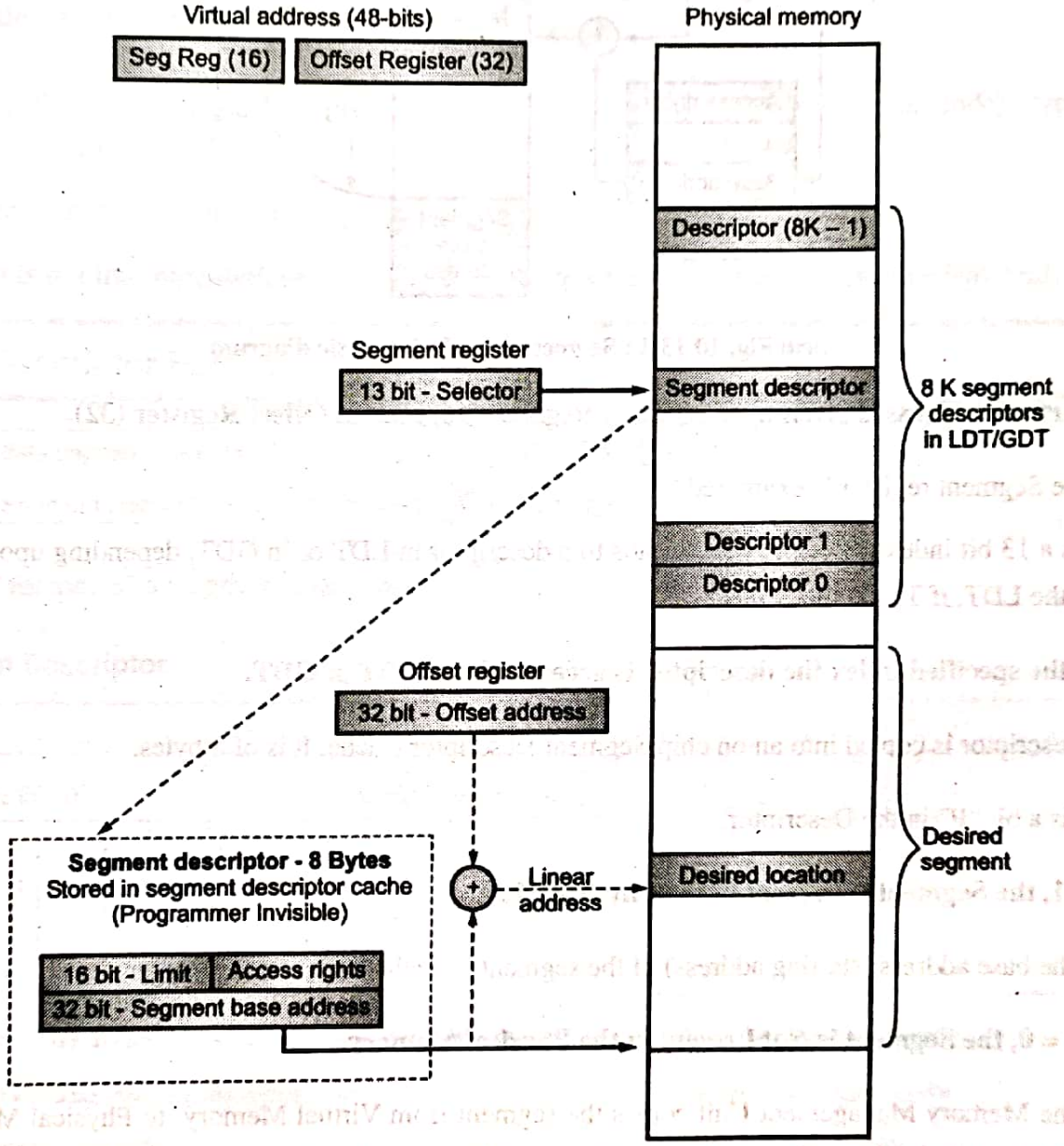
12. **The total 48 bit address having 16 bit segment address and 32 bit offset address is called Virtual address. It is converted into a 32 bit physical address using two translations: Segment translation (compulsory) and page translation (optional).**

13. **Segment translation converts 48-bit Virtual address into 32 bit Linear Address which is further converted into a 32 bit Physical Address by Page translation.**

## 10.13 80386 I Segment Translation

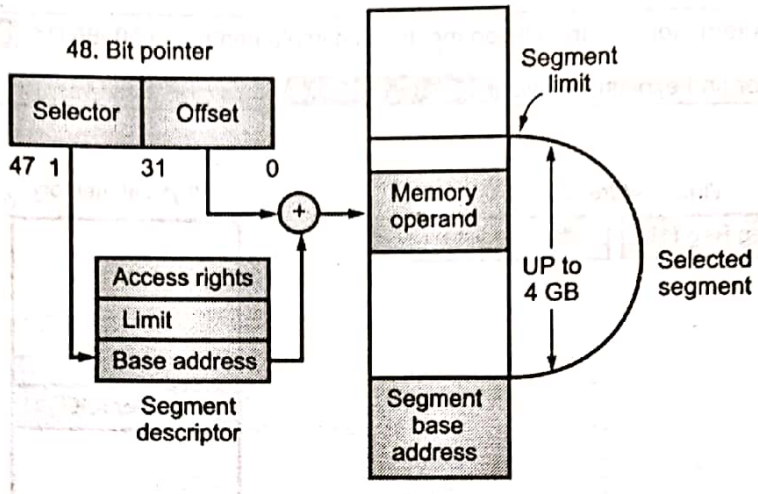
Q. Explain, with neat diagram, address translation mechanism implemented on 80386 DX. (May 17, 10 Marks)

Q. Draw format of selector and explain its field. (Dec. 17, 5 Marks)




(1F13)**Fig. 10.13.1 : Segment translation**

☞ **Basic diagram**




(1F14) **Fig. 10.13.2 : Segment translation basic diagram**

1. **The Virtual Address is given by a Segment Register (16) and an Offset Register (32).**
2. First the Segment register is examined.

   It gives a 13 bit index (selector) which leads to a descriptor in LDT or in GDT, depending upon the TI bit. If TI = 1 the LDT, if TI = 0 then GDT.
3. **Using the specified index the descriptor is accessed from LDT of GDT.**
4. This Descriptor is copied into an on chip Segment Descriptor Cache. It is of 8 bytes.
5. There is a bit "P" in the Descriptor.
6. **If P = 1, the Segment is Present in the Physical Memory.**
7. Then the base address (starting address) of the segment is available in the Descriptor.
8. **If "P" = 0, the Segment is Not Present in the Physical Memory.**
9. Now the Memory Management Unit copies the segment from Virtual Memory to Physical Memory, makes the "P" bit = 1 in the Descriptor and stores its base address into the Descriptor for further use.
10. **To this base address, the 32-bit offset address is added.**
11. **This finally gives the 32 bit Linear Address of the desired location.**
12. For subsequent access to the same segment, the Descriptor which is already cached is used.

13. The "P" bit will be 1.

14. So the base address will be directly obtained from the Descriptor and the offset address will be added to it to produce the Linear Address.

15. **Before granting access to the desired location, a protection check is performed based on the information given by the access rights in the Descriptor.**

16. Several types of checks such as limit not exceeded, type of segment (data or code), type of access (read/write/execute), privilege level etc carried out.

    If the checks are valid then access is granted, else a general protection fault occurs.

17. **If Paging is not implemented, then the 32 bit linear address is the final 32 bit Physical Address.**

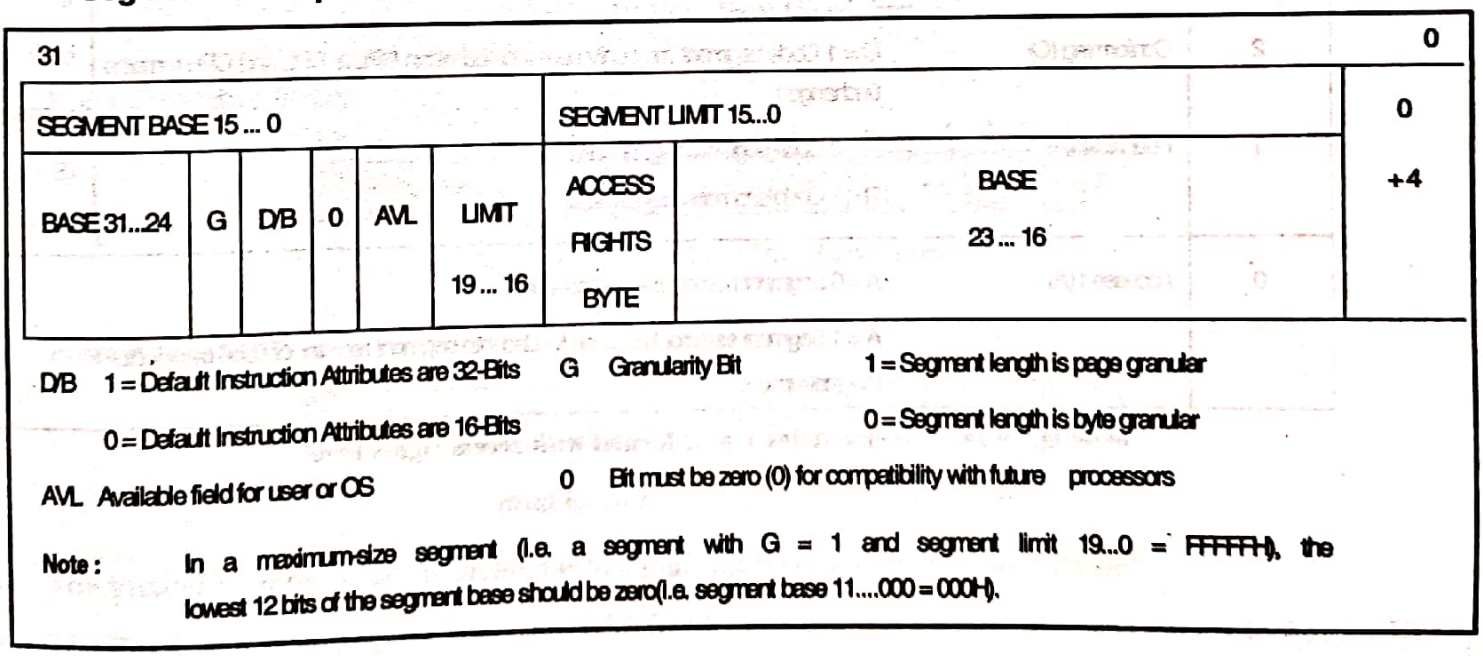
## 10.14 Descriptor Format

Q. Explain data segment descriptor with neat diagram. (Dec. 16, 10 Marks)

Q. Explain Segment Descriptor of 80386 Processor (May 18, 10 Marks)

### General format of a Segment Descriptor

### Segment Descriptor




D/B 1 = Default Instruction Attributes are 32-Bits
0 = Default Instruction Attributes are 16-Bits

G Granularity Bit
1 = Segment length is page granular
0 = Segment length is byte granular

AVL Available field for user or OS

0 Bit must be zero (0) for compatibility with future processors

Note : In a maximum-size segment (i.e. a segment with G = 1 and segment limit 19...0 = FFFFFH), the lowest 12 bits of the segment base should be zero(i.e. segment base 11....000 = 000H).

## ☞ Access Rights Byte Definition for Code and Data Descriptions

| | Bit position | Name | Function | |
|---|---|---|---|---|
| Types Field Definition | 7 | Present (P) | P = 1 Segment is mapped into physical memory.<br>P = 0 No mapping to physical memory exist , base and limit are not used. | |
| | 6-5 | Descriptor privilege Level (DPL) | Segment privilege attribute used in privilege tests. | |
| | 4 | Segment descriptor (S) | S = 1 Code or Data (includes stacks) segment descriptor<br>S = 0 System segment descriptor or gate descriptor | |
| | 3 | Executable (E) | E = 0 Descriptor type is data segment : | If data segment (S = 1, E = 0) |
| | 2 | Expansion Direction (ED) | ED = 0 Expand up segment, offsets must be ≤ limit.<br>ED = 1 Expand down segment, offsets must be > limit. | |
| | 1 | Writeable (W) | W = 0 Data segment may not be written into.<br>W = 1 Data segment may be written into. | |
| | 3 | Executable (E) | E = 1 Descriptor type is code segment : | If code segment (S = 1, E = 1) |
| | 2 | Conforming (C) | C = 1 Code segment may only be executed when CPL ≥ DPL and CPL remains unchanged. | |
| | 1 | Readable (R) | R = 0 Code segment may not be read<br>R = 1 Code segment may be read. | |
| | 0 | Accessed (A) | A = 0 Segment has not been accessed<br>A = 1 Segment selector has been loaded into segment register or used by selector test instructions. | |

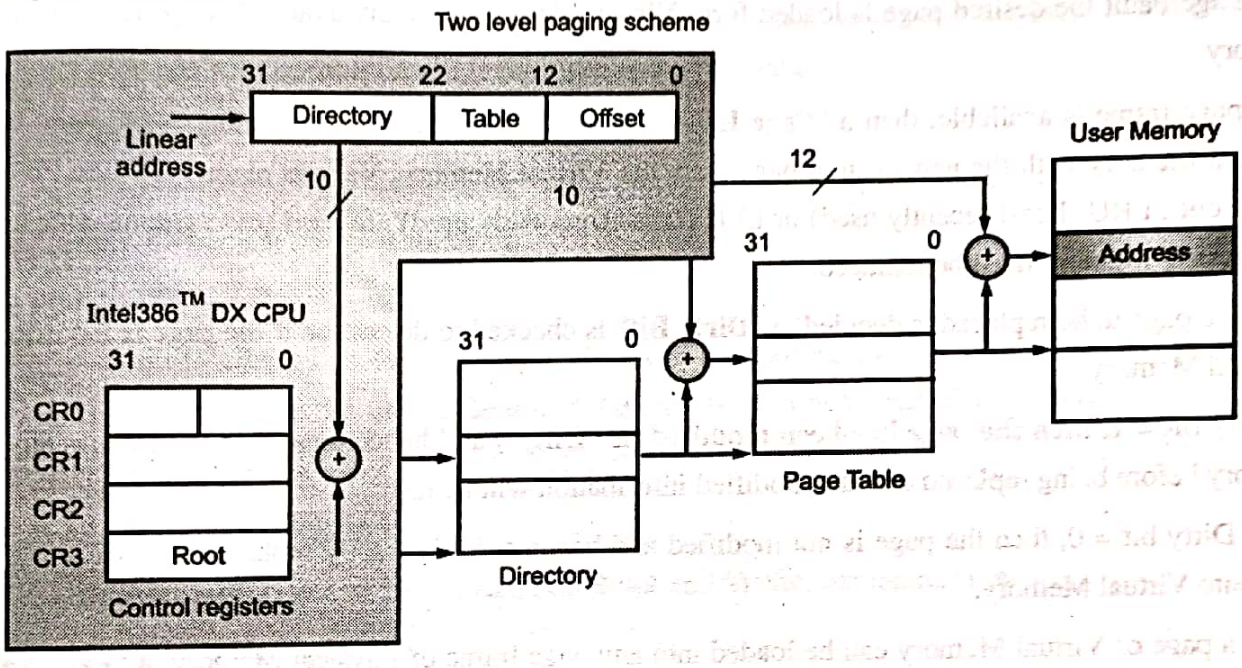
(1F16)**Fig. 10.14.1 : Segment descriptor format with access rights byte**

## 10.15 80386 | Page Translation

Q. Explain, with neat diagram, address translation mechanism implemented on 80386 DX. (May 17, 10 Marks)

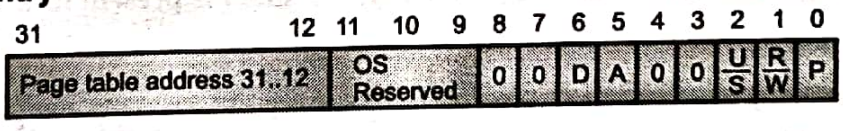
Q. Draw format of selector and explain its field. (Dec. 17, 5 Marks)

### Paging Mechanism



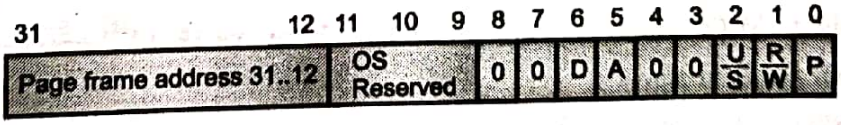

(1F17)**Fig. 10.15.1 : Page translation**

### Page Directory Entry




(1F18)**Fig. 10.15.2 : Page Directory Entry**

### Page Table Entry




(1F19)**Fig. 10.15.3 : Page Table Entry**

1. **The Virtual Memory space is divided into equal size blocks of 4KB called "Pages".**
2. **The Physical Memory space (also called Main Memory) is also divided into equal size blocks of 4 KB called Page Frames (also simply called pages).**
3. **As Physical Memory is of 4 GB and page size is 4 KB there are total 1 M pages ($2^{20}$) in the Physical Memory.**

4. A page from Virtual Memory is loaded into any available page frame of Physical Memory.
5. Whenever a page is required to be accessed, the μP first checks if the desired page is present in the Physical Memory.
   If so, it is called a "HIT" and the operation is performed on the Physical Memory.
6. **A "Page Fault" (MISS) occurs when the desired page is not present in the Physical Memory.**
7. On a Page Fault the desired page is loaded form Virtual Memory into any available page frame of Physical Memory.
8. If no page frame is available, then a **"Page Replacement"** is performed by replacing an old page from the Physical Memory with the new desired page from the Virtual Memory. Various algorithms like FIFO (First in first out), LRU (Least recently used) or LFU (Least frequently used) are used to determine which page of the Physical Memory must be replaced.
9. Once the page to be replaced is decided, a **"Dirty Bit"** is checked to determine if the page is modified in the Physical Memory.
10. **If Dirty bit = 1, then the page has been modified** (is "Dirty") and hence must be copied back into Virtual Memory before being replaced else the modified information will be lost.
11. If the Dirty bit = 0, then the page is not modified and hence can be directly replaced without being copied back into Virtual Memory.
12. Since a page of Virtual Memory can be loaded into any page frame of Physical Memory, a **"Page Table"** is required to give the mapping between Virtual Memory page number and Physical Memory page frame number.
13. Simply speaking the **Page table tells which page of Virtual Memory is present in which page of Physical Memory.**
14. But since there are too many page frames in the Physical Memory ($2^{20}$ i.e. 1M), the page table will become too large and searches will become extremely slow.
15. Hence the mechanism is further subdivided.
16. Instead of having straight 1M ($2^{20}$) entries in the page table, **there are 1K ($2^{10}$) entries in a page table and there are 1K ($2^{10}$) such page tables.**
17. $\{2^{20} = 2^{10} \times 2^{10} \ldots\ldots!\}$
18. Each page table is of 4KB and has 1K "Page Table Entries" (PTEs) each of size 4 bytes. Each PTE gives information about a Page Frame.
19. The PTE has following information:

    **20 bit page frame address :** Gives the upper 20 bits of the starting address of the page frame. Lower 12 bits are 0...0 as the page is of 4 KB and starts from a

    4 KB aligned location (refer Bharat Sir's lecture notes...)

**D : Dirty bit** indicates whether the page has been modified (1) or not (0).

**A : Accessed Bit** tells whether the page has been accessed or not (1 means accessed). This is used by replacement algorithms

**U/S : User or Supervisor and R/W : Read or Read and Write** give protection information

**P : Present bit** indicates whether the page is present in the Physical Memory.

If P = 1 then the page is present and the 20 bit address field is valid, else the page is not present in the Physical Memory and the 20 bit address field is obviously invalid.

20. Information about all the page tables is stored in the "Page Directory".
21. The page directory is of 4KB and has 1K "Page Directory Entries" (PDEs) each of size 4 bytes. Each PDE gives information about a Page Table.
22. The PDE has following information:

    **20 bit page table address :** Gives the upper 20 bits of the starting address of the corresponding page table. Lower 12 bits are 0...0 as the page table is of 4 KB and starts from a 4 KB aligned location

    **D : Dirty bit** (explained above)

    **A : Accessed Bit** (explained above)

    **U/S : User or Supervisor and R/W : Read or Read and Write** (explained above)

    **P : Present bit** (explained above)
23. The Page Directory is of 4 KB and begins from a 4 KB aligned location.
24. **The address of the page directory is given by the PGBR (page Directory Base Register) field in CR3.**
25. The 32 bit Linear Address can be divided into three parts.

    **The higher 10 bits select one PDE out of 1K PDEs in the page directory.**

    This gives the starting address in the page table.

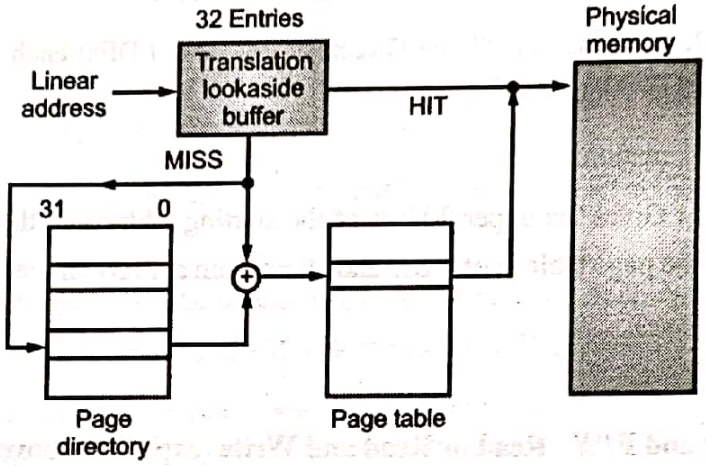
    **The next 10 bits select one PTE out of 1K PTEs in the page table.**

    This gives the starting address of the page frame.

    **Finally, the lowest 12 bits (offset) select a location within the 4KB page.**
26. This means, to access any location, µP must first access a PDE in the page directory then a PTE in the page table, then access the page. This can make the process very slow. To speed up the process a **"Translation Look-aside Buffer"** is used (called TLB).
27. **The TLB is an on chip cache which stores 32 most recently used PTEs and PDEs.** This makes subsequent access to these pages (whose information is cached in the TLB) much faster as there is no need to access the Page directory and the page table. µP can directly obtain the starting address of the page frame from the TLB and hence directly access the page. Due to principle of "Locality of reference" most systems get a Hit ratio of >98% on the TLB, thus making the operations very fast.

| U/S | R/W | Permitted Level 3 | Permitted Access Levels 0, 1 or 2 |
|---|---|---|---|
| 0 | 0 | None | Read/Write |
| 0 | 1 | None | Read/Write |
| 1 | 0 | Read-Only | Read/Write |
| 1 | 1 | Read/Write | Read/Write |

### ☞ Translation Look-aside Buffer (TLB)




(1F20)**Fig. 10.15.4 : TLB**

## ➲ 10.16 University Questions and Answers

### ➠ Dec. 2015

**Q. 2(a)** Explain memory management in details in 80386DX processor. *(Refer section 10.12)* **(10 Marks)**

### ➠ May 2016

**Q. 1(b)** Explain V86 mode of 80386DX. *(Refer section 10.3.1)* **(5 Marks)**

### ➠ Dec. 2016

**Q. 1(b)** Explain flag register of 80386DX. *(Refer section 10.3)* **(5 Marks)**

**Q. 3(b)** Explain data segment descriptor with neat diagram. *(Refer section 10.14)* **(10 Marks)**

**Q. 6(b)** Write short note on : Virtual 86 mode of 80386DX *(Refer section 10.3.1)* **(5 Marks)**

**Q. 6(d)** Write short note on ; Control registers of 80386DX *(Refer section 10.4)* **(5 Marks)**

### May 2017

**Q. 1(b)** What is GDT? Explain structure of GDT. *(Refer section 10.9)* **(5 Marks)**

**Q. 3(b)** Explain, with neat diagram, address translation mechanism implemented on 80386 DX. *(Refer sections 10.13 and 10.15)* **(10 Marks)**

**Q. 6(b)** Write short note on : Virtual 86 mode of 80386 DX *(Refer section 10.3.1)* **(5 Marks)**

**Q. 6(d)** Write short note on : Control registers of 80386 DX. *(Refer section 10.4)* **(5 Marks)**

### Dec. 2017

**Q. 1(b)** Draw format of selector and explain its field. *(Refer section 10.12)* **(5 Marks)**

**Q. 4(b)** Draw format of selector and explain its field. *(Refer sections 10.13 and 10.15)* **(5 Marks)**

**Q. 5(b)** Draw and explain EFLAG register format of 80386 DX. *(Refer section 10.3)* **(10 Marks)**

### May 2018

**Q. 1(e)** Explain Virtual Mode (VM86) 80386 processor *(Refer section 10.3.1)* **(5 Marks)**

**Q. 4(b)** Explain Segment Descriptor of 80386 Processor *(Refer section 10.14)* **(10 Marks)**

### Dec. 2018

**Q. 1(c)** Explain flag register of 80386 microprocessor. *(Refer section 10.3)* **(5 Marks)**

**Q. 4(a)** Explain the modes of operation of 80386 microprocessor *(Refer section 10.5)* **(10 Marks)**

### May 2019

**Q. 1(c)** Explain VM, RF, IOPL and NT flags of 80386 microprocessor. *(Refer section 10.3)* **(5 Marks)**

**Q. 4(a)** Differentiate Real Mode, Protected Mode and virtual 8086 mode of 80386 microprocessor. *(Refer sections 10.5 and 10.6)* **(10 Marks)**

---

*Chapter ends...*

❑❑❑


# Pentium Processor

## 11.1 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:

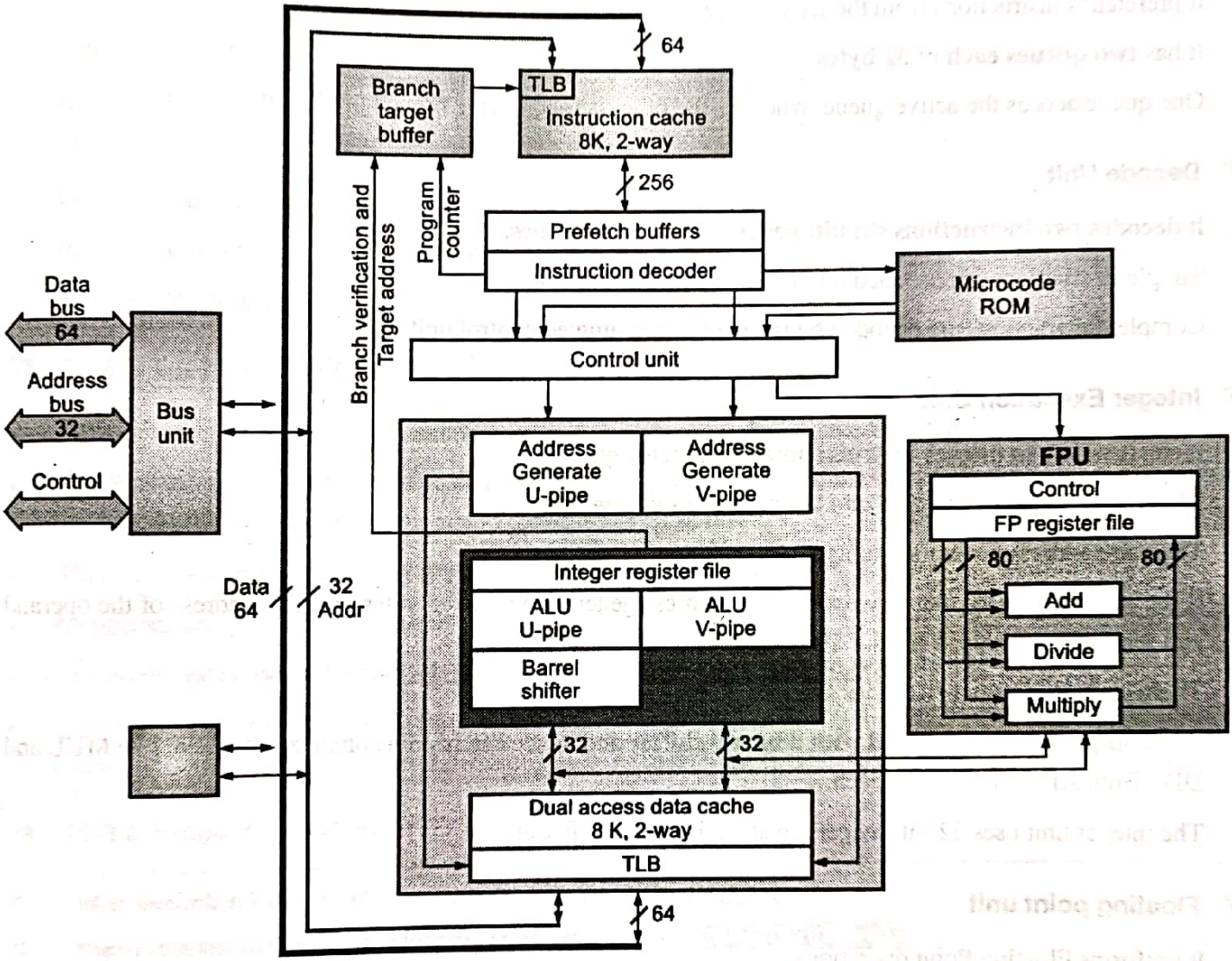
    **PF – Prefetch; D1 – Decode; D2 – Address Translation;**

    **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).

✓Syllabus Topic : Pentium Architecture

## 11.2 Pentium I Super Scalar Architecture

**Q.** Draw and explain block diagram of Pentium processor. **(May 17, 10 Marks).**




(1G1)**Fig. 11.2.1 : Pentium 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.**

✓ Syllabus Topic : Integer Pipeline Stages

## 11.3 Pentium I Integer Pipeline Stages

Q. Explain in brief, pipeline stages on Pentium processor. (May 16, 5 Marks)

Q. Explain integer pipeline of Pentium processor? (May 17, 5 Marks)

Pentium performs integer instructions in a **five-stage pipeline.**

PF - Prefetch

Dl - Instruction Decode

D2 - Address Generate

EX - Execute - ALU and Cache Access

WB - Write-Back

### 11.3.1 Stage 1 : Prefetch

- Here instructions are fetched from the **L1 Cache** and stores them into the Prefetch queue.
- The Prefetch queue is of **32 bytes** as it needs atleast two full instructions to be present inside for feeding the two pipelines, and maximum size of an instruction is **15 bytes.**
- There are two Prefetch queues but **only one of them is active** at a time.
- It supplies the instructions to the two pipes.
- The other one is used when branch prediction logic predicts a branch to be **"taken".**
- Since the bus from L1 cache to the prefetcher is of **256 bits** (32 Bytes), the entire queue can be fetched in **1 Cycle.** (T State)

### 11.3.2 Stage 2 : Decode

Q. Write instruction issue algorithm used in Pentium. (Dec. 17, 5 Marks)

Q. Explain the instructions issue algorithm of Pentium processor (Dec. 18, 10 Marks)

Q. Explain an instruction issue algorithm of Pentium processor. (May 19, 5 Marks)

- The decode stage decodes the instruction opcode.
- It also checks for instruction pairing and performs branch prediction.
- Certain rules are provided for instruction pairing. Not all instructions are pairable.
- If the two instructions can be paired, the first one is given to the u pipe and the second one to the v pipe. If not, then the first one is given to the u pipe and the second one is held back and then paired with the forthcoming instruction.

### ☞ Instruction Pairing Algorithm (Issue Algorithm)

**Consider two consecutive instructions I1 and I2, decoded by the μP...**

**If all the following are true :**

**I1 is a Simple instruction**

**I2 is a Simple instruction**

**I1 is not a Jump instruction**

**Destination of I1 not the same as Source of I2**

**Destination of I1 not the same as Destination of I2**

**Then**

**Issue I1 to U-Pipe and I2 to V-Pipe**

**Else**

**Issue I1 to U-Pipe**

### ☞ Branch Prediction

- The Pentium processor includes branch prediction logic.
- This prevents flushing of pipelines during a branch operation.
- When a branch operation is correctly predicted, no performance penalty is incurred.

## 11.3.3 Stage 3 : Decode 2 or Address Generation Stage

- **It performs address generation where it generates the physical address of the**
- **required memory operand using segment translation and page translation.**
- **Even protection checks are performed at this stage.**
- The address calculation is fast due to segment descriptor caches and TLB.
- In most cases the address translation is performed in 1 cycle itself.

## 11.3.4 Stage 4 : Execution Stage

- The Execution stage mainly uses the ALU.
- **The U pipeline's ALU has a barrel shifter,** while the **V pipeline's does not.**
- Instructions involving shifting like MUL, DIV etc can **only be done by U pipeline.**
- Operands are either provided by registers or by data cache (assuming a hit).
- Both, u and v pipes can access the **data cache simultaneously.**
- During execution, if the u pipe instruction stalls, the v pipe one has to also stall.
- But if the v pipe instruction stalls, the u pipe one can continue.

### 11.3.5 Stage 5 : Write-Back stage

- As the name suggests, the **result is written back** into the appropriate registers.
- The **flags are updated** accordingly.

✓**Syllabus Topic : Floating Point Pipeline Stages**

## 11.4 Floating Point Instruction Pipeline Stages

**Q.** Explain in brief, pipeline stages on Pentium processor. **(May 16, 5 Marks)**

**Q.** Draw and explain floating point pipeline for Pentium processor. **(May 18, 5 Marks)**

| Stage | Description |
|---|---|
| Prefetch | Identical to the integer prefetch stage. |
| Instruction De-code 1 | Identical to the integer D1 stage. |
| Instruction De-code 2 | Identical to the integer D2 stage. |
| Execution Stage (Ex) | Register read, memory read, or memory write performed as required by the instruction (to access an operand). |
| FP Execution 1 stage | Information from register or memory is written into a FP register. Data is converted to floating-point format before being loaded into the floating-point unit. |
| FP Execution 2 stage | Floating-point operation performed within floating-point unit. |
| Write FP Result | Floating-point results are rounded and the result is written to the target floating-point register. |
| Error Reporting | If an error is detected, an error reporting stage is entered where the error is reported and the FPU status word is up-dated. |

- Most floating point instructions are issued singly to the U pipeline and cannot be paired with integer instructions. It consists of eight pipeline stages.
- The first four stages are shared with integer pipeline and the last four reside within the floating point unit itself.

✓Syllabus Topic : Branch Prediction Logic

## 11.5 Pentium I Branch Prediction Logic

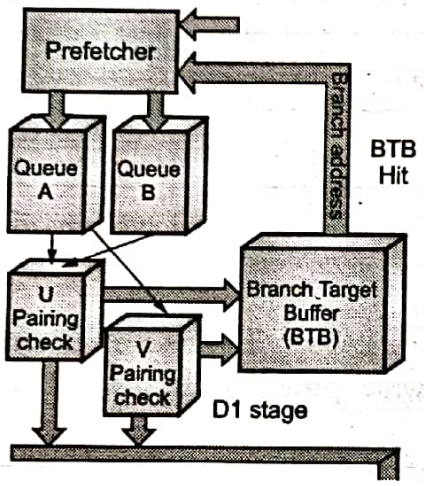
Q. Explain branch prediction logic used in Pentium. **(Dec. 15, 10 Marks)**

Q. Explain, in brief, branch prediction mechanism is on Pentium processor. **(Dec. 16, 10 Marks)**

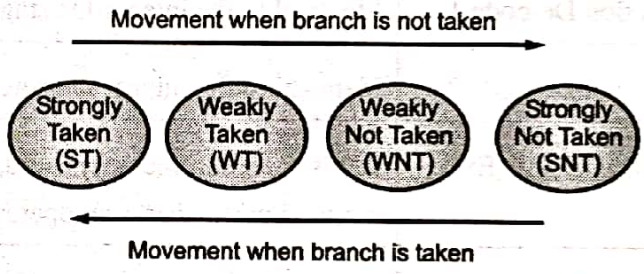
Q. Write short note on : Branch prediction logic. **(May 17, 5 Marks)**

Q. How flushing problem is minimized in Pentium? Explain. **(Dec. 17, 10 Marks)**

Q. Explain the branch prediction logic used in Pentium processor **(Dec. 18, 10 Marks)**




(1G2)**Fig. 11.5.1 : Branch Prediction Logic**




(1G3)**Fig. 11.5.2 : History bits**

- The Pentium processor includes branch prediction logic, allowing it to minimize pipeline flushing.
- When a branch operation is correctly predicted, no performance penalty is incurred.
- However, when branch prediction is not correct, a three cycle penalty is incurred if the branch is executed in the U pipeline and upto four (3+1 extra may be needed) cycle penalty if the branch is in the V pipeline.
- The prediction mechanism is implemented using a four-way, set-associative Cache with 256 entries.
- This is referred to as the **Branch Target Buffer**, or BTB.
- The directory entry for each line contains the following information.
  1. **A valid bit** that indicates whether or not the entry is in use.
  2. **Two History bits** that-track how often the branch has been taken each time that it entered the pipeline before.
  3. The **Memory Address** of the branch instruction for identification.
- The **Branch Target Buffer**, or BTB, is a look-aside cache that sits off to the side of the Dl stages of the two pipelines and monitors for branch instructions.
- During D1 stage, when an instruction is decoded and identified as a branch instruction, the address of the instruction is searched in the BTB for a previous history.

- If no history exists, then prediction is made that the branch will not be taken.
- If there is a history (BTB hit), then prediction is made as follows:
  - If the History bits are 00 or 01 (Strongly Not taken or weakly not taken), then the prediction is that the branch will not be taken.
  - If the History bits are 10 or 11 (Strongly taken or weakly taken), then the prediction is that the branch will be taken.
- **If the branch is predicted to be taken,** then the active queue is no longer used. Instead, the prefetcher starts fetching instructions from the branch address and stores them into the second queue which now becomes the active queue. This queue now starts feeding instructions into the two pipes.
- **If branch is predicted to be not taken,** then nothing changes, and the active queue remains active and instructions are fetched from the sequentially next locations.
- When the instruction reaches the execution stage, the branch will either be taken or not taken. If taken, the next instruction to be executed should be the one fetched from the branch target address,
- If the branch is not taken the next instruction executed should be the one fetched from the next sequential memory address after the branch instruction.
- When the branch is taken for the first time, the execution unit provides feedback to the prediction logic. The branch target address is sent back and recorded in the BTB.
- A directory entry is made containing the source memory address that the branch instruction was fetched from and history bits are set to .indicate that the branch has been strongly taken.
- The history bits can indicate one of four possible states.

| History Bits | Resulting Description | Prediction made | If actually taken | If actually not taken |
|---|---|---|---|---|
| 00 | Strongly Not Taken | Branch Will Not Be Taken | Upgrades to Weakly Not Taken | Remains Strongly Not Taken |
| 01 | Weakly Not Taken | Branch Will Not Be Taken | Upgrades to Weakly Taken | Downgrades to Strongly Not Taken |
| 10 | Weakly Taken | Branch Will Be Taken | Upgrades to Strongly Taken | Downgrades to Weakly Not Taken |
| 11 | Strongly Taken | Branch Will Be Taken | Remains Strongly Taken | Downgrades to Weakly Taken |

During execution stage the μP realizes whether the prediction was true or false. The following actions thus take place....

1. If the branch was correctly predicted taken, the entry's history bits are upgraded and no further action is necessary. The correct instructions are already in the pipelines behind the branch instruction.
2. If the branch was incorrectly predicted taken, the entry is downgraded. The instructions in the pipelines behind the branch are incorrect and must be flushed. The branch prediction logic instructs the prefetcher to switch back to the other queue and resume sequential code fetches.
3. If the branch was correctly predicted not taken and there is a corresponding entry in the BTB, downgrade the entry's history bits. If the branch was correctly predicted not taken (because a BTB miss occurred in the D1 stage) and there isn't a corresponding entry in the BTB, do not make an entry in the BTB.
4. If the branch was incorrectly predicted not taken and there is a corresponding entry in the BTB, upgrade the entry's history bits. If the branch was incorrectly predicted not taken and there isn't a corresponding entry in the BTB, make an entry and mark it strongly taken.

✓**Syllabus Topic : Cache Organisation**

## 11.6 Pentium I Cache Memory Organisation

### 11.6.1 Features of Pentium Cache

1. The Pentium processor has a separated code and data cache each of 8k bytes.
2. The cache line size is 32-bytes.
3. Since the Pentium processor has data bus of 8 bytes (64 - bits), it requires a burst of four consecutive transfers to fill the cache line of 32 bytes.
4. Each cache is organized as two-way set-associative.
5. The data cache can be configured as a write-through or a write-back cache on a line-byline basis and it follows the MESI protocol.
6. The code cache does not require a write policy, as it is a read-only cache.
7. Each cache has a dedicated translation look aside buffer (TLB) to translate linear addresses to physical addresses.
8. The data cache tags are triple ported to support two data transfers and a snoop cycle in the" same clock.
9. The code cache tags are also triple ported to support snooping and split line access simultaneously.
10. Individual pages in the main memory can be configured as cacheable or non-cacheable by software or hardware.
11. The cache can be enabled or disabled by software or hardware.

✓Syllabus Topic : MESI Model

## 11.6.2 The MESI Cache Consistency Model

- The MESI (modified-Exclusive-Shared-Invalid) protocol provides a method to maintain cache coherency.
- The MESI protocol is only for the data cache and the SI protocol for the code cache.
- Each line in the data cache can be in one of the four MESI states as indicated by two bits stored along with the tag address.

### Modified

- It indicates that this line in cache has been updated or modified due to a write hit in the cache.
- In this case, when the cache subsystem snoops the system bus and finds a snoop hit, it write the modified line back to memory (update the memory).

### Exclusive

It is the intermediate state between Shared and Modified.

### Shared

It indicates that this line may be resent in several caches and an exact duplicate of the information exists in each source (caches and main memory).

### Invalid

- It is the initial state after reset and indicates that the line in not present in the cache.
- During reset, the MESI state bits for the processor's internal LI caches and the L2 cache are forced to the invalid (I) state. Hence, all initial accesses to LI and L2 cache are cache misses.

## 11.6.3 Code Cache

**Q.** Explain, in brief, cache organization of pentium processor. (May 16, May 19, 10 Marks)

**Q.** Explain, with neat diagram, cache memory organization is supported by Pentium processor. (Dec. 16, May 17, 10 Marks)

- The code cache is 8 KB read only cache and is organized as a 2-way set associative cache. The two ways are called way zero and way one. Each cache line is 32 byte.
- Total size of cache = 8 KB.
- Size of each way = 4 KB.
- Cache line size = 32 bytes.
- Total number of lines in one way = 128
- There is a separate 128 entry directory associated with each cache way. The 4GB memory space of the

processor is mapped into the 4KB cache way, i.e. the 4 GB memory space is assumed to be divided into pages each of size 4 KB.

- Hence total number of pages = 1M or $2^{20}$
- Thus, each cache directory entry stores this 20 bit tag (page no) address A [31-12].
- The entry also consists of one state bit (to indicate S/I) and a parity bit P.
- The parity bit is used to detect errors when reading each entry.
- The cache directories are triple ported to allow three simultaneous directory accesses.
- Two of the ports support the split line access capability, while the third port is used for snooping.
- Each cache line holds four quad words and a parity bit for each quad word
- When the prefetcher issues an instruction request, the code cache is checked to see if a copy is available.
- Assuming a cache miss, a cache line-fill request is made to the bus unit, i.e. a cache line is brought in from L2 cache/memory.

### ☞ Main Memory Address (32-bits) Interpretation

| Page Number (Tag) | Line Number (index) | Location within the line (byte) |
|---|---|---|
| **20 bits** | **7 bits** | **5 bits** |

### ☞ Code Cache Directory Entry

| Parity | Tag (page Number – Address[31-12]) | S/I |
|---|---|---|

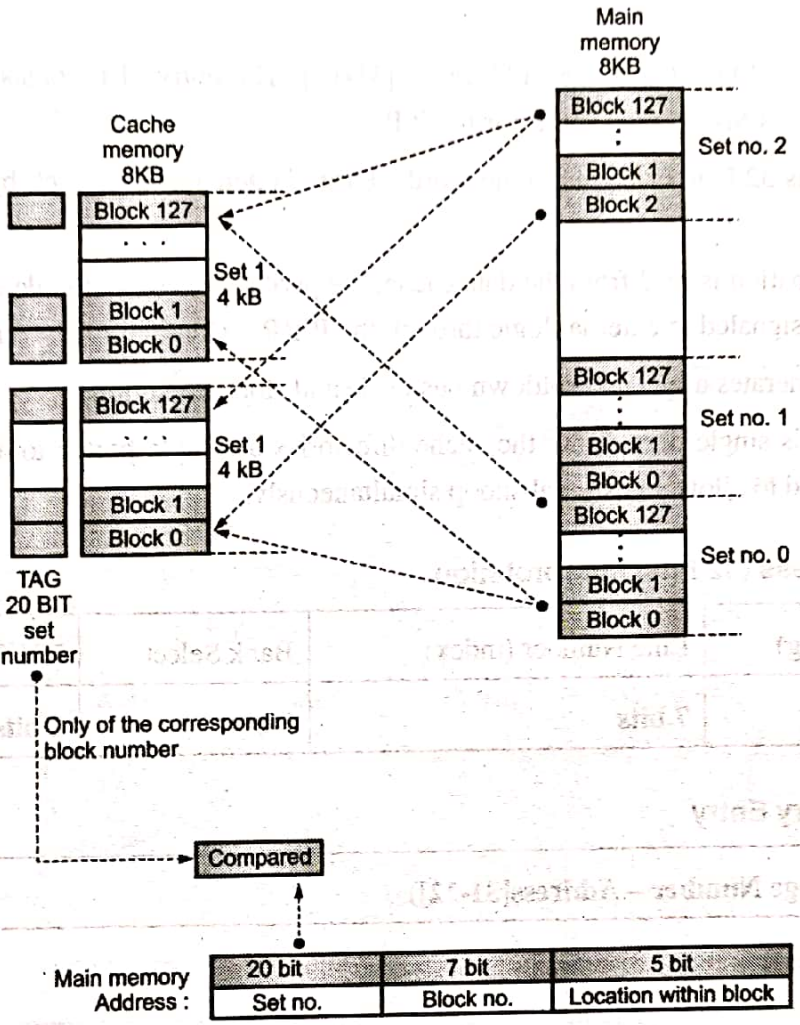
### ☞ Code Cache line (32-Bytes)

| P3 | QW3 | P2 | QW2 | P1 | QW1 | P0 | QW0 |
|---|---|---|---|---|---|---|---|

### ☞ Split-Line Access

- In a CISC processor like Pentium, instructions are of variable length.
- In the Pentium processor the smallest instruction is one byte while the maximum legal length is 15 bytes.
- A code cache miss always results in a 32-byte cache line fill, if it's a cacheable address.
- Multi-byte instructions may span two sequential lines stored in the code cache.
- When the prefetcher determines that the instruction is straddled across two lines, it would have to perform two sequential cache accesses, which would hamper performance.
- **For this reason the Pentium processor incorporates a split line access which allows the upper half of one line and the lower half of the next line to be accessed in one cycle.**
- When a split line access is made the bytes must be rotated so that they are in proper order. In order for the split access to work efficiently, instruction boundaries within the cache line need to be defined.

- When an instruction is decoded the first time the length of the instruction is fed back to the cache.
- Each code cache entry marks instruction boundaries within the line so that if necessary split line accesses can be performed.




(1G4)**Fig. 11.6.1 : Code and Data Cache Organization**

## 11.6.4 The Data Cache

**Q.** Explain, in brief, cache organization of pentium processor. (May 16, May 19, 10 Marks)

**Q.** Explain, with neat diagram, cache memory organization is supported by Pentium processor. (Dec. 16, May 17, 10 Marks)

**Q.** Discuss data cache organization of Pentium. (Dec. 17, 10 Marks)

**Q.** Explain Data Cache architecture for Pentium processor. (May 18, 10 Marks)

The data cache is 8KB and is organized as a 2-way set associative cache. The two ways are called way 0 and way 1. Each cache line is 32 bytes.

Total size of cache = 8KB

Size of each way = 4 KB

Cache line size = 32 bytes.

Number of lines = 128.

- Each 4 KB cache way is divided into 128 lines. There are thus correspondingly 128 entries in each tag directory.
- Each directory stores a 20-bit tag (page) address A [31-12]. The entry also consists of two state bits (to indicate one of four states M-E-S or I) and a parity bit P.
- Each data cache line is 32 byte or eight double words. Parity is generated for each byte within a data cache line.
- When a byte of information is read from the data cache, the parity is checked. On detecting a parity error, an internal parity error is signaled to external logic through the IERR # (Internal Error) output.
- One processor also generates a special shutdown bus cycle and stops execution.
- The data cache itself is single ported, but the cache directories are triple ported to allow access form both pipelines (U and V) and to allow an external snoop simultaneously.

### ☞ Main Memory Address (32-bits) Interpretation

| Page Number (Tag) | Line Number (index) | Bank Select | XX (Don't care) |
|---|---|---|---|
| **20 bits** | **7 bits** | **3 bits** | **2 bits** |

### ☞ Data Cache Directory Entry

| Parity | Tag (page Number – Address[31-12]) | MESI |
|---|---|---|
| | | 2bits<br>00-Invalid<br>01-Exclusive<br>10-Modified<br>11-Shared |

### ☞ DATA Cache line (32-Bytes)

| P31 | Byte31 | ... | ... | P1 | Byte 1 | P0 | Byte 0 |
|---|---|---|---|---|---|---|---|

## ☞ 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 |

## 11.7 University Questions and Answers

### Dec. 2015

**Q. 4(a)** Explain branch prediction logic used in Pentium. *(Refer section 11.5)* **(10 Marks)**

### May 2016

**Q. 1(c)** Explain in brief, pipeline stages on Pentium processor. *(Refer sections 11.3 and 11.4)* **(5 Marks)**

**Q. 3(b)** Explain, in brief, cache organization of pentium processor. *(Refer section 11.6.3 and 11.6.4)* **(8 Marks)**

**Q. 5(a)** Draw and explain architecture of Pentium processor. *(Refer section 11.2)* **(10 Marks)**

**Q. 5(b)** Draw timing diagram of read operation on 8086 based system. *(Refer section 11.4.2)* **(10 Marks)**

### Dec. 2016

**Q. 4(a)** Explain, in brief, branch prediction mechanism is on Pentium processor. *(Refer section 11.5)* **(10 Marks)**

**Q. 4(b)** Explain, with neat diagram, cache memory organization is supported by Pentium processor. *(Refer sections 11.6.3 and 11.6.4)* **(10 Marks)**

### May 2017

**Q. 1(c)** Explain integer pipeline of Pentium processor? *(Refer section 11.3)* **(5 Marks)**

**Q. 4(a)** Explain, with neat diagram, cache memory organization is supported by Pentium processor. *(Refer section 11.6.3 and 11.6.4)* **(10 Marks)**

**Q. 4(b)** Draw and explain block diagram of Pentium processor. *(Refer section 11.2)* **(10 Marks)**

**Q. 6(c)** Write short note on : Branch prediction logic *(Refer section 11.5)* **(5 Marks)**

### Dec. 2017

**Q. 1(a)** Write instruction issue algorithm used in Pentium. *(Refer section 11.3.2)* **(5 Marks)**

**Q. 3(a)** How flushing problem is minimized in Pentium? Explain. *(Refer section 11.5)* **(10 Marks)**

**Q. 4(a)** Discuss data cache organization of Pentium. *(Refer section 11.6.4)* **(10 Marks)**

### May 2018

**Q. 1(b)** Draw and explain floating point pipeline for Pentium processor. *(Refer section 11.4)* **(5 Marks)**

**Q. 5(b)** Explain Data Cache architecture for Pentium processor *(Refer section 11.6.4)* **(10 Marks)**

### Dec. 2018

**Q. 3(a)** Explain the branch prediction logic used in Pentium processor *(Refer section 11.5)* **(10 Marks)**

**Q. 4(b)(II)** Explain the instructions issue algorithm of Pentium processor *(Refer section 11.3.2)* **(10 Marks)**

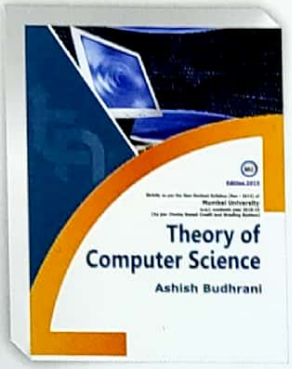
### May 2019

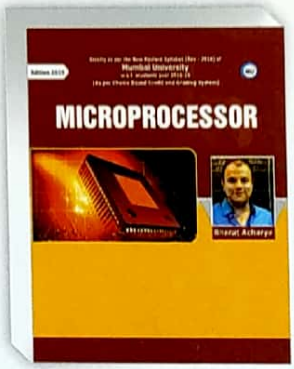
**Q. 1(d)** Explain an instruction issue algorithm of Pentium processor. *(Refer section 11.3.2)* **(5 Marks)**

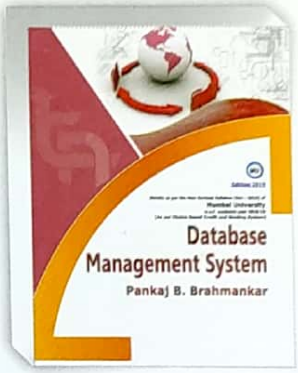
**Q. 2(b)** Explain cache organization of Pentium processor. *(Refer sections 11.6.3 and 11.6.4)* **(10 Marks)**

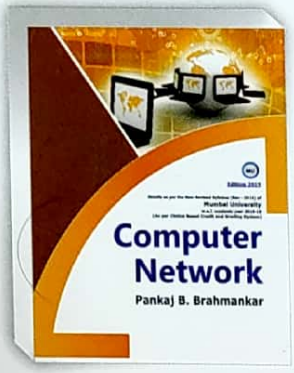
*Chapter ends...*

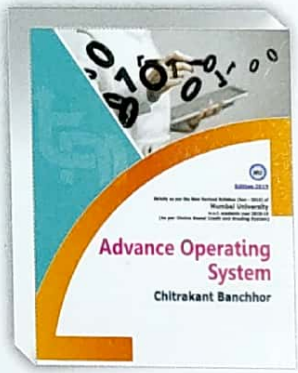
## Semester 5 - Computer Engineering




TECH-NEO PUBLICATIONS
Where Authors Inspire Innovation
A Sachin Shah Venture

SCAN TO VISIT
www.techneobooks.com

For Orders Contact :

**Krishna Book Collections**
Ground Floor, Krishna Niwas Building, Behind BEST Niwas Building, Near Napoo Hall, Chandavarkar Road, Matunga East, Mumbai 400019. **E-mail :** dharmeshsota05@gmail.com
**Mobile No.:** Dharmesh Sota - 9820741455 Tulsidas Sota - 9833133921 / 9833082745 / 9833082761

**Student's Agencies (I) Pvt. Ltd.**
109, Konark Shram, Behind Everest Building, 156, Tardeo Road, Mumbai - 400034. **Tel. No. :** 022-40496161, 9167290777 **E-mail :** students@gmail.com

**Mital Traders**
B-228, Antophill Warehousing Co. Vidyalankar College Road, Nr. Barkat Ali Naka, Wadala (E), Mumbai - 400037.
**Tel. No. :** 022 24183322, 9821569201/8879747908.

**Raj Traders**
Shop No. 05, MMC 17-1/1, Opp. Mod Shoes, Sai Baba Mandir Road, Near Rly. Staion, Chembur (E), Mumbai, Maharashtra - 400071.

ISBN : 978-93-89366-23-5

Price ₹ 280/-
(A311)

info@techneobooks.com
www.techneobooks.com

Tech-Neo Books
Tech-Neo Publications
Tech-Neo Publications
Tech-Neo Publications