

# **Microprocessor (8085) And its Applications**

## About the Author

**A Nagoor Kani** is a multifaceted personality with efficient technical expertise and management skills. He obtained his BE degree in Electrical and Electronics Engineering from Thiagarajar College of Engineering, Madurai, and MS (Electronics and Control) through Distance Learning program of BITS, Pilani. He is a life member of ISTE and IETE.

He started his career as a self-employed industrialist (1986-1989) and then changed over to teaching in 1989. He has worked as Lecturer in Dr MGR Engineering College (1989-1990) and as Asst. Professor in Satyabhama Engineering College (1990-1997). In 1993, he started a teaching centre for BE students named Institute of Electrical Engineering, which was renamed RBA Tutorials in 2005.

A Nagoor Kani launched his own organization in 1997. The ventures currently run by him are RBA engineering (involved in manufacturing of lab equipments, microprocessor trainer kits and undertake Electrical contracts and provide electrical consultancy), RBA Innovations (involved in developing projects for engineering students and industries), RBA Tutorials (conducting tutorial classes for engineering students and coaching for GATE, IES, IAS) and RBA Publications (publishing of engineering books), RBA Software (involved in web-design and maintenance). His optimistic and innovative ideas have made the RBA Group a very successful venture.

A Nagoor Kani is a well-known name in major engineering colleges in India. He is an eminent writer and till now he has authored several engineering books (published by Tata McGraw Hill Education and RBA Publications) which are very popular among engineering students. He has written books in the areas of Control Systems, Signals and Systems, Microcontrollers, Digital Signal Processing, Electric Circuits, Electrical Machines and Power Systems.

# **Microprocessor (8085) And its Applications**

**THIRD EDITION**

*A Nagoor Kani*

*Founder  
RBA Group  
Chennai*

**Tata McGraw Hill**

Published by the Tata McGraw Hill Education Private Limited,  
7 West Patel Nagar, New Delhi 110 008.

## *Dedicated to my*

*Brother-in-law Mr. K. Abdul Rawoof, M.A.*

*Sister Mrs. A. Mumtaj Rawoof, B.Sc. and*

*Their Daughter Dr. A. Shafela Sherin, B.D.S. and*

*Son Dr. A. Mohammed Fazil, M.B.B.S.*



# CONTENTS

<i>Preface</i> .....	<i>xii</i>
<i>Acknowledgements</i> .....	<i>xiii</i>

## CHAPTER - 1 INTRODUCTION TO MICROPROCESSOR

1.1 Terms used in microprocessor literature	1. 1
1.2 Evolution of microprocessor	1. 3
1.3 Basic functional blocks of a microprocessor	1. 6
1.4 Microprocessor-based system (Organization of microcomputer)	1. 7
1.5 Concept of multiplexing in microprocessor	1. 9
1.6 Micro, Mini and Large computers	1. 9
1.7 INTEL 8085	1.12
1.8 INTEL 8086	1.20
1.9 ZILOG Z80	1.29
1.10 MOTOROLA 6800	1.33
1.11 Summary	1.36
1.12 Short questions and answers	1.38

## CHAPTER - 2 INSTRUCTION SET OF 8085

2.1 Origin of software	2. 1
2.2 Processor cycles	2. 1
2.3 Machine cycles of 8085	2. 1
2.3.1 Timing diagram	2. 2
2.4 Instruction format of 8085	2.11
2.5 Addressing modes	2.11
2.6 Instruction set	2.12

2.7	<i>Data transfer instructions</i>	2.21
2.8	<i>Arithmetic instructions</i>	2.29
2.9	<i>Logical instructions</i>	2.36
2.10	<i>Branching instructions</i>	2.43
2.11	<i>Machine control instructions</i>	2.46
2.12	<i>Timing diagram of 8085 instructions</i>	2.48
2.13	<i>Summary</i>	2.57
2.14	<i>Short questions and answers</i>	2.57

### CHAPTER - 3 MEMORY AND IO INTERFACING

3.1	<i>Introduction to memory</i>	3. 1
3.2	<i>Semiconductor memory</i>	3. 1
3.3	<i>ROM and PROM</i>	3. 3
3.4	<i>EPROM</i>	3. 4
3.5	<i>Static RAM</i>	3. 7
3.6	<i>DRAM and NVRAM</i>	3.11
3.7	<i>Interfacing static RAM and EPROM</i>	3.12
3.8	<i>Memory organization in 8085-based system</i>	3.16
3.9	<i>IO structure of a typical microcomputer</i>	3.19
3.10	<i>Interfacing IO and peripheral devices</i>	3.20
3.11	<i>Summary</i>	3.37
3.12	<i>Short questions and answers</i>	3.38

### CHAPTER - 4 INTERRUPTS

4.1	<i>Interrupt and its need</i>	4. 1
4.2	<i>Classification of interrupts</i>	4. 2
4.3	<i>Interrupts of 8085</i>	4. 3
4.3.1	<i>Software interrupts of 8085</i>	4. 3
4.3.2	<i>Hardware interrupts of 8085</i>	4. 4
4.3.3	<i>Priorities of interrupts of 8085</i>	4. 5
4.4	<i>Enabling, disabling and masking of 8085 interrupts</i>	4. 5



4.5	<i>Polling of interrupts</i>	4. 7
4.6	<i>INTR and its expansion</i>	4.11
4.7	<i>Programmable interrupt controller - INTEL 8259</i>	4.12
4.7.1	<i>Interfacing 8259 with 8085 microprocessor</i>	4. 13
4.7.2	<i>Functional block diagram of 8259</i>	4. 15
4.7.3	<i>Processing of interrupts by 8259</i>	4. 17
4.7.4	<i>Programming 8259(or initializing 8259)</i>	4. 18
4.8	<i>Summary</i>	4.22
4.9	<i>Short questions and answers</i>	4.23

## CHAPTER - 5 ASSEMBLY LANGUAGE PROGRAMMING

5.1	<i>Levels of programming</i>	5. 1
5.2	<i>Flowchart</i>	5. 2
5.3	<i>Assembly language program development tools</i>	5. 3
5.4	<i>Variables and constants used in assemblers</i>	5. 8
5.5	<i>Assembler directives</i>	5. 9
5.6	<i>Procedure and Macro</i>	5.11
5.7	<i>Delay routine</i>	5.12
5.8	<i>List and array</i>	5.15
5.9	<i>Stack</i>	5.16
5.10	<i>Examples of 8085 assembly language programs</i>	5.17
5.11	<i>Summary</i>	5.63
5.12	<i>Short questions and answers</i>	5.64

## CHAPTER - 6 PERIPHERAL DEVICES AND INTERFACING

6.1	<i>Programmable peripheral devices</i>	6. 1
6.2	<i>Parallel data communication interface</i>	6. 1
6.2.1	<i>Parallel data transfer schemes</i>	6.2
6.2.2	<i>INTEL 8212</i>	6.4
6.2.3	<i>Programmable IO port and timer - INTEL 8155/8156</i>	6.6

x

6.2.4	Programmable peripheral interface - INTEL 8255	6.12
6.2.5	Programmable IO port and memory - INTEL 8355	6.20
6.2.6	Programmable IO port and memory - INTEL 8755	6.21
6.2.7	DMA data transfer scheme	6.22
6.2.8	DMA controller - INTEL 8237	6.24
6.2.9	DMA controller - INTEL 8257	6.37
6.3	Serial data communication interface	6.45
6.3.1	Serial data communication	6.45
6.3.2	USART - INTEL 8251A	6.50
6.4	Keyboard and display interface	6.55
6.4.1	Keyboard interface using ports	6.55
6.4.2	Display interface using ports	6.58
6.4.3	Latches and buffers as IO devices	6.63
6.4.4	Keyboard/Display controller - INTEL 8279	6.65
6.4.5	Keyboard and display interface using 8279	6.69
6.5	Programmable timer - INTEL 8254	6.72
6.6	DAC interface	6.82
6.6.1	DAC0800	6.84
6.7	ADC interface	6.86
6.7.1	ADC0809	6.88
6.8	Summary	6.92
6.9	Short questions and answers	6.94

## CHAPTER - 7 MICROCOMPUTER SYSTEM DESIGN AND APPLICATIONS

7.1	Designing a microprocessor-based system	7. 1
7.2	8085-based minimum system	7. 3
7.3	Temperature control system	7. 5
7.4	Motor speed control system	7. 7

7.5	<i>Traffic light control system</i>	7. 9
7.6	<i>Stepper motor control system</i>	7.13
<i>APPENDIX I 8085A instructions in hexadecimal order</i>		<i>A. 1</i>
<i>APPENDIX II 8085A instructions in alphabetical order</i>		<i>A. 3</i>
<i>APPENDIX III List of microprocessors released by INTEL</i>		<i>A. 5</i>
<i>GENERAL INDEX</i>		<i>I. 1</i>
<i>CHIP INDEX</i>		<i>I. 5</i>

# Preface

The main objective of this book is to explore the basic concepts of most popular INTEL 8085 microprocessor and its programming and interfacing techniques in a simple and easy-to-understand manner.

This text on 8085 microprocessor has been crafted and designed to meet student's requirements. Considering the complex technical nature of this subject, equal emphasis has been given to programming and design aspects. Considerable effort has been made to explain the assembly language programs with step-by-step algorithm and flowchart. The peripheral interfacing techniques has been explained with simple sketches clearly showing the necessary signals. Short questions and answers with varied difficulty levels are given in the text to help students get an intuitive grasp on the subject.

This book with its lucid writing style and germane pedagogical features will prove to be a master text for engineering students and practitioners.

The chapter-1 briefs about evolution of microprocessor and basics of microprocessor-based system. An introductory discussion on popular microprocessors 8085, 8086, Z80 and 6800 are presented in chapter-1.

The machine cycles of 8085 processor and their timing diagrams are discussed in chapter-2. Also the instructions of 8085 microprocessor are explained with example in chapter-2.

The details of semiconductor memory and their interfacing with 8085 microprocessor are presented in chapter-3. Design examples are also included for better understanding of the concept of the memory and IO interfacing with 8085 microprocessor.

The importance of interrupts and their implementation in 8085 system are discussed in chapter-4. Methods for expanding the interrupts of 8085 are also discussed in chapter-4.

The concepts of assembly language programming are discussed in chapter-5. A number of assembly language example programs using 8085 microprocessor instructions are included in this chapter. These example programs are assembled using X8085A assembler and verified in RBA-8085 trainer kit.

The concepts of port, keyboard and 7-segment display and their interfacing are discussed in chapter-6. Simple discussions about USART, DMA controllers, Programmable timer, ADC and DAC, and their interfacing with 8085 microprocessor are also presented in chapter-6.

Some of the 8085 microprocessor-based systems for specific applications are discussed in chapter-7. The 8085 microprocessor instructions along with their opcodes are listed in Appendix-I and Appendix-II as ready reference for assembly language programmers.

The author has taken care to present the concepts of microprocessor in a way easy to grasp by students. The readers can feel free to convey their criticism and suggestions to [kani@vsnl.com](mailto:kani@vsnl.com) for further improvement of the book.

**A. Nagoor Kani**

# Acknowledgements

I express my heartfelt thanks to my wife Ms C Gnanaparanjothi Nagoor Kani and my sons N Bharath Raj alias Chandrakani Allaudeen and N Vikram Raj for the support, encouragement and cooperation they have extended to me throughout my career.

It is my pleasure to acknowledge the contributions to our technical editors,

Ms B Hemavathy, Ms S Pavithra, Ms K Thangaselvi for editing and proofreading of the manuscript, and Ms A Selvi, Ms M Faritha for type setting and preparing the layout of the book.

My sincere thanks to all reviewers for their valuable suggestions and comments which helps me to explore the subject to greater depth.

.....  
.....  
.....  
.....  
.....

I am also grateful to Ms Vibha Mahajan, Mr Ebi John, Ms Smruti Snigha, Ms Nimisha Kapoor, Ms Koyel Ghosh, Mr P L Pandita, Ms Sohini Mukherjee, ..... and ..... of Tata McGraw Hill Education for their concern and care in publishing this work.

My special thanks to Ms. Smruti Snigha of McGraw Hill Education for her care in bringing out this work at the right time.

I thank all my office staff for their co-operation in carrying out my day-to-day activities.

Finally, a special note of appreciation is due to my sisters, brothers, relatives, friends, students and the entire teaching community for their overwhelming support and encouragement to my writing.

**A. Nagoor Kani**



# INTRODUCTION TO MICROPROCESSOR

## 1.1 TERMS USED IN MICROPROCESSOR LITERATURE

<i>Bit</i>	: A digit of the binary number or code is called bit.
<i>Nibble</i>	: The 4-bit (4-digit) binary number or code is called nibble.
<i>Byte</i>	: The 8-bit (8-digit) binary number or code is called byte.
<i>Word</i>	: The 16-bit (16-digit) binary number or code is called word.
<i>Double Word</i>	: The 32-bit (32-digit) binary number or code is called double word.
<i>Multiple Word</i>	: The 64, 128, ... bit/digit binary numbers or codes are called multiple words.
<i>Data</i>	: The quantity (binary number/code) operated by an instruction of a program is called data. The size of data is specified as bit, byte, word, etc.
<i>Address</i>	: Address is an identification number in binary for memory locations. The 8086 processor uses 20-bit address for memory.
<i>Memory Word Size (or Addressability)</i>	: The memory word size or addressability is the size of binary information that can be stored in a memory location. The memory word size for 8085 processor-based system is 8-bit.

*[The address and program codes in microprocessor system are given in binary (i.e., as a combination of "0" and "1"). With  $n$ -bit binary we can generate  $2^n$  different binary codes or address.]*

<i>Microprocessor</i>	: The microprocessor is a program controlled semiconductor device(IC), which fetches (from memory), decodes and executes instructions. It is used as CPU (Central Processing Unit) in computers.
-----------------------	--

The basic functional blocks of a microprocessor are ALU (Arithmetic Logic Unit), an array of registers and a control unit. The microprocessor is identified with the size of data, and the ALU of the processor it can work with at a time. The 8085 processor has 8-bit ALU, hence it is called 8-bit processor. The 8086 processor has 16-bit ALU, hence it is called 16-bit processor.

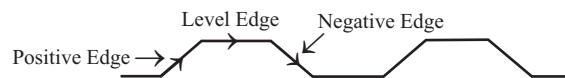
<i>Bus</i>	: A bus is a group of conducting lines that carries data, addresses and control signals. Buses can be classified into Data bus, Address bus and Control bus. The group of conducting lines that carries data is called data bus. The group of conducting lines that carries addresses is called address bus. The group of conducting lines that carries control signals is called control bus.
------------	--

- CPU Bus** : The group of conducting lines that are directly connected to microprocessor is called CPU bus. In a CPU bus the signals are multiplexed, i.e., more than one signal are passed through the same line but at different timings.
- System Bus** : The group of conducting lines that carries data, addresses and control signals in a microcomputer system is called system bus. Multiplexing is not allowed in system bus.

*[In microprocessor-based systems, each bit of information (data/address/control signal) is send through a separate conducting line. Due to practical limitations, the manufacturers of microprocessors may provide multiplexed pins, i.e., one pin is used for more than one purpose. This leads to multiplexed CPU bus. For example in 8085 processor the low byte of address and data are send through same pins but at different timings. But when the system is formed, the multiplexed bus lines should be demultiplexed by using latches, ports, transceivers, etc. The demultiplexed bus lines are called system bus. In a system, separate conducting line will be provided for each bit of data, address and control signals.]*

- Clock** : A clock is a square wave, which is used to synchronize various devices in the microprocessor and in the system. Every microprocessor system requires a clock for its functioning. The time taken for the microprocessor and the system to execute an instruction or program are measured only in terms of the time period of its clock.

The clock has three edges. They are: rising edge (positive edge), level



edge and falling edge (negative edge). The devices are made sensitive to any one of the edge for better functioning i.e., the device will recognize the clock only when the edge is asserted or arrived).

- Tristate Logic** : Almost all the devices used in the microprocessor-based system uses tristate logic. In devices with tristate logic, three logic levels will be available. They are **high** state, **low** state and **high impedance** state.

The **high** and **low** are normal logic levels for data, address or control signals. The **high impedance** state is electrical open circuit condition. The **high impedance** state is provided to keep the device electrically isolated from the system. The tristate devices will normally remain in **high impedance** state and their pins are physically connected in the system bus but electrically isolated. In **high impedance** state, they cannot receive or send any signal or information. These devices are provided with chip enable/ chip select pins. When the signal at this pin is asserted to the right level, they come out from **high impedance** state to normal levels.



## 1.2 EVOLUTION OF MICROPROCESSOR

History shows us that the ancient Babylonians first began using the abacus (a primitive calculator made of beads) in about 500 BC. This simple calculating machine eventually sparked human mind into the development of calculating machinery that uses gears and wheels (Blaise Pascal in 1642). The giant computing machines of the 1940s and 1950s were constructed with relays and vacuum tubes. Next, the transistor and solid-state electronics were used to build the mighty computers of the 1960s. Finally, the advent of the Integrated Circuit (IC) led to the development of the microprocessor and microprocessor-based computer system.

In 1971, INTEL corporation released the world's first microprocessor - the INTEL 4004, a 4-bit microprocessor. It addresses 4096 memory locations of word size 4-bit. The instruction set consists of 45 different instructions. It is a monolithic IC employing large scale integration in PMOS Technology. The INTEL 4004 was soon followed by a variety of microprocessors, with most of the major semiconductor manufacturers producing one or more types.

### First Generation Microprocessors

The microprocessors introduced between 1971 and 1973 were the first generation processors. They were designed using PMOS technology. This technology provided low cost, slow speed and low output currents and was not compatible with TTL (Transistor Transistor Logic) levels.

The first generation processors require a lot of additional support of ICs to form a system. They may require as high as 30 ICs to form a system. The 4-bit processors are provided with only 16 pins, but 8-bit and 16-bit processors are provided with 40 pins. Due to limitations of pins, the signals are multiplexed. A list of first generation microprocessors are given below:

- |                              |   |                   |
|------------------------------|---|-------------------|
| • INTEL 4004                 | } | 4-bit processors  |
| • INTEL 4040                 |   |                   |
| • FAIR CHILD PPS - 25        |   |                   |
| • NATIONAL IMP - 4           |   |                   |
| • ROCKWELL PPP - 4           |   |                   |
| • MICRO SYSTEMS INTL. MC - 1 |   |                   |
| • INTEL 8008                 | } | 8-bit processors  |
| • NATIONAL IMP - 8           |   |                   |
| • ROCKWELL PPS - 8           |   |                   |
| • AMI 7200                   |   |                   |
| • MOSTEK 5065                | } | 16-bit processors |
| • NATIONAL IMP/16            |   |                   |
| • NATIONAL PACE              |   |                   |

### Second Generation Microprocessors

The second generation microprocessors appeared in 1973 and were manufactured in NMOS Technology. The NMOS technology offers faster speed and higher density than PMOS and it is TTL compatible. Some of the second generation processors are given below:

- |                   |   |                  |
|-------------------|---|------------------|
| • INTEL 8080      | } | 8-bit processors |
| • INTEL 8085      |   |                  |
| • FAIRCHILD F - 8 |   |                  |

• MOTOROLA M6800	}	8-bit processors
• MOTOROLA M6809		
• NATIONAL CMP -8		
• RCA COSMAC		
• MOS TECH. 6500		
• SIGNETICS 2650		
• ZILOG Z80	}	12-bit processors
• INTERSIL 6100		
• TOSHIBA TLCS - 12		
• TI TMS 9900	}	16-bit processors
• DEC - WD MCP - 1600		
• GENERAL INSTRUMENT CP 1600		
• DATA GENERAL $\mu$ N601		

### Characteristics of second generation microprocessors

- Larger chip size ( $170 \times 200$  mils). [1mil =  $10^{-3}$ inch]
- 40 pins.
- More numbers of on-chip decoded timing signals.
- The ability to address large memory spaces.
- The ability to address more IO ports.
- Faster operation.
- More powerful instruction set.
- A greater number of levels of subroutine nesting.
- Better interrupt handling capabilities.

### Third Generation Microprocessors

After 1978, the third generation microprocessors were introduced. These are 16-bit processors and designed using HMOS (High density MOS) Technology. Some of the third generation microprocessors are given below:

- |               |                  |                               |
|---------------|------------------|-------------------------------|
| • INTEL 8086  | • INTEL 80286    | • ZILOG Z8000                 |
| • INTEL 8088  | • MOTOROLA 68000 | • NATIONAL NS 16016           |
| • INTEL 80186 | • MOTOROLA 68010 | • TEXAS INSTRUMENTS TMS 99000 |

The HMOS technology offers better Speed Power Product (SPP) and higher packing density than NMOS.

$$\begin{aligned}
 \text{Speed Power Product} &= \text{Speed} \times \text{Power} \\
 &= \text{Nanosecond} \times \text{Milliwatt} \\
 &= \text{Picojoules}
 \end{aligned}$$

- Speed Power Product of HMOS is four times better than NMOS.  
SPP of NMOS = 4 picojoules (pJ)  
SPP of HMOS = 1 picojoules (pJ)
- Circuit densities provided by HMOS are approximately twice than those of NMOS.  
Packing density of NMOS = 1852.5 gates/mm<sup>2</sup>  
Packing density of HMOS = 4128 gates/mm<sup>2</sup> (1 mm =  $10^{-6}$  meter)

### **Characteristics of third generation microprocessors**

- Provided with 40/48/64 pins.
- High speed and very strong processing capability.
- Easier to program.
- Allow for dynamically relocatable programs.
- Size of internal registers are 8/16/32 bits.
- The processor has multiply/divide arithmetic hardware.
- Physical memory space is from 1 to 16 Mega bytes.
- The processor has segmented addresses and virtual memory features.
- More powerful interrupt handling capabilities.
- Flexible IO port addressing.
- Different modes of operations (e.g., user and supervisor modes of M68000).

### **Fourth Generation Microprocessors**

The fourth generation microprocessors were introduced in the year 1980. The fourth generation processors are 32-bit processors and are fabricated using the low-power version of the HMOS technology called the HCMOS. These 32-bit microprocessors have increased sophistications that compete strongly with the mainframes. Some of the fourth generation microprocessors are given below:

- INTEL 80386
- INTEL 80486
- NATIONAL NS16032
- MOTOROLA M68020
- BELLMAC - 32
- MOTOROLA M68030
- MOTOROLA MC88100

### **Characteristics of fourth generation microprocessors**

- Physical memory space of  $2^{24}$  bytes = 16 Mb (Mega bytes).
- Virtual memory space of  $2^{40}$  bytes = 1 Tb (Tera bytes).
- Floating point hardware is incorporated.
- Supports increased number of addressing modes.

### **Fifth Generation Microprocessors**

In microprocessor technology, INTEL has taken a leading edge and is developing more and more new processors. The INTEL **pentium** processor released in the year 1993 is considered as a fifth generation processor. The pentium is 32-bit processor with 64-bit data bus and available in wide range of clock speeds from 60 MHz to 3.2 GHz. With the improvement in semiconductor technology, the processing speed of microprocessors have been increased tremendously. The 8085 released in the year 1976 executes 0.5 Million Instructions Per Second (0.5 MIPS). The 80486 executes 54 Million Instructions Per Second. The pentium is optimized to execute two instructions in one clock period. Therefore, a pentium processor working at 1GHz clock can execute 2000 Million Instructions Per Second (2000 MIPS). The various processors released by INTEL are listed in Appendix-III.

### **Applications of Microprocessors of Different Generations**

#### **First generation processor**

- Calculators
- Game machines
- Home appliances
- Accounting systems
- Intelligent instrumentation
- Low and special purpose applications

**Second generation processors**

- Complex industrial controllers
- Communication perprocessors
- Process control systems
- Data acquisition systems
- Instrumentation
- Intelligent terminals
- Military applications

**Third generation processors**

- Business and data processing applications
- Sophisticated real time control
- Advanced communications
- Distributed processing networks

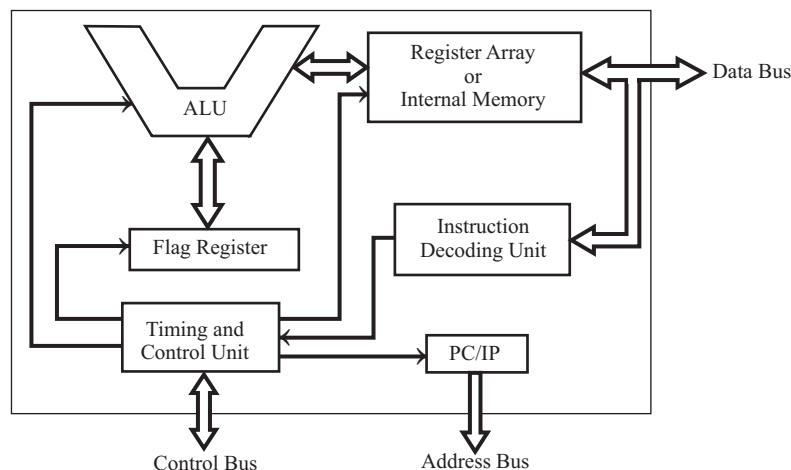
**Fourth generation processors**

- General purpose computing in applications requiring mainframe type computing power.
- Multiuser, multifunction environments.
- Office information equipment.

**1.3 BASIC FUNCTIONAL BLOCKS OF A MICROPROCESSOR**

The microprocessor is a programmable IC which is capable of performing arithmetic and logical operations. The basic functional block diagram of a microprocessor is shown in Fig. 1.1.

The basic functional blocks of a microprocessor are ALU, flag register, register array, Program Counter (PC)/Instruction Pointer (IP), instruction decoding unit, timing and control unit.



**Fig. 1.1** : Block diagram showing basic functional blocks of a microprocessor.

ALU is the computational unit of the microprocessor which performs arithmetic and logical operations on binary data. The various conditions of the result are stored as status bits called flags in the flag register. For example, consider a sign flag, one of the bit position of flag register is called sign flag and it is used to store the status of sign of the result of ALU operation (output data of ALU). If the result is negative, then "1" is stored in the sign flag and if the result is positive, then "0" is stored in the sign flag.

The register array is the internal storage device and so it is also called *internal memory*. The input data for ALU, the output data of ALU (result of computations) and any other binary information needed for processing are stored in the register array.

For any microprocessor, there will be a set of instructions given by the manufacturer of the microprocessor. For doing any useful work with the microprocessor, we have to write a program using these instructions, and store them in a memory device external to the microprocessor.

The program counter generates the address of the instructions to be fetched from the memory and send through address bus to the memory. The memory will send the instruction codes and data through the data bus. The instruction codes are decoded by the decoding unit and send information to timing and control unit. The data is stored in the register array for processing by ALU.

The control unit will generate the necessary control signals for internal and external operations of the microprocessor.

#### 1.4 MICROPROCESSOR-BASED SYSTEM (ORGANIZATION OF MICROCOMPUTER)

The microprocessor is a semiconductor device (or Integrated Circuit) manufactured by the VLSI (Very Large Scale Integration) technique. It includes the ALU, register arrays and control circuit on a single chip. To perform a function or useful task, we have to form a system by using microprocessor as a CPU (Central Processing Unit) and interfacing memory, input and output devices to it. A system designed using a microprocessor as its CPU is called a *microcomputer or single board microcomputer*. The microprocessor-based system consists of microprocessor as CPU, semiconductor memories like EPROM and RAM, input device, output device and interfacing devices. The memories, input device, output device and interfacing devices are called *peripherals*.

The commonly used EPROM and static RAM in microcomputers are given below:

##### EPROM

INTEL 2708 (1 kb)  
INTEL 2716 (2 kb)  
INTEL 2732 (4 kb)  
INTEL 2764 (8 kb)

##### Static RAM

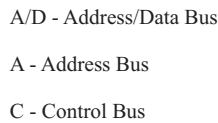
MOTOROLA 6208 (1 kb)  
MOTOROLA 6216 (2 kb)  
MOTOROLA 6232 (4 kb)  
MOTOROLA 6264 (8 kb)

*Note : kb refer to Kilo bytes.*

The popular input devices are keyboard, floppy disk, etc., and the output devices are printer, LED/LCD displays, CRT monitor, etc.

The block diagram of microprocessor-based system (organization of microcomputer) is shown in Fig. 1.2. In this system, the microprocessor is the master and all other peripherals are slaves. The master controls all the peripherals and initiates all operations.

The buses are group of lines that carry data, addresses or control signals. The CPU bus has multiplexed lines, i.e., the same line is used to carry different signals. The CPU interface is provided to demultiplex the multiplexed lines, to generate chip select signals and additional control signals. The system bus has separate lines for each signal.



**Fig. 1.2 :** Microprocessor-based system (organization of microcomputer).

All the slaves in the system are connected to the same system bus. At any time instant communication takes place between the master and one of the slaves. All the slaves have tristate logic and hence normally remains in **high impedance** state. The processor selects a slave by sending an address. When a slave is selected, it comes to the normal logic and communicates with the processor.

The EPROM memory is used to store permanent programs and data. The RAM memory is used to store temporary programs and data. The input device is used to enter the program, data and to operate the system. The output device is also used for examining the results. Since the speed of IO devices does not match with the speed of microprocessor, an interface device is provided between system bus and IO devices. Generally, IO devices are slow devices.

The work done by the processor can be classified into the following three groups :

1. Work done internal to the processor.
2. Work done external to the processor.
3. Operations initiated by the slaves or peripherals.

The work done internal to the processor are addition, subtraction, logical operations, data transfer within registers, etc. The work done external to the processor are reading/writing the memory, and reading/writing the IO devices or the peripherals. If the peripheral requires the attention of the master, then it can interrupt the master and initiate an operation.

The microprocessor is the master, which controls all the activities of the system. To perform a specific job or task, the microprocessor has to execute a program stored in memory. The program consists of a set of instructions stored in consecutive memory location. In order to execute the program, the microprocessor issues address and control signals, to fetch the instruction and data from memory one by one. After fetching each instruction it decodes the instruction and carries out the task specified by the instruction.

## 1.5 CONCEPT OF MULTIPLEXING IN MICROPROCESSOR

Multiplexing is transferring different information at different well-defined times through same lines. A group of such lines are called multiplexed bus. The result of multiplexing is that fewer pins are required for microprocessors to communicate with the outside world.

Due to the pin number limitations, most microprocessors cannot provide simultaneously similar lines (such as address, data, status signals, etc.). Hence, multiplexing of one or more of these buses is performed. Most often data lines are multiplexed with some or all address lines to form an address/data bus. (For example, in 8085, the lower 8 address lines are multiplexed with data lines.) The status signals emitted by the microprocessor are sometimes multiplexed either with the data lines (as done in INTEL 8080A) or with some of the address lines (as done in the INTEL 8086).

Whenever multiplexing is used, the CPU interface of the system must include the necessary hardware to demultiplex those lines to produce the separate address, data and control buses required for the system. Demultiplexing of a multiplexed bus can be handled either at the CPU interface or locally at appropriate points in the system. Besides a slower system operation, a multiplexed bus also results in additional interface hardware requirements.

### Demultiplexing of Address/Data Lines in 8085 Processor

In order to demultiplex the address/data lines (of the processor), the processor provides a signal called ALE (Address Latch Enable). The ALE is asserted **high** and then **low** by the processor at the beginning of every machine cycle. At the same time the low byte address is given out through  $AD_0 - AD_7$  lines. The demultiplexing of address/data lines using 8-bit D-latch 74LS373 is shown in Fig. 1.3.

The ALE is connected to the **Enable Pin (EN)** of an external 8-bit latch. When ALE is asserted **high** and then **low**, the addresses are latched into the output lines of the latch. It holds the low byte of the address until the next machine cycle. After latching the address, the  $AD_0 - AD_7$  lines are free for data transfer. The first T-state of every machine cycle is used for address latching in 8085 and the remaining T states are used for reading or writing operation.

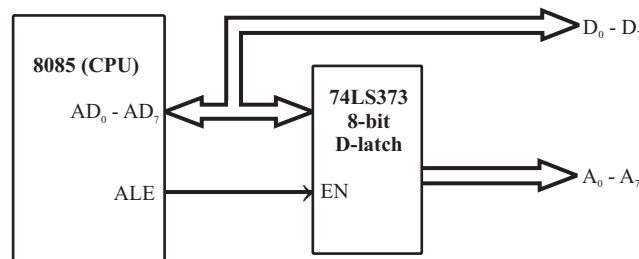


Fig. 1.3 : Demultiplexing of address and data lines in an 8085 processor.

## 1.6 MICRO, MINI AND LARGE COMPUTERS

The computers are broadly classified into three categories : Main frame, Mini and Microcomputers.

In today's technology, the distinction between these categories are fast vanishing. The microcomputers have superseded the performance of minicomputers and now they are competing with mainframes.

### **Mainframes (Large Computers)**

The largest and most powerful computers are called mainframes. Mainframe computers may occupy an entire room. They are designed to work at very high speed with large data words and have massive amounts of memory. Computers of this type are used for military defence control, business data processing and for creating computer graphics displays for science fiction movies. Examples of this type of computers are IBM 4381, Honeywell DPS8 and CRAY X-MP/48.

### **Minicomputers**

The scaled-down versions of mainframe computers are often called minicomputers. The main unit of a minicomputer usually fits in a single rack or box. A minicomputer runs more slowly, work directly with smaller data words and does not have as much memory as a mainframe. Computers of this type are used for business data processing, industrial control and scientific research. Examples of this type of computers are the Digital Equipment Corp.-VAX 11/730, the Data General-MV/800011 and HCL's-MAGNUM.

The CPU of the minicomputers have more than one microprocessor and their coprocessors.

### **Microcomputers**

As the name implies, microcomputers are small computers. They range from small controllers that work directly with 4-bit words and can address a few thousand bytes of memory to large units that work directly with 64-bit words and can address millions or billions of bytes of memory.

Some of the more powerful microcomputers have all or most of the features of earlier minicomputers. Therefore, it has become very hard to draw a sharp line between these two types.

One distinguishing feature of a microcomputer is that the CPU is usually a single microprocessor. The examples of microcomputer are IBM PC/80, AT286, AT386, AT486, etc.

### **Comparison of Large, Mini and Micro Computers**

The computers can be compared based on the following features:

1. Speed of execution
2. Size of data
3. Memory capacity it can support
4. Types of data
5. IO devices and peripheral support devices
6. Applications

They are further listed in Table-1.1.

### **Advantages of Microprocessor-Based System**

1. Computational or processing speed is high.
2. Intelligence has been brought to systems.
3. Automation of industrial processes and office administration.
4. Since the devices are programmable, there is flexibility to alter the system by changing the software alone.
5. Less number of components, compact in size and less cost. It is also more reliable.
6. Both operation and maintenance are easier.



**TABLE - 1.1 : COMPARISON OF MAINFRAMES, MINICOMPUTERS AND MICROCOMPUTERS**

Features	Mainframe / Large computers	Minicomputers	Microcomputers
1. Speed of execution	Executes 100 to 3000 Million Instructions <b>Per Second</b> (MIPS).	Executes up to 100 Million Instructions <b>Per Second</b> (MIPS).	Ranges from 0.5 MIPS (8085) to 54 MIPS (80486)
2. Size of data	64 or 128 bits.	32 or 64 bits.	4 bits to 64 bits.
3. Memory capacity	Thousands of Gb.	In the range of few Gb.	Ranges from kb to Gb (kb-kilo-byte, Gb-Giga-byte)
4. Types of data	All types (Binary, ASCII, decimal, floating point data, complex numbers, etc.).	Same as main frame.	Depends on the micro-processor used as CPU. 8086 - Binary, BCD, ASCII.
5. IO devices and support peripherals	All types of IO devices and peripheral devices will be interfaced in the system.	Common input devices are tapes, floppies, etc. Types of output device depends on the user needs.	Common IO devices are keyboard, floppies and output devices are CRT and printer. The support peripheral device depends on applications of the system.
6. Applications	Scientific calculation, weather data processing, large business data processing, real time applications, multiuser business systems, instructional system in colleges.	Multiuser business systems, word processing, process control, office automation, hospital patient care systems.	Personal computing, calculators, small business systems, control applications, instrumentation systems.

### Disadvantages of Microprocessor-Based System

1. It has limitations on the size of data.
2. The applications are limited by the physical address space.
3. The analog signals cannot be processed directly and digitizing the analog signals introduces errors.
4. The speed of execution is slow and so real time applications are not possible.
5. Most of the microprocessors do not support floating point operations.

### Applications of Microprocessor-Based Systems

Typical applications for different systems, categorized by the data size of ALU of the processor, include the following :

#### 4-bit systems

- |  |  |
|--|--|
| <ul style="list-style-type: none"> <li>● Accounting systems</li> <li>● Home Appliances</li> <li>● Calculators</li> </ul> | <ul style="list-style-type: none"> <li>● Game Machines</li> <li>● Intelligent Instrumentation</li> <li>● Terminals (simple)</li> </ul> |
|--|--|

#### 8-bit systems

- |   |  |
|---|--|
| <ul style="list-style-type: none"> <li>● Control systems</li> <li>● Intelligent terminals and instruments</li> <li>● Process control systems</li> </ul> | <ul style="list-style-type: none"> <li>● Point of sale terminals</li> <li>● Traffic controllers</li> <li>● Communication preprocessors (data concentrators)</li> </ul> |
|---|--|

#### 16-bit systems

- |   |   |
|---|---|
| <ul style="list-style-type: none"> <li>● Data acquisition system</li> <li>● Intelligent terminals</li> <li>● Automatic testing systems</li> </ul> | <ul style="list-style-type: none"> <li>● Numerical control</li> <li>● Process control</li> <li>● Supervisory control (gas, power, water distribution) and many more applications</li> </ul> |
|---|---|

## 1.7 INTEL 8085

The INTEL 8085 is an 8-bit microprocessor released in the year 1976. The 8085 is designed using NMOS technology and now it is manufactured using HMOS technology and it contains approximately 6500 transistors. The 8085 is packed in a 40-pin DIP (**D**ual **I**n-line **P**ackage) and requires a single 5-volt supply.

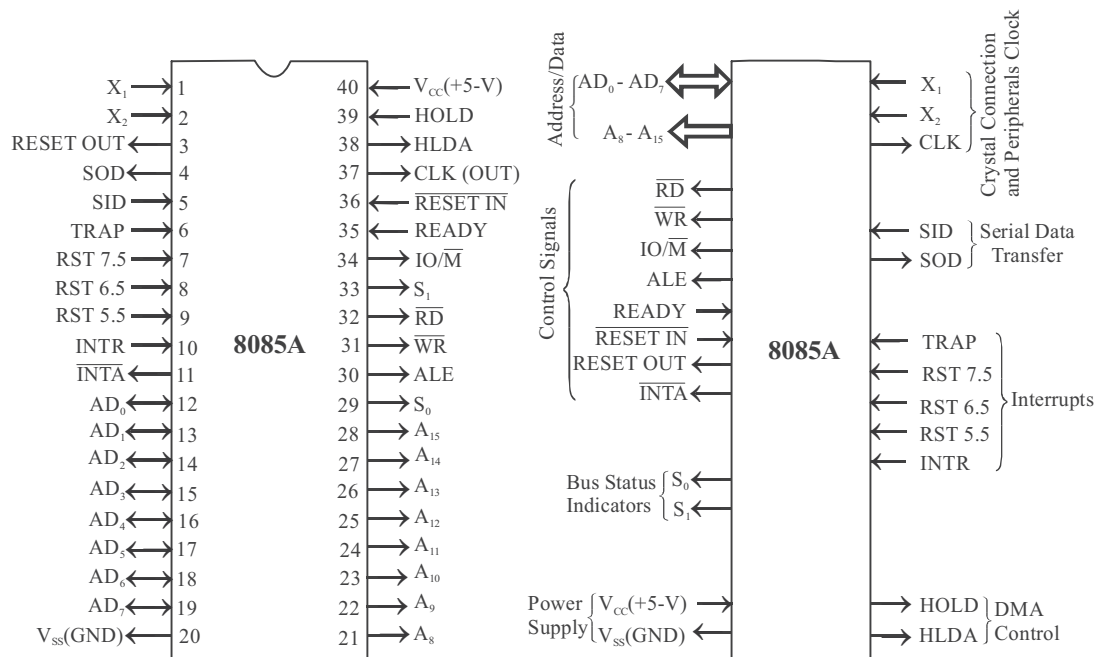
The 8085 has an internal clock oscillator. It generates a clock signal internally and divides by two and then uses as internal clock. This internal clock is also given out through CLK pin for the clock requirement of peripheral devices.

The NMOS 8085 is available in two versions : 8085A and 8085A-2 with maximum internal clock frequency of 3.03MHz and 5MHz, respectively. The enhanced version of 8085 is designed with HMOS transistors. It is available in three versions: 8085AH, 8085AH-2 and 8085AH-1 with maximum internal clock of 3MHz, 5MHz and 6MHz, respectively.

The basic data size of 8085 is 8-bit. Therefore, the memory word size of the memories interfaced with 8085 processor is also 8-bit or byte. The 8085 uses a 16-bit address to access memory and hence it can address upto  $2^{16} = 65,536_{10} = 64 \text{ k}$  memory locations. Since one-byte of

information can be stored in one memory location, the maximum memory capacity of 8085-based system is 64 kilo bytes. For accessing IO-mapped devices, the 8085 uses a separate 8-bit address and so it can generate  $2^8=256_{10}$  IO addresses.

The pin configuration of 8085 is shown in Fig. 1.4. The signals of 8085 are listed in Table-1.2. The 8085 has 8 pins AD<sub>0</sub> to AD<sub>7</sub> for data transfer, which are multiplexed with low byte of address. The 8085 provides a signal ALE (**A**ddress **L**atch **E**nable) to demultiplex the low byte address and data using an external latch. The demultiplexing of address and data lines in 8085 is shown in Fig. 1.3 in Section 1.5.



**Fig. 1.4 :** 8085 microprocessor signals and pin assignment.

During memory access, the 16-bit memory address are output on  $AD_0$  to  $AD_7$  and  $A_8$  to  $A_{15}$  lines. During IO access of IO-mapped devices, the 8-bit IO address are output on both  $AD_0$  to  $AD_7$  and  $A_8$  to  $A_{15}$  lines. The 8085 processor differentiates the memory and IO address using the signal  $IO/\overline{M}$ . When the processor output gives a memory address, the  $IO/\overline{M}$  is asserted **low** and when the processor output gives an IO address, the  $IO/\overline{M}$  is asserted **high**.

The  $\overline{\text{RD}}$  signal is asserted **low** by the processor during a memory or IO read operation. The  $\overline{\text{WR}}$  signal is asserted **low** by the processor during a memory or IO write operation. The  $\text{S}_0$  and  $\text{S}_1$  are bus status indicators. The output signals on these lines during various bus activity (or machine cycles) are listed in Table-1.3.

**TABLE - 1.2 : 8085 SIGNAL DESCRIPTION SUMMARY**

Pin name	Description	Type
$AD_0 - AD_7$	Address/Data	Bidirectional, Tristate
$A_8 - A_{15}$	Address	Output, Tristate
ALE	Address latch enable	Output, Tristate
$\overline{RD}$	Read control	Output, Tristate
$\overline{WR}$	Write control	Output, Tristate
$IO/\overline{M}$	IO or memory indicator	Output, Tristate
$S_0, S_1$	Bus state indicators	Output
READY	Wait state request	Input
SID	Serial input data	Input
SOD	Serial output data	Output
HOLD	Hold request	Input
HLDA	Hold acknowledge	Output
INTR	Interrupt request	Input
TRAP	Nonmaskable interrupt request	Input
RST 5.5	Hardware vectored interrupt request	Input
RST 6.5	Hardware vectored interrupt request	Input
RST 7.5	Hardware vectored interrupt request	Input
INTA	Interrupt acknowledge	Output
RESET IN	System reset	Input
RESET OUT	Peripherals reset	Output
$X_1, X_2$	Crystal or RC connection	Input
CLK (OUT)	Clock signal	Output
$V_{cc}$	+5-V	Power supply
$V_{ss}$	Ground	Power supply

*Note : An overbar on the signal, indicates that it is active low (i.e., the signal is normally high and when the signal is activated it is low).*

**TABLE - 1.3 : BUS STATUS SIGNALS**

$IO/\overline{M}$	$S_1$	$S_0$	Operation performed by 8085
0	0	1	Memory write
0	1	0	Memory read
1	0	1	IO write
1	1	0	IO read
0	1	1	Opcode fetch
1	1	1	Interrupt acknowledge

The READY is an input signal that can be used by slow peripherals to get extra time in order to communicate with 8085. The 8085 will work only when READY is tied to logic **high**. Whenever READY is tied to logic **low**, the 8085 will enter a wait state. When the system has slow peripheral devices, additional hardware is provided in the system to make the READY input **low** during the required extra time while executing a machine cycle, so that the processor can remain in wait state during this extra time.

The HOLD and HLDA signals are used for the **Direct Memory Access (DMA)** type of data transfer. This type of data transfers are achieved by employing a DMA controller in the system. When DMA is required, the DMA controller will place a **high** signal on the HOLD pin of 8085. When HOLD input is asserted **high**, the processor will enter a wait state and drive all its tristate pins to **high impedance** state and send an acknowledge signal to DMA controller through HLDA pin. Upon receiving the acknowledge signal, the DMA controller will take control of the bus and perform DMA transfer, and at the end it asserts HOLD signal **low**. When HOLD is asserted **low**, the processor will resume its execution.

The 8085 has five interrupt pins. The order of priority of the interrupts is TRAP, RST 7.5, RST 6.5, RST 5.5 and INTR. The interrupts TRAP, RST 7.5, RST 6.5 and RST 5.5 are hardware vectored interrupt and are enabled by appropriate signals at the appropriate pins of 8085. When a vectored interrupt is enabled and if it is accepted, then the program execution branches to vector addresses specified by INTEL. The interrupts RST 7.5, RST 6.5 and RST 5.5 are maskable interrupts by software.

The INTR is enabled by appropriate signal at its pin. In order to service INTR, one of the eight opcodes (RST 0 to RST 7) has to be provided on the  $AD_0 - AD_7$  bus by external logic. The 8085 then executes this instruction and vectors to the appropriate address to service the interrupt. The vector address for an interrupt RST  $n$  is given by  $(08 \times n)_H$ . The vector addresses of the interrupts of 8085 are listed in Table-1.4. (The interrupt TRAP is RST 4.5.)

**TABLE - 1.4 : VECTOR ADDRESSES OF INTERRUPTS**

Interrupt	Vector address	Interrupt	Vector address
RST 0	0000 <sub>H</sub>	RST 5	0028 <sub>H</sub>
RST 1	0008 <sub>H</sub>	RST 5.5	002C <sub>H</sub>
RST 2	0010 <sub>H</sub>	RST 6	0030 <sub>H</sub>
RST 3	0018 <sub>H</sub>	RST 6.5	0034 <sub>H</sub>
RST 4	0020 <sub>H</sub>	RST 7	0038 <sub>H</sub>
TRAP	0024 <sub>H</sub>	RST 7.5	003C <sub>H</sub>

The 8085 has the clock generation circuit on the chip but an external quartz crystal or LC circuit or RC circuit should be connected at the pins  $X_1$  and  $X_2$ . The frequency at  $X_1$  and  $X_2$  is divided by two internally, and are used as internal clock. The frequency of output clock signal at CLK(OUT) pin is same as that of internal clock.

The  $\overline{\text{RESET IN}}$  is the system reset input signal and it is used to bring the processor to a known state. For proper reset, the  $\overline{\text{RESET IN}}$  pin should be held **low** for at least three clock periods. When  $\overline{\text{RESET IN}}$  pin is asserted **low**, the program counter, instruction register, interrupt mask bits and all internal registers are cleared or reset. Also, the RESET OUT signal is asserted **high** to clear or reset all the peripheral devices in the system. After a reset, the content of program counter will be  $0000_{\text{H}}$  and so the processor will start executing the program stored at  $0000_{\text{H}}$ .

The pins SID and SOD can be used for serial data communication between 8085 and any serial device under software control.

### Driving $X_1$ and $X_2$ Inputs

The  $X_1$  and  $X_2$  pins of 8085 processor are provided to connect an external quartz crystal or LC circuit. It can also be driven by RC circuit or an external clock source. This connection is necessary for the internal oscillator to generate the clock signal for the processor. An oscillator consists of an amplifier and a feedback circuit. The feedback circuit of an oscillator can be RC type, LC type or quartz crystal (a quartz crystal is electrically equivalent to RLC circuit). The feedback circuit also decides the frequency of the signal generated by the oscillator.

In 8085 processor, the oscillator circuit is provided internally except the feedback circuit. This feature, facilitates the system designer to choose his own frequency for clock signals. But this frequency should not exceed the maximum clock frequency specified by the manufacturer. Another reason for keeping feedback circuit external to the processor is that the high Q circuits (quartz crystal or large values of L) cannot be fabricated by IC technology.

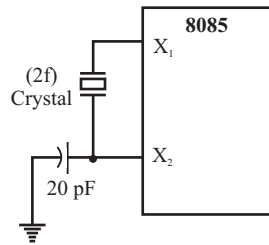
In 8085, the frequency generated by the oscillator circuit will be double than that of internal clock frequency. (The maximum clock frequencies specified by the manufacturer are internal clock frequencies.) In other words, the frequency at  $X_1 - X_2$  pins of 8085 is divided by two, internally. This means that in order to obtain an internal clock of 3.03 MHz, a clock source of 6.06 MHz must be connected to  $X_1 - X_2$ . (Crystal/LC/RC should be designed to double the internal frequency.)

The quartz crystals are the best choice for connecting at  $X_1 - X_2$ , because they are less expensive, highly stable, have large Q, occupy very small space and frequencies do not drift with ageing. For crystals with less than 4 MHz, a capacitor of 20 pF should be connected between  $X_2$  and the ground to ensure the starting up of the crystal at the right frequency.

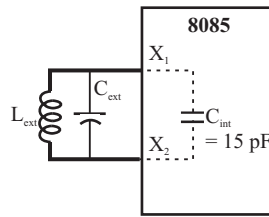
When LC circuit is used, the value of  $L_{\text{ext}}$  and  $C_{\text{ext}}$  can be chosen using the formula,

$$f = \frac{1}{2\pi L_{\text{ext}}(C_{\text{ext}} + C_{\text{int}})}$$

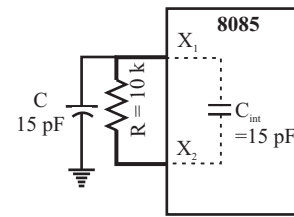
To minimize the variations in frequency, it is recommended that the value for  $C_{\text{ext}}$  should be twice than that of  $C_{\text{int}}$  or 30 pF. The use of LC circuit is not recommended for external frequencies higher than 5 MHz.



**Fig. 1.5a :** Capacitor required for crystals with frequency less than 4 Mhz.



**Fig. 1.5b :** LC tuned circuit clock driver.



**Fig. 1.5c :** RC circuit clock source.

**Fig. 1.5 :** Clock driver circuits for an 8085.

An RC circuit may also be used as the clock source for the 8085A if an accurate clock frequency is of no concern. Its advantage is its low component cost. The values shown in Fig. 1.5 are for generating an approximate external frequency of 3 MHz. Note that the frequencies higher or lower than 3 MHz should not be attempted on this circuit.

### Architecture of INTEL 8085

The architecture of 8085 is shown in Fig. 1.6. The 8085 includes the ALU, timing and control unit, instruction register and decoder, register array, interrupt control and serial IO control.

The ALU performs the arithmetic and logical operations. The operations performed by ALU of 8085 are **addition, subtraction, increment, decrement, logical AND, OR, EXCLUSIVE-OR, compare, complement and left/right shift**. The accumulator and temporary register are used to hold the data during an arithmetic/logical operation. After an operation, the result is stored in the accumulator and the flags are set or reset according to the result of the operation. The accumulator and flag register together are called **Program Status Word (PSW)**.

There are five flags in 8085, they are: **Sign Flag (SF), Zero Flag (ZF), Auxiliary Carry Flag (AF), Parity Flag (PF) and Carry Flag (CF)**. The bit positions reserved for these flags in the flag register are shown in Fig. 1.7.

After an ALU operation if the most significant bit of the result is 1, the sign flag is set. The zero flag is set if the ALU operation results in zero and it is reset if the result is nonzero. In an arithmetic operation, when a carry is generated by the lower nibble, the auxiliary carry flag is set. After an arithmetic or logical operation if the result has an even number of 1's the parity flag is set, otherwise it is reset.

If an arithmetic operation results in a carry, the carry flag is set, otherwise it is reset. Among the five flags, the AF Flag is used internally for BCD arithmetic and other four flags can be used by the programmer to check the conditions of the result of an operation.

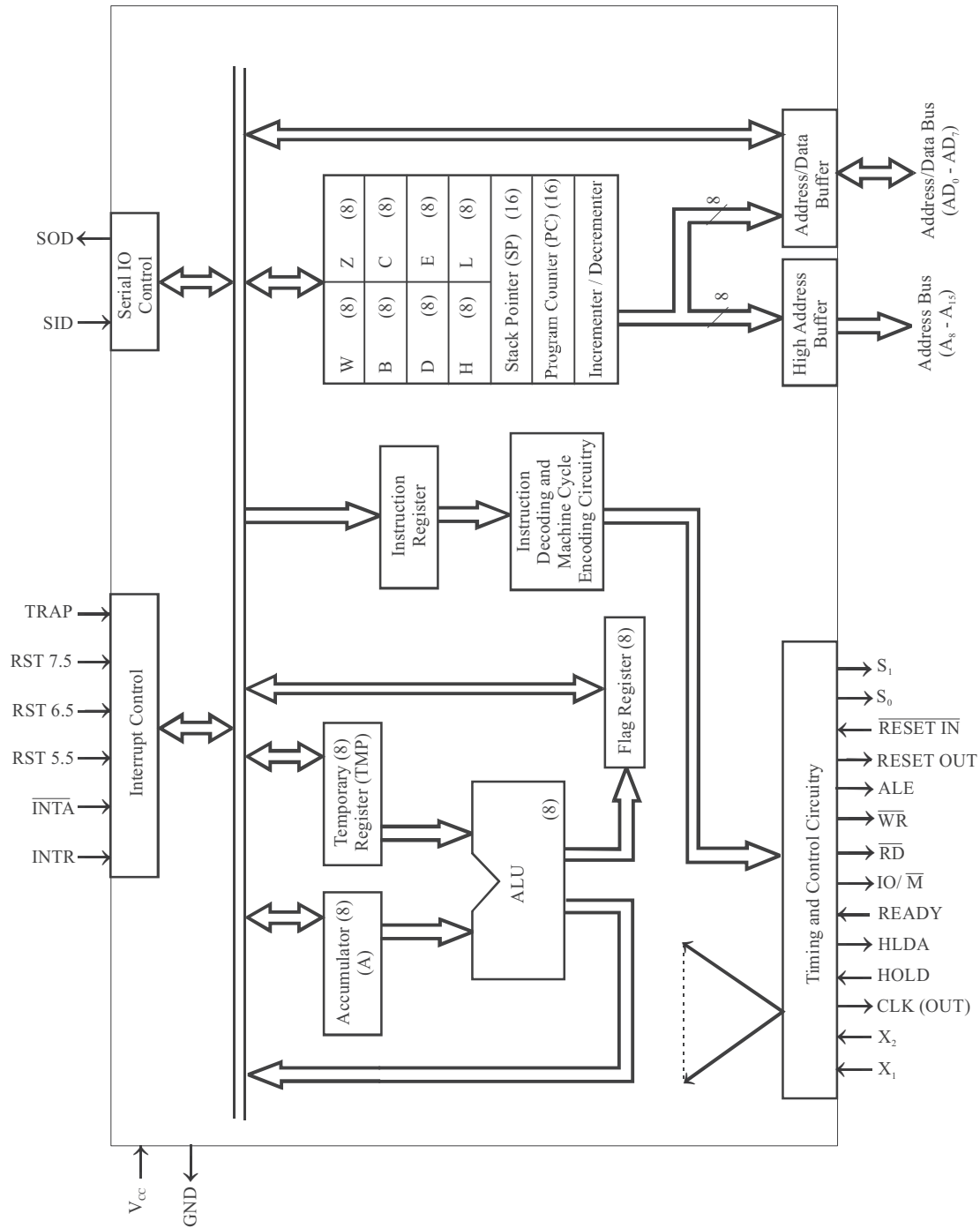


Fig. 1.6 : Architecture of INTEL 8085 microprocessor.



B <sub>7</sub>	B <sub>6</sub>	B <sub>5</sub>	B <sub>4</sub>	B <sub>3</sub>	B <sub>2</sub>	B <sub>1</sub>	B <sub>0</sub>
SF	ZF		AF		PF		CF

**Fig. 1.7 :** Bit positions of various flags in the flag register of 8085.

The timing and control unit synchronizes all the microprocessor operations with the clock, and generates the control signals necessary for communication between the microprocessor and peripherals.

When an instruction is fetched from memory, it is placed in instruction register. It is then decoded and encoded into various machine cycles. Apart from Accumulator (A-register), there are six general purpose programmable registers B, C, D, E, H and L. They can be used as 8-bit registers or paired to store 16-bit data. The allowed pairs are BC, DE and HL. The temporary registers TMP, W and Z cannot be used by the programmer.

The **Stack Pointer (SP)**, holds the address of the stack top. The stack is a sequence of RAM memory locations defined by the programmer. The stack is used to save the content of registers during the execution of a program.

The **Program Counter (PC)** keeps a track of program execution. To execute a program, the starting address of the program is loaded in program counter. The PC sends out an address to fetch a byte of instruction from memory and increment its content automatically. Hence, when a byte of instruction is fetched, the PC holds the address of the next byte of the instruction or next instruction.

### **Instruction Execution and Data Flow in 8085**

The program instructions are stored in memory which is an external device. In order to execute a program in 8085, the starting address of the program should be loaded in program counter. The 8085 output is the content of program counter to address bus and asserts read control signal **low**. Also, the program counter is incremented.

The address and the read control signal enables the memory to output the content of memory location on the data bus. Now the content of data bus is the opcode of an instruction. The read control signal is made **high** by timing and control unit after a specified time. At the rising edge of read control signals, the opcode is latched into microprocessor internal bus and placed in instruction register.

The instruction decoding unit, decodes the instruction and provides information to timing and control unit to take further actions.

## 1.8 INTEL 8086

INTEL 8086 is the first 16-bit processor released by INTEL in the year 1978. The 8086 is designed using the HMOS technology and now it is manufactured using HMOS III technology and contains approximately 29,000 transistors. The 8086 is packed in a 40-pin DIP and requires a single 5-volt supply.

The 8086 does not have internal clock circuit. The 8086 requires an external asymmetric clock source with 33% duty cycle. The 8284 clock generator is used to generate the required clock for 8086. The maximum internal clock of 8086 is 5 MHz. The other versions of 8086 with different clock rates are 8086-1, 8086-2 and 8086-4 with maximum internal clock frequency of 10 MHz, 8 MHz and 4 MHz respectively.

The 8086 uses a 20-bit address to access memory and hence it can directly address up to one mega-byte ( $2^{20} = 1\text{Mega}$ ) of memory space. The one mega-byte (1Mb) of addressable memory space of 8086 is organized as two memory banks of 512 kilo bytes each ( $512\text{ kb} + 512\text{ kb} = 1\text{Mb}$ ). The memory banks are called even (or lower) bank and odd (or upper) bank. The address line  $A_0$  is used to select even bank and the control signal  $\overline{BHE}$  is used to select odd bank.

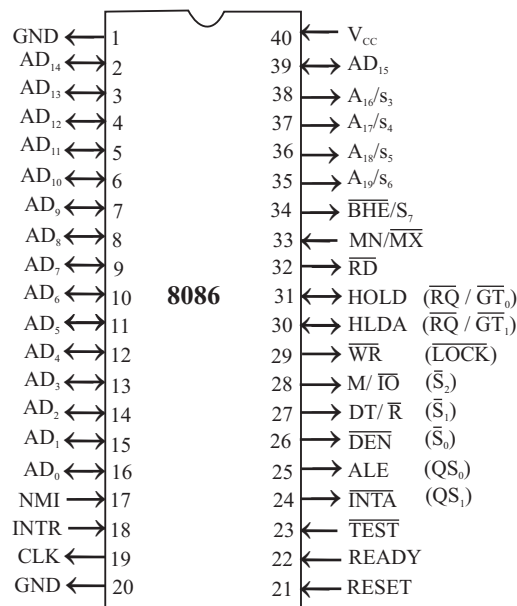
For accessing IO-mapped devices, the 8086 uses a separate 16-bit address, and so the 8086 can generate  $64\text{ k}(2^{16})$  IO addresses. The signal  $M/\overline{IO}$  is used to differentiate the memory and IO addresses. For memory address, the signal  $M/\overline{IO}$  is asserted **high** and for IO address the signal  $M/\overline{IO}$  is asserted **low** by the processor.

The 8086 can operate in two modes, and they are minimum mode and maximum mode. The mode is decided by a signal at  $MN/\overline{MX}$  pin. When the  $MN/\overline{MX}$  is tied **high**, it works in minimum mode and the system is called uniprocessor system. When  $MN/\overline{MX}$  is tied **low**, it works in maximum mode and the system is called multiprocessor system. Usually, the pin  $MN/\overline{MX}$  is permanently tied to **low** or **high** so that the 8086 system can work in any one of the two modes. The 8086 can work with 8087 coprocessor in maximum mode. In this mode, an external bus controller 8288 is required to generate bus control signals.

The 8086 has two families of processors. They are 8086 and 8088. The 8088 uses 8-bit data bus externally but 8086 uses 16-bit data bus externally. The 8086 can access memory in words but 8088 can access memory in bytes. The IBM designed its first **Personal Computer (PC)** using INTEL 8088 microprocessor as CPU.

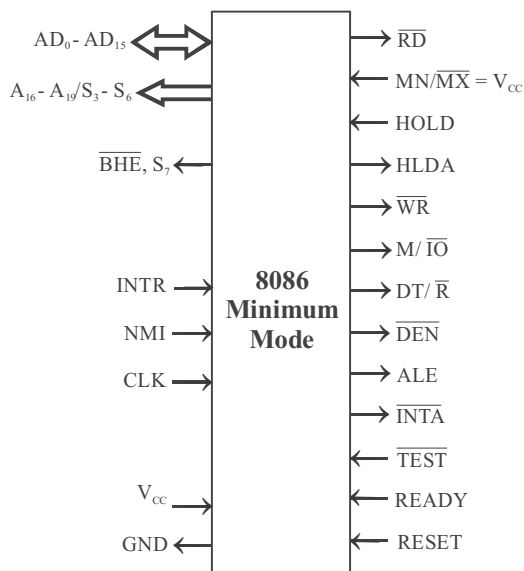
### Pins and Signals of INTEL 8086

The 8086 pins and signals are shown in Fig. 1.8. The 8086 is a 40-pin IC and all the 8086 pins are TTL compatible. The signal assigned to pins 24 to 31 will be different for minimum and maximum mode of operation. The signal assigned to all other pins are common for minimum and maximum mode of operation.

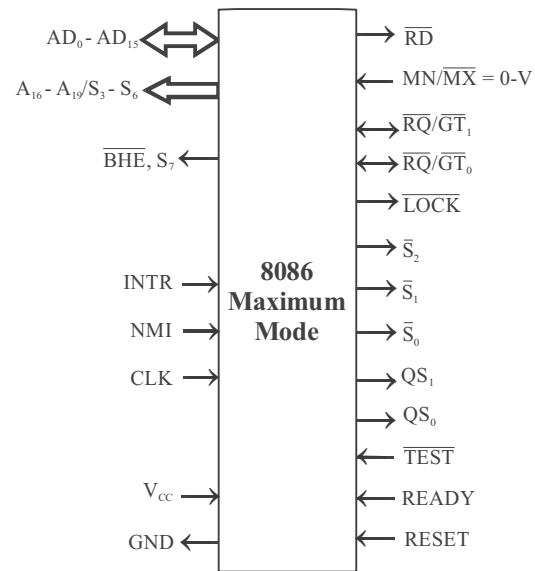


*Note : Signals shown in parenthesis are maximum mode signals.*

**Fig. a :** 8086 pin assignments.



**Fig. b :** 8086-Minimum mode.



**Fig. c :** 8086-Maximum mode.

**Fig. 1.8 :** 8086 pin and signals.

**TABLE - 1.5 : COMMON SIGNALS**

Name	Description / Function	Type
$AD_{15} - AD_0$	Address/Data	Bidirectional, Tristate
$A_{19}/S_6 - A_{16}/S_3$	Address/Status	Output, Tristate
$\overline{BHE}/S_7$	Bus high enable/Status	Output, Tristate
$MN/\overline{MX}$	Minimum/Maximum mode control	Input
$\overline{RD}$	Read control	Output, Tristate
$\overline{TEST}$	Wait on test control	Input
READY	Wait on state control	Input
RESET	System reset	Input
NMI	Nonmaskable interrupt request	Input
INTR	Interrupt request	Input
CLK	System clock	Input
$V_{cc}$	+ 5-V	Power supply input
GND	Ground	Power supply ground

**TABLE - 1.6 : MINIMUM MODE SIGNALS [ $MN/\overline{MX}=V_{cc}$  (logic high)]**

Name	Description / Function	Type
HOLD	Hold request	Input
HLDA	Hold acknowledge	Output
$\overline{WR}$	Write control	Output, Tristate
$M/\overline{IO}$	Memory/IO control	Output, Tristate
$DT/\overline{R}$	Data transmit/Receive	Output, Tristate
$\overline{DEN}$	Data enable	Output, Tristate
ALE	Address latch enable	Output
$\overline{INTA}$	Interrupt acknowledge	Output

**TABLE - 1.7 : MAXIMUM MODE SIGNALS [ $MN/\overline{MX} = \text{Ground (Logic low)}$ ]**

Name	Description / Function	Type
$\overline{RQ}/\overline{GT}_1, \overline{RQ}/\overline{GT}_0$	Request/Grant bus access control	Bidirectional
$\overline{LOCK}$	Bus priority lock control	Output, Tristate
$\overline{S}_2, \overline{S}_1, \overline{S}_0$	Bus cycle status	Output, Tristate
$QS_1, QS_0$	Instruction queue status	Output

### Common signals

The signals common for minimum and maximum mode are listed in Table-1.5. The lower sixteen lines of address are multiplexed with data and the upper four lines of address are multiplexed with status signals. During the first clock period of a bus cycle, the entire 20-bit address is available on these lines. During all other clock period of a bus cycle, the data and status signals will be available on these lines.

The status signals on  $S_3$  and  $S_4$  specifies the segment register used for calculating physical address. The output on the status lines  $S_3$  and  $S_4$  when the processor is accessing various segments are listed in Table-1.8.

**TABLE - 1.8 : STATUS SIGNAL DURING MEMORY SEGMENT ACCESS**

Status signal		Segment register
$S_4$	$S_3$	
0	0	Extra segment
0	1	Stack segment
1	0	Code or no segment
1	1	Data segment

The status lines  $S_3$  and  $S_4$  can be used to expand the memory upto 4 Mb. The status line  $S_5$  indicates the status of 8086 interrupt enable flag. A **low** on the line  $S_6$  indicates that 8086 is on the bus (i.e., it indicates that 8086 is the bus master) and during hold acknowledge this pin is driven to **high impedance** state. The output signal  $\overline{BHE}$  on the first T-state of a bus cycle is maintained as status signal  $S_7$  on the same pin.

The 8086 outputs a **low** on  $\overline{BHE}$  pin during read, write and interrupt acknowledge cycles when the data is to be transferred to the high order data bus. The  $\overline{BHE}$  can be used in conjunction with  $AD_0$  to select memory banks.

When the processor reads from memory or an IO location, it asserts  $\overline{RD}$  **low**. The  $\overline{TEST}$  input is tested by the WAIT instruction. The 8086 will enter a wait state after execution of the WAIT instruction, and it will resume execution only when  $\overline{TEST}$  is made **low** by an external hardware. This is used to synchronize an external activity to the processor internal operation.  $\overline{TEST}$  input is synchronized internally during each clock cycle on the leading edge of the clock signal.

INTR is the maskable interrupt and INTR must be held **high** until it is recognized to generate an interrupt signal. NMI is the nonmaskable interrupt input activated by a leading edge signal.

RESET is the system reset input signal. For power-ON reset, it is held **high** for 50 microsecond. For reset while working, it is held **high** for at least four clock cycles. When the processor is resetted, the DS, SS, ES, IP and flag register are cleared, Code Segment (CS) register is initialized to  $FFFF_H$  and queue is emptied. After reset, the processor will start fetching instruction from 20-bit physical address  $FFFF0_H$ .

READY is an input signal to the processor, used by the memory or IO devices to get extra time for data transfer or to introduce **wait states** in the bus cycles. Normally, READY is tied **high**. If the READY is tied **low**, the 8086 introduces wait states after second T-state of a bus cycle and it will complete the bus cycle only when READY is made **high** again.

CLK input is the clock signal that provides basic timing for the 8086 and bus controller. The 8086 does not have on-chip clock generation circuit. Hence, the 8284 clock generator chip is used to generate the required clock. A quartz crystal whose frequency is thrice that of internal clock of 8086 must be connected to 8284. The 8284 generates the clock at crystal frequency. The 8284 divides the generated clock by three and modifies the duty cycle to 33% and outputs on CLK pin of 8284. This CLK output of 8284 must be connected to the 8086 CLK pin. The 8284 also provides the RESET and READY signal to 8086.

### Minimum mode signals

The minimum mode signals of 8086 are listed in Table-1.6. For minimum mode of operation the MN/ $\overline{\text{MX}}$  pin is tied to  $V_{cc}$  (logic **high**). In minimum mode, the 8086 itself generates all bus control signals. The minimum mode signals are explained below.

- $\overline{\text{DT}}/\overline{\text{R}}$**  - [*Data Transmit/Receive*] It is an output signal from the processor to control the direction of data flow through the data transceivers.
- $\overline{\text{DEN}}$**  - (*Data Enable*) - It is an output signal from the processor used as output enable for the data transceivers.
- ALE** - (*Address Latch Enable*) - It is used to demultiplex the address and data lines using external latches.
- $\text{M}/\overline{\text{IO}}$**  - It is used to differentiate memory access and IO access. For IN and OUT instructions it is **low**. For memory reference instructions it is **high**.
- $\overline{\text{WR}}$**  - It is write control signal and it is asserted **low** whenever the processor writes data to memory or IO port.
- $\overline{\text{INTA}}$**  - (*Interrupt Acknowledge*) - The 8086 outputs **low** on this line to acknowledge when the interrupt request is accepted by the processor.
- HOLD** - It is an input signal to the processor from other bus masters as a request to grant the control of the bus. It is usually used by DMA controller to get the control of bus.
- HLDA** - (*Hold Acknowledge*) - It is an acknowledge signal by the processor to the master requesting the control of the bus through HOLD. The acknowledge is asserted **high** when the processor accepts the HOLD. [*On accepting the hold the processor drives all the tristate pins to high impedance state and sends an acknowledge to the device which requested HOLD. On receiving the acknowledge, the other master will take control of the bus.*]

### Maximum mode signals

The maximum mode signals of 8086 are listed in Table-1.7. The 8086-based system can be made to work in maximum mode by grounding the MN/ $\overline{\text{MX}}$  pin (i.e., MN/ $\overline{\text{MX}}$  is tied to logic **low**). In maximum mode, the pins 24 to 31 are redefined as follows :

$\overline{S}_0, \overline{S}_1, \overline{S}_2$  - These are status signals and they are used by the 8288 bus controller to generate bus timing and control signals. The status signals are decoded as shown in Table-1.9.

**TABLE - 1.9 : STATUS SIGNALS DURING VARIOUS MACHINE CYCLES**

Status signal			Machine cycle
$\overline{S}_2$	$\overline{S}_1$	$\overline{S}_0$	
0	0	0	Interrupt acknowledge
0	0	1	Read IO port
0	1	0	Write IO port
0	1	1	Halt
1	0	0	Code access
1	0	1	Read memory
1	1	0	Write memory
1	1	1	Passive/Inactive

$\overline{RQ}/\overline{GT}_0, \overline{RQ}/\overline{GT}_1$  - (Bus Request/Bus Grant) These requests are used by the other local bus masters to force the processor to release the local bus at the end of the processor's current bus cycle. These pins are bidirectional. The request on  $\overline{GT}_0$  will have higher priority than  $\overline{GT}_1$ .

**The bus request to 8086 work as follows :**

- When a local bus master requires system bus control, it sends a low pulse to 8086.
- At the end of current bus cycle, the processor (8086) drives its pins to **high impedance** state and sends an acknowledge as a low pulse on the same pin to the device which request the bus control.
- On receiving the acknowledge, the local master will take control of system bus. After completing its work, at the end, the local bus master sends a **low** signal on the same pin to 8086 to inform the end of control. Now 8086 can regain the control of the bus.

$\overline{LOCK}$  - It is an output signal, activated by the LOCK prefix instruction and remains active until the completion of the instruction prefixed by LOCK. The 8086 outputs **low** on the  $\overline{LOCK}$  pin while executing an instruction prefixed by LOCK to prevent other bus masters from gaining control of the system bus.

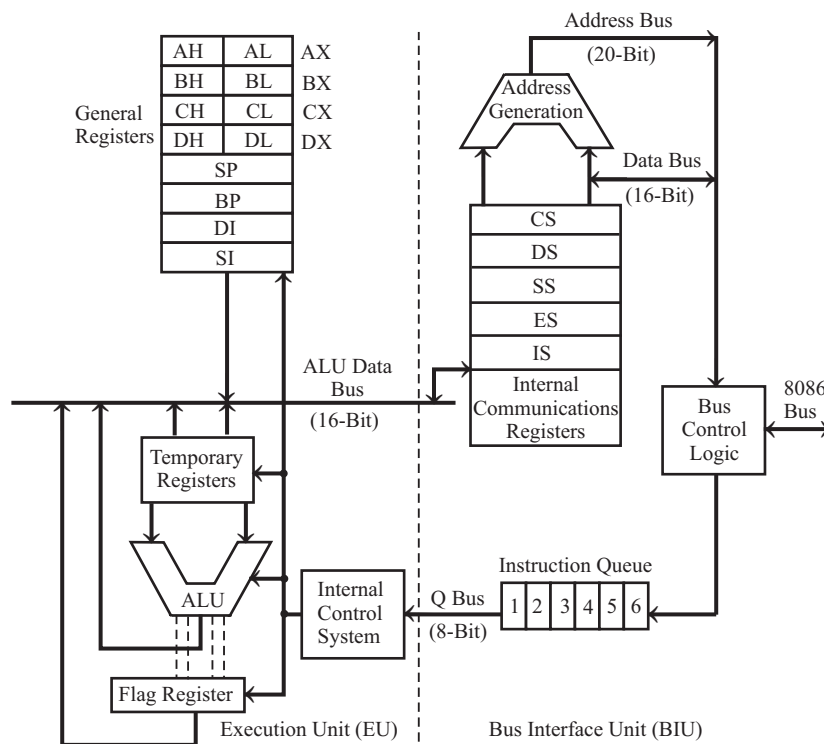
$QS_1, QS_0$  - (Queue Status) : The processor provides the status of queue on these lines. The queue status can be used by external device to track the internal status of the queue in 8086. The  $QS_0$  and  $QS_1$  are valid during the clock period following any queue operation. The output on  $QS_0$  and  $QS_1$  can be interpreted as shown in Table-1.10.

**TABLE - 1.10 : QUEUE STATUS**

Queue status		Queue operation
$QS_1$	$QS_0$	
0	0	No operation
0	1	First byte of an opcode from queue
1	0	Empty the queue
1	1	Subsequent byte from queue

### Architecture of INTEL 8086

The 8086 has pipelined architecture. In pipelined architecture, the processor will have number of functional units and the execution time of functional units are overlapped. Each functional unit works independently most of the time. The simplified block diagram of the internal architecture of 8086 is shown in Fig. 1.9. The architecture of 8086 can be internally divided into two separate functional units : **Bus Interface Unit (BIU)** and **Execution Unit (EU)**.



**Fig. 1.9 :** Internal architecture of 8086.

The BIU fetches instructions, reads data from memory and IO ports, writes data to memory and IO ports. The BIU contains segment registers, instruction pointer, instruction queue, address generation unit and bus control unit. The EU executes instructions that have already been fetched by the BIU. The BIU and EU functions independently.

The instruction queue is a FIFO (First-In-First-Out) group of registers. The size of queue is 6 bytes. The BIU fetches instruction code from memory and stores in queue. The EU fetches instruction codes from the queue.

The BIU has four numbers of 16-bit segment registers. They are **C**ode **S**egment (CS) register, **D**ata **S**egment (DS) register, **S**tack **S**egment (SS) register and **E**xtra **S**egment (ES) register. The 8086 memory space can be divided into segments of 64 kilo bytes (64kb). The 4 segment



registers are used to hold four segment base addresses. Hence, 8086 can directly address 4 segments of 64 kb at any time instant ( $4 \times 64 = 256$  kb within 1Mb memory space.) This feature of 8086 allows the system designer to allocate separate area for storing program codes and data.

The contents of segment registers are programmable. Hence, the processor can access the code and data in any part of the memory by changing the contents of segment registers. The memory segment can be continuous, partially overlapped, fully overlapped or disjointed.

***Note :** Since segment registers are programmable it is possible to design multitasking & multiuser system using 8086. The program code and data for each task or user can be stored in separate segments. The program execution can be switched from one task or user to another by changing the content of segment registers.*

The dedicated address generation unit generates 20-bit physical address from the segment base and an offset or effective address. The segment base address is logically shifted left four times and added to offset [*logically shifting left four times is equal to multiplying by  $16_{10}$ .*]

The address for fetching instruction codes is generated by logically shifting the content of CS to left four times and then adding to the content of IP (Instruction Pointer). The IP holds the offset address of the program codes. The content of IP gets incremented by two after every bus cycle. [*In one bus cycle the processor fetches two bytes of the instruction code.*]

The data address is computed by using the content of DS or ES as base address and an offset or effective address specified by the instruction. The stack address is computed by using the content of SS as base address and the content of SP (Stack Pointer) as offset address or effective address.

The bus control logic of the BIU generates all the bus control signals such as read and write signals for memory and IO. The EU consists of ALU, flag register and general purpose registers. The EU decodes and executes the instructions. A decoder in the EU control system translates instructions.

**TABLE - 1.11 : SPECIAL FUNCTIONS OF 8086 REGISTERS**

Register	Name of the register	Special function
AX	16-bit Accumulator	Stores the 16-bit result of certain arithmetic and logical operations.
AL	8-bit Accumulator	Stores the 8-bit result of certain arithmetic and logical operations.
BX	Base Register	Used to hold the base value in base addressing mode to access memory data.
CX	Count Register	Used to hold the count value in SHIFT, ROTATE and LOOP instructions.
DX	Data Register	Used to hold data for multiplication and division operations.
SP	Stack Pointer	Used to hold the offset address of top of stack memory.
BP	Base Pointer	Used to hold the base value in base addressing using stack segment register to access data from stack memory.
SI	Source Index	Used to hold the index value of source operand (data) for string instructions.
DI	Destination Index	Used to hold the index value of destination operand (data) for string instructions.

The EU has 16-bit ALU to perform arithmetic and logical operations. The EU has eight numbers of 16-bit general purpose registers. They are AX, BX, CX, DX, SP, BP, SI and DI.

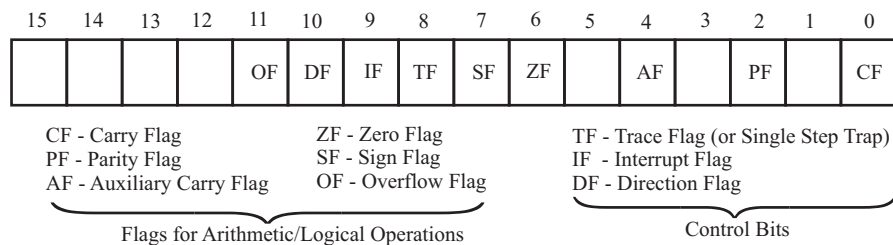
Some of the 16-bit registers can also be used as two numbers of 8-bit registers as given below:

AX - can be used as AH and AL ;      CX - can be used as CH and CL  
 BX - can be used as BH and BL ;      DX - can be used as DH and DL

The general purpose registers can be used for data storage, when they are not involved in special functions assigned to them. These registers are named after special functions carried out by each one of them as given in Table-1.11.

### 8086 flag register

The size of 8086 flag register is 16-bit and in this, nine bits are defined as flags. The six flags are used to indicate the status of the result of arithmetic or logical operations. Three flags are used to control the processor operation and so they are also called as control bits. The various flags of 8086 processor and their bit positions in flag register are shown in Fig. 1.10.



**Fig. 1.10** : Bit positions of various flags in the flag register of 8086.

**Carry Flag (CF)** is set if there is a carry from addition or borrow from subtraction. **Auxiliary carry Flag (AF)** is set if there is a carry from lower nibble to higher nibble of the low order 8-bit of a 16-bit number.

**Overflow Flag (OF)** is set to **one** if there is an arithmetic overflow, that is, if the size of the result exceeds the capacity of destination location. **Sign Flag (SF)** is set to **one** if the most significant bit of the result is **one** and SF is cleared to **zero** for non-negative result. **Parity Flag (PF)** is set to **one** if the result has even parity and PF is cleared to **zero** for odd parity of the result. **Zero Flag (ZF)** is set to **one** if the result is zero and ZF is cleared to **zero** for nonzero result.

The three control bits in the flag register can be set or reset by the programmer. The **Direction Flag (DF)** is set to **one** for autodecrement and DF is reset to **zero** for autoincrement of SI and DI registers during string data accessing. Setting **Interrupt Flag (IF)** to **one** causes the 8086 to recognize external maskable interrupts and clearing IF to **zero** disables the interrupts.

Setting **Trace Flag (TF)** to **one** places the 8086 in the single step mode. In this mode, the 8086 generates an internal interrupt after execution of each instruction. The single stepping is used for debugging a program.

### Instruction and Data Flow in 8086

The 8086 microprocessor allows the user to define different memory areas for storing program and data. The program memory can be accessed by using CS-register and the data memory can be accessed by using DS, ES and SS registers.

The program instructions are stored in program memory which is an external device. To execute a program in 8086, the base address and offset address of the first instruction of the program should be loaded in CS-register and IP, respectively. The 8086 computes the 20-bit physical address of the program instruction by multiplying the content of CS-register by  $16_{10}$  and adding to the content of IP. The 20-bit physical address is given out on the address bus. Then  $\overline{RD}$  is asserted **low**. Also other control signals necessary for program memory read operation are asserted. The IP is incremented by two to point next instruction or next word of the same instruction.

The address and control signals enable the memory to output one word (two bytes) of program memory on the data bus. After a predefined time,  $\overline{RD}$  is asserted **high** and at this instant the content of data bus is latched into two empty locations of instruction queue. Then BIU start fetching the next word of the program code as explained above. The BIU keeps on fetching the program codes, word by word from consecutive memory locations whenever two locations of queue is empty. When a branch instruction is encountered, the queue is emptied and then filled with program codes from new address loaded in CS and IP by the branch instruction.

The EU reads the program instructions from queue, decodes and executes them one by one. If the execution of an instruction require data from memory (or to store data in memory) then BIU is interrupted to read (or write) data in memory. When BIU is interrupted it completes the fetching of current instruction word and then starts reading or writing the data by generating a 20-bit data memory address. The 20-bit data memory address is obtained by multiplying the content of segment base register specified by the instruction by  $16_{10}$  and adding to an effective or offset address specified by the instruction.

### 1.9 ZILOG Z80

The ZILOG Z80 is an 8-bit microprocessor, manufactured in NMOS technology. The Z80 is available in a 40-pin DIP (**D**ual **I**n-line **P**ackage). It requires a single external clock and a single 5-V power supply. The maximum internal clock of standard Z80 is 2.5 MHz and for Z80-A it is 4 MHz. The Z80 provides more registers, extra addressing modes and a much larger instruction set than 8085. It also has built-in-logic to refresh dynamic RAM memories.

The signals of Z80 and its simplified functional block diagram (architecture) are shown in Fig. 1.11 and Fig. 1.12, respectively. The Z80 communicates with other system modules via three functionally separate buses : data, address and control buses.

The Z80 has separate pins for data and address. It operates on 8-bit data and uses 16-bit memory address. The physical memory size of Z80 system is 64 kb. The IO devices can be mapped by memory mapping or IO mapping similar to that of 8085. For IO-mapped devices, an 8-bit address is allotted. During memory refresh time, the seven lower-order bits of the address bus ( $A_0 - A_6$ ) contain a valid refresh address.

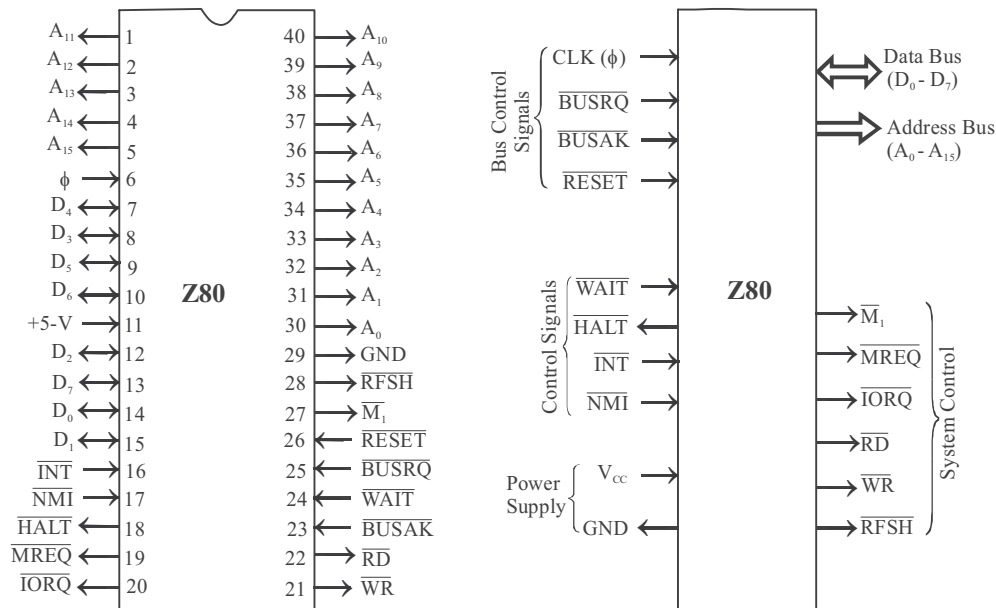


Fig. 1.11 : Signals of Z80.

The control bus has three types of control signals. They are listed below:

1. System control signals

$\overline{\text{M}}_1$	-	First machine cycle of an instruction
$\overline{\text{MREQ}}$	-	Memory request
$\overline{\text{IORQ}}$	-	IO request
$\overline{\text{RD}}$	-	Read control
$\overline{\text{WR}}$	-	Write control
$\overline{\text{RFSH}}$	-	Refresh cycle

2. CPU control signals

$\overline{\text{WAIT}}$	-	Wait request
$\overline{\text{HALT}}$	-	Halt request
$\overline{\text{INT}}$	-	Interrupt request
$\overline{\text{NMI}}$	-	Non-maskable interrupt

3. Bus control signals

$\overline{\text{BUSRQ}}$	-	BUS request
$\overline{\text{BUSAk}}$	-	BUS acknowledge
$\overline{\text{RESET}}$	-	System reset
$\overline{\text{CLK(f)}}$	-	Clock input

The ALU is 8-bit wide and performs similar functions to those of the 8085 ALU. The Z80 has two independent 8-bit accumulators, A and A' and two independent flag registers, F and F'. The ALU operation involving accumulator A affects the flag register F. The ALU operation involving accumulator A' affects the flag register F'.

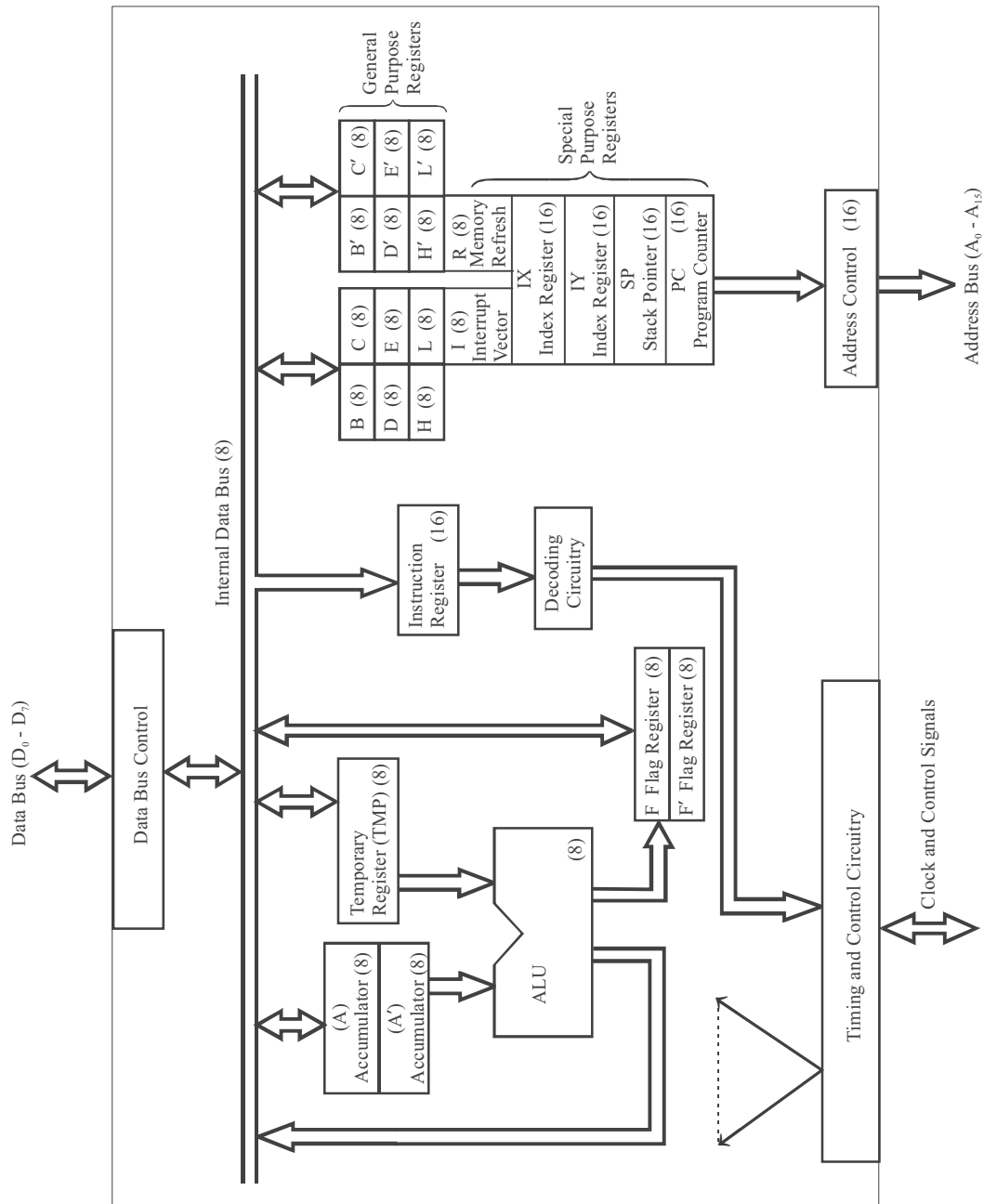


Fig. 1.12 : Architecture of Z80 microprocessor.

The flag registers has six flags and they are Sign (S and S'), Zero (Z and Z'), Carry (C and C'), Parity or Over flow (P/O and P'/O'), Half carry (H and H'), and subtract (N and N').

The Z80 has two sets of 8-bit general purpose register. Each set has 6 registers. They are B, C, D, E, H and L and B', C', D', E', H', and L'. They can be used individually as 8-bit registers or as 16-bit register pairs. The allowed pairs are BC, DE and HL and B'C', D'E' and H'L'.

At any time instant the programmer can select and work with either main register set or alternate register set. To work in alternate register set, the programmer has to use a single **Exchange Instruction (EXI)** for the entire set of instruction. This alternate set allows background mode of operation or handling fast interrupt response requirements while servicing an interrupt or executing a subroutine. While executing a program if one set of registers are not sufficient then we need not push them to stack. Alternatively, we can deactivate them without destroying its contents and switch to alternate set of registers through exchange instructions.

The 16-bit **Program Counter (PC)** and **Stack Pointer (SP)** registers are same as that of the 8085 microprocessor and operate in exactly the same way. The registers IX and IY allows two independent indexed addressing mode.

The Z80 includes an 8-bit interrupt vector (I). It is used in one of the interrupt response mode of the processor. It holds the upper eight bits of a memory pointer (or vector address). The lower eight bits of this pointer are supplied (as a vector number) by the interrupting device that requests service. The CPU then uses this 16-bit vector address to make an indirect call to the memory location that holds the first instruction of the interrupt service routine. This feature allows the vector table to be located anywhere in memory.

The Z80 also contains an 8-bit memory refresh register (R) that contains the current memory refresh address, thus providing for automatic, totally transparent refresh of external dynamic RAM memories. Although the programmer can load this register for testing purposes, the R register is not normally used by the programmer.

The Z80 can execute 158 instruction types. The microprocessor includes all the instructions of 8080A microprocessor with total software compatibility at the machine code level.

***Note :** The 8085 has the same instructions of 8080 except two new instructions, SIM and RIM. Hence, 8085 is also software compatible with Z80.*

The new instructions in Z80 include 1/4/8/16 bit operations, exchange instructions, block transfer and block search instructions and a full set of rotate and shift instructions applicable to any register, rather than just to the accumulator.

The size of Z80 instruction is 1 to 4 bytes. One byte instruction has 1-byte opcode alone. The 2-byte instruction has 1 or 2 byte opcode plus data byte / device number / displacement.

In multibyte instructions, the opcode is 1-byte or 2 bytes. The remaining bytes are data / device number / displacement / address.

The device number is 8-bit IO port address. The data byte is the immediate operand. The displacement is a signed 2's complement number which is added to a 16-bit number residing in an index register, during indexed addressing.

Every Z80 instruction consists of one to six machine cycles. All types of machine cycles consist of either three or four states. Some Z80 instructions always insert wait states ( $T_w$ ) between the states  $T_2$  and  $T_3$ . The basic operation of the Z80 is analogous to that of the INTEL 8085. The main difference is that instead of  $\overline{IO/\overline{M}}$  of 8085, the Z80 has  $\overline{MREQ}$  and  $\overline{IORQ}$ . They are activated along with  $\overline{RD}$  and  $\overline{WR}$  for the memory or IO access.

### 1.10 MOTOROLA 6800

The Motorola 6800 product family was originally introduced in 1974. The 6800 microprocessor CPU is manufactured in NMOS technology on a 40-pin chip, has TTL compatible pins and it is the first 8-bit single chip microprocessor to exploit a single 5-V power supply. The 6800 CPU can drive from seven to ten 6800 family devices without buffering. A two-phase external clock (1MHz, maximum) must be externally supplied.

The simplified functional block diagram (architecture) of the 6800 processor and its signal are shown in Figs. 1.13 and 1.14 respectively.

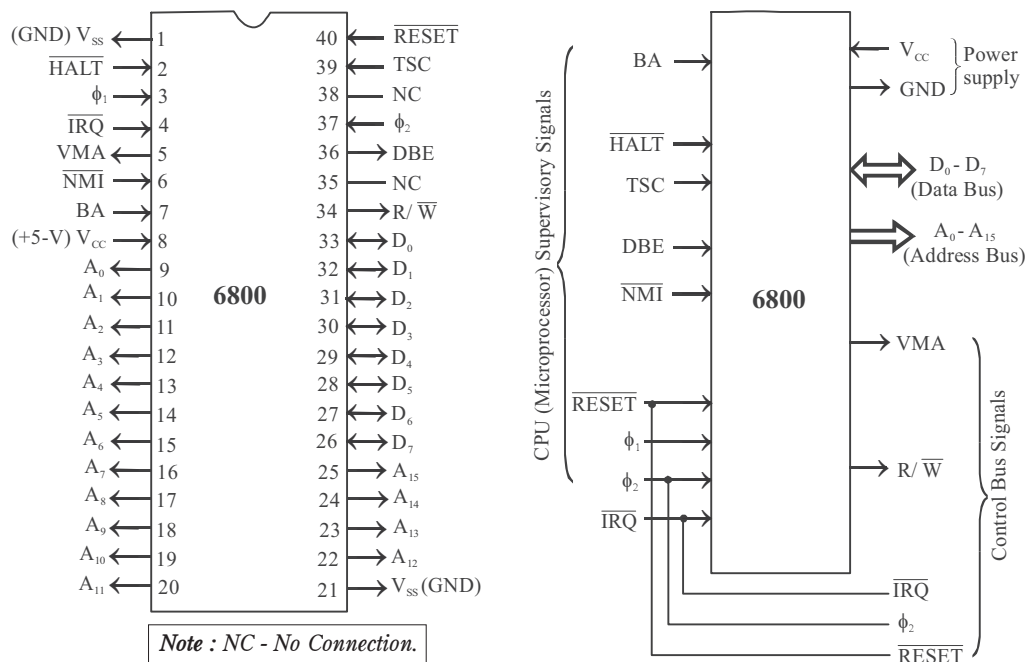


Fig. 1.13 : Signals of Motorola 6800.

The 6800 CPU has three buses to communicate with the other system modules, they are data, address and control buses. The data bus is bidirectional and has 8 lines,  $D_0-D_7$ . The address bus has 16 lines,  $A_0-A_{15}$ . The processor operates on 8-bit data and uses 16-bit address for memory and IO devices.

The microprocessor does not distinguish between memory and peripheral addresses. Therefore some of the 64 k addresses must be reserved for peripheral addresses. The control bus carries two types of signals called **Control bus signals** and **CPU (microprocessor) supervisory signals**.



Control bus signals :	VMA	-	Valid memory address
	R/W	-	Read/Write control
	IRQ	-	Interrupt request
	$\phi_2$	-	Phase-2 of clock
	RESET	-	System reset
CPU supervisory signals :	BA	-	Bus acknowledge
	HALT	-	Halt request
	TSC	-	Tristate control
	DBE	-	Data bus enable
	NMI	-	Nonmaskable interrupt
	$\phi_1$	-	Phase-1 of clock
	IRQ	-	Interrupt request
	RESET	-	System reset

The  $\overline{\text{HALT}}$  pin is used for DMA data transfer in block transfer mode or cycle stealing mode. When  $\overline{\text{HALT}}$  is asserted **low**, the microprocessor halts all its activity at the completion of the current instruction.

The **Tristate Control (TSC)** may be used to implement DMA on a cycle stealing basis. If TSC is placed in a **high** state, the address bus and the R/W line get a **high impedance** state 500 ns later. The data bus is not affected by TSC and has its own enable (DBE). This approach assures rapid response to the DMA request. Since the internal memory of the 6800 are dynamic, however, the TSC terminal cannot be held in **high** state for longer than 5  $\mu\text{s}$  if loss of data in the microprocessor is to be avoided.

The architecture of 6800 includes the ALU, 16-bit **Program Counter (PC)**, 16-bit stack pointer, 16-bit index or general purpose register, two 8-bit accumulators and a condition code register.

The stack pointer allows a LIFO (**Last-In-First-Out**) stack to be implemented at any address in memory and to be limited in size only by the memory space. The index register may be used to store data or a 16-bit memory address for use in the indexed mode of addressing. The **Condition Code Register (CCR)** indicates the results of an ALU operation. The flags in CCR are **Negative (N)**, **Zero (Z)**, **Overflow (O)**, **Carry (C)**, **Half carry (H)** and **Interrupt enable/disable (I)**. The unused bits of the CCR are 1's.

The ALU performs arithmetic and logical operations including AND, OR, EXCLUSIVE-OR, NEGATE, COMPARE, ADD, SUBTRACT and DECIMAL ADJUST which allows BCD arithmetic to be performed. Immediate, direct, indexed and relative addressing modes are used in 6800.

In the indexed addressing mode, the address contained in the second byte of the instruction is added to the lowest eight bits of the index register. The carry is then added to the higher order bits of the index register. The result is used to address memory.

In relative addressing, the address contained in the second byte of the instruction is added to the lowest eight bits of the PC. To this result, a value of +2 is added, which allows the user to address data within a range of -125 to +129 bytes of the present instruction.

The 6800 has a set of seventy two instructions. They are classified as data handling, arithmetic, logic, control transfer, data test, condition codes, address maintenance and interrupt handling.



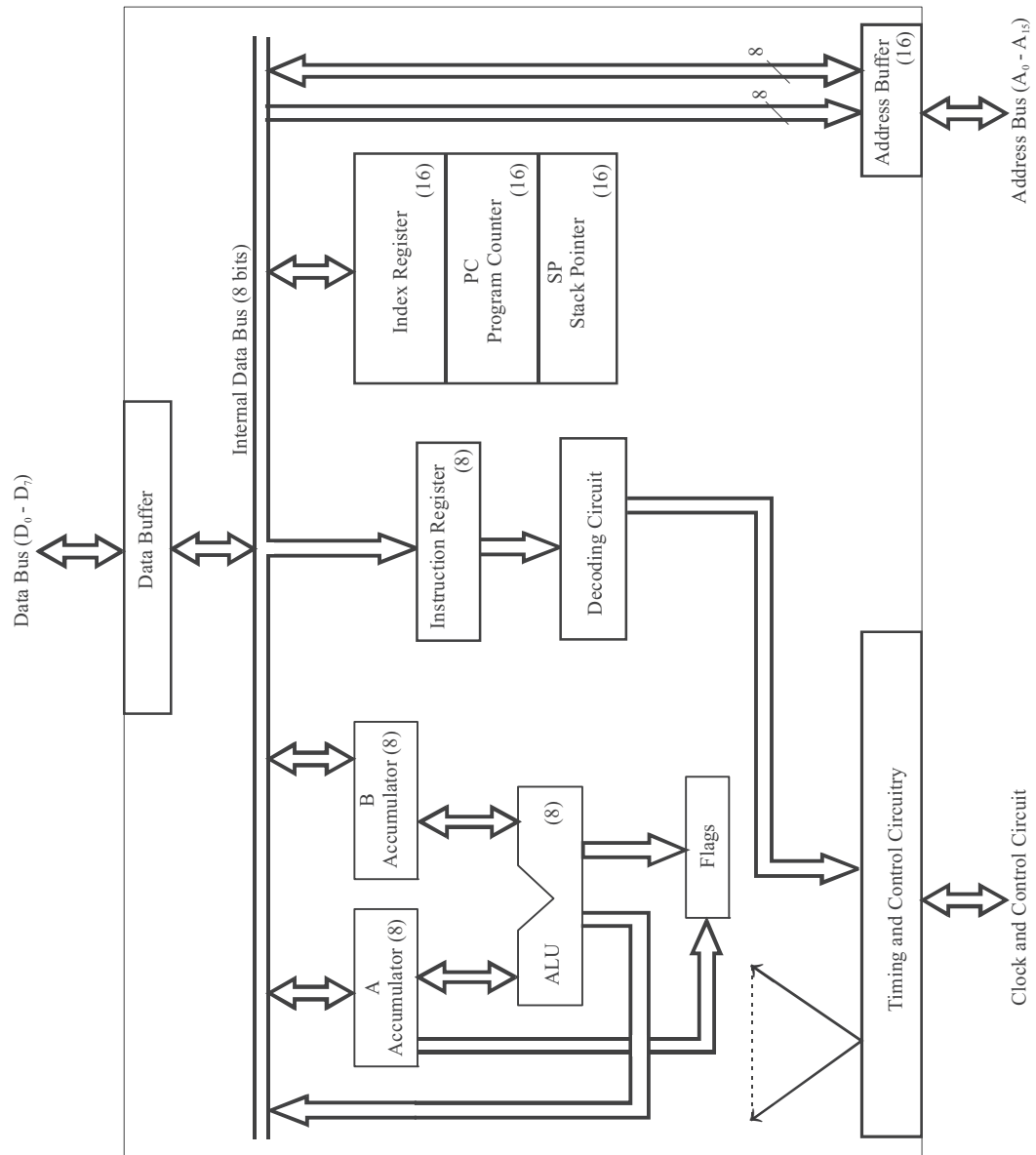


Fig. 1.14 : Architecture of motorola 6800 microprocessor.

The data handling instructions include several instructions for moving data between two accumulators, memory and the stack. Data may be altered with Clear, Increment, Decrement, Complement (1's and 2's), Rotate and Shift instructions.

The arithmetic instructions include Add, Subtract and Decimal Adjust Accumulator. The AND, OR and EXCLUSIVE-OR comprise the logical instructions.

The control transfer instructions include Unconditional Branch, Jump and Jump-to-subroutine. The Branch instruction uses relative addressing while the Jump instruction uses direct or indirect addressing. A number of conditional branches are available which test the condition of one or more bits of the condition code register.

The data test instructions set the condition codes (alter the flags) without altering the data. They include Bit Test (for comparing individual bits of accumulator A or B with a memory word), Compare and Test (for determining the sign of a number).

Condition code instructions are provided which enable the programmer to set or reset directly the Carry, Interrupt or Over flow flags. The entire contents of the condition code register may be moved to or from the accumulator A with a single instruction. Eleven instructions are provided for address maintenance. These instructions allow operations on the index register, e.g., Compare, Increment, Decrement and Transfer to or from the memory or the stack pointer. Similar instructions are available for operation on addresses stored in the stack pointer.

The interrupt handling instructions include a **Software Interrupt (SWI)** which stores the status of the processor in the stack before processing the interrupt and a **Return from Interrupt (RTI)** instruction which restores the status of the microprocessor after an interrupt is processed. A **Wait for Interrupt (WAI)** instruction causes the status to be stored in the stack and places the processor in a halt condition until a hardware interrupt occurs.

A 6800 instruction may be one, two or three bytes long, its length being closely related to the addressing mode used.

Usually, every 6800 instruction cycle consists of two to eight machine cycles, all of which are identical in length (except interrupt instructions which require longer instruction execution cycles). In the 6800, a machine cycle is one and the same thing as a clock cycle (or state).

The operation of the 6800 is very simple, since it consists of only three types of machine cycles, a read machine cycle (during which a byte of data is input into the CPU), a write machine cycle (during which a byte of data is output by the CPU) and an interrupt operation machine cycle (during which the CPU is busy and no activity occurs on system buses). The timing of any 6800 instruction is simply a concatenation of these three basic machine cycle types.

The control signals required to access a memory location are  $R/\overline{W}$ , VMA and DBE. Under normal circumstances, DBE is identical to  $\phi_2$ . The signal  $R/\overline{W}$  controls the reading or writing operation. For read operation  $R/\overline{W}$  is asserted **high** and for write operation  $R/\overline{W}$  is asserted **low**.

### 1.11 SUMMARY

- A microprocessor is a program controlled semiconductor device (IC) which fetches, decodes and executes instructions.
- The basic units or blocks of a microprocessor are ALU, an array of registers and a control unit.
- A bus is a group of conducting lines that carries data, address and control signals.
- With n-bit binary we can generate  $2^n$  different binary codes or address.
- The clock is a square wave, which is used to synchronize various devices in the microprocessor and in the system.

- The three logic levels of tristate logic are **high**, **low** and **high impedance** state.
- The world's first microprocessor INTEL 4004, was released by INTEL corporation in the year 1971.
- The NMOS process offers faster speed and higher density than PMOS and it is TTL compatible.
- The computing system designed using a microprocessor as its CPU is called a microcomputer.
- The EPROM memory is used to store permanent programs and data. RAM is used to store temporary programs and data.
- The computers are classified into Micro, Mini and Large computers.
- Large computers work at high speeds with large data words and have massive amount of memory.
- Minicomputers are used for business data processing, industrial control and scientific research.
- Microprocessor-based system offers high speed, intelligence, automation, flexibility, compactness and flexibility.
- Microprocessor-based system has limitation on the size of data, less execution speed, limited address space, does not support floating point operations.
- INTEL 8085 is an 8-bit microprocessor.
- INTEL 8085 operates on 8-bit data. It uses 16-bit address for memory and 8-bit address for IO devices.
- The physical memory address space of 8085 is 64 k ( $2^{16} = 64 \text{ k}$ ) and IO address space is 256 ( $2^8 = 256$ ).
- The maximum internal clock of 8085A is 3.03 MHz.
- The operations performed by the ALU of 8085 are addition, subtraction, increment, decrement, logical AND, OR, EXCLUSIVE-OR, compare, complement and left / right shift.
- The 8085 has five flags and they are sign, zero, auxiliary carry, parity and carry.
- The 8086 is designed using the HMOS technology and contains approximately 29000 transistors.
- INTEL 8086 is 16-bit processor, because it has a 16-bit ALU.
- The 8086 uses 20-bit address to access memory and hence it can directly address up to 1 mega-byte ( $2^{20} = 1 \text{ Mega}$ ) of memory.
- The 8086 uses 16-bit address for IO devices.
- The 8284 clock generator is used to generate clock for 8086.
- The maximum internal clock of 8086 is 5 MHz.
- The 8086 can operate in maximum (multiprocessor) mode and minimum (uniprocessor) mode.
- The 8086 has pipelined architecture. In pipelined architecture the processor will have number of functional units.
- The 8086 has two functional units namely **Bus Interface Unit (BIU)** and **Execution Unit (EU)**.
- The 8086 has 9 flags. They are carry, parity, auxiliary carry, zero, sign, overflow, trace (or single step trap), interrupt and direction flag.
- The Z80 uses 16-bit address for memory and so it can directly address 64 k ( $2^{16} = 64 \text{ k}$ ) memory locations.
- The maximum internal clock of standard Z80 is 2.5 MHz.
- The flags of Z80 are sign, zero, carry, parity/overflow, half carry and subtract flag.
- Motorola 6800 is an 8-bit processor because it has an 8-bit ALU.
- M6800 uses 16-bit address to access memory locations and so it can directly address 64 k ( $2^{16} = 64 \text{ k}$ ) memory locations.
- M6800 requires a two-phase external clock whose maximum frequency can be 1MHz.
- The flag register of M6800 is called **Condition Code Register (CCR)**.
- The flags of M6800 are negative, zero, overflow, carry, half carry and interrupt enable/disable flag.

## 1.12 SHORT QUESTIONS AND ANSWERS

---

### 1.1 What is a microprocessor ?

A microprocessor is a program controlled semiconductor device (IC), which fetches, decodes and executes instructions.

### 1.2 What are the basic functional blocks of a microprocessor ?

The basic functional blocks of a microprocessor are ALU, an array of registers and control unit.

### 1.3 What is a bus ?

Bus is a group of conducting lines that carries data, addresses and control signals.

### 1.4 Define bit, byte and word.

A digit of the binary number or code is called bit. The bit is also the fundamental storage unit of computer memory.

The 8-bit (8-digit) binary number or code is called byte and 16-bit binary number or code is called word. (Some microprocessor manufacturers refer to the basic data size operated by the processor as word.)

### 1.5 State the relation between the number of address pins and physical memory space?

The size of the binary number used to address the memory decides the physical memory space. If a microprocessor has n-address pins then it can directly address  $2^n$  memory locations. (The memory locations that are directly addressed by the processor are called physical memory space.)

### 1.6 Why is data bus is bidirectional?

The microprocessor has to fetch (read) the data from memory or input device for processing and after processing it has to store (write) the data in memory or output device. Hence, the data bus is bidirectional.

### 1.7 Why is address bus unidirectional?

The address is an identification number used by the microprocessor to identify or access a memory location or IO device. It is an output signal from the processor. Hence, the address bus is unidirectional.

### 1.8 State the difference between CPU and ALU.

The ALU is the unit that performs the arithmetic or logical operations. The CPU is the unit that includes ALU and control unit. Apart from processing the data, the CPU controls the entire system functioning. Usually, a microprocessor will be the CPU of a system and it is called the brain of the computer.

### 1.9 What is a tristate logic? Why it is needed in microprocessor system?

In a tristate logic, three logic levels are used **high**, **low** and **high impedance** state. The **high** and **low** are normal logic levels and **high impedance** state is electrical open circuit condition.

In a microprocessor system, all the peripheral/slave devices are connected to a common bus. But communication (data transfer) takes place between the master (microprocessor) and one slave (peripheral) at any time instant. During this time instant, all other devices should be isolated from the bus. Therefore, normally all the slaves (peripherals) will remain in **high impedance** state (i.e., in electrical isolation). The master will select a slave by sending address and chip select signal. When the slave is selected, it comes to normal logic and it can communicate with the master.

**1.10 What is HMOS and HCMOS.**

The HMOS is High density n-type Metal Oxide Silicon field effect transistors. The third generation microprocessors are fabricated using HMOS transistors.

The HCMOS is High density n-type Complementary Metal Oxide Silicon field effect transistors. It is the low power version of HMOS and the fourth generation microprocessors are fabricated using HCMOS transistors.

**1.11 What are the drawbacks of first generation microprocessors.**

The first generation processors are fabricated using PMOS technology and it has the drawbacks like slow speed, provides low output currents and was not compatible with TTL logic levels.

**1.12 What is a microcomputer? Explain the difference between a microprocessor and a microcomputer.**

A system designed using a microprocessor as its CPU is called microcomputer. The term microcomputer refers to the whole system, whereas the microprocessor is the CPU of the system.

**1.13 What is the function of microprocessor in a system?**

The microprocessor is the master in the system, which controls all the activity of the system. It issues address and control signals and fetches the instruction and data from memory. Then it executes the instruction to take appropriate action.

**1.14 List the components of microprocessor-based (single board microcomputer) system.**

The microprocessor-based system consist of microprocessor as CPU, semiconductor memories like EPROM and RAM, input device, output device and interfacing devices.

**1.15 Why interfacing is needed for IO devices?**

Generally IO devices are slow devices. Therefore, the speed of IO devices does not match with the speed of microprocessor. And so an interface is provided between system bus and IO devices.

**1.16 What is the difference between CPU bus and system bus?**

The CPU bus has multiplexed lines but the system bus has separate lines for each signal. (The multiplexed CPU lines are demultiplexed by the CPU interface circuit to form system bus.)

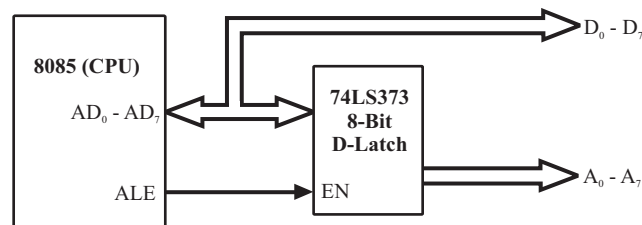
**1.17 What is multiplexing and what is its advantage?**

Multiplexing is transferring different information at different well-defined times through same lines. A group of such lines is called multiplexed bus. The advantage of multiplexing is that fewer pins are required for microprocessors to communicate with the outside world.

**1.18 How the address and data lines are demultiplexed in 8085?**

The low order address and data lines of 8085 are demultiplexed using an external 8-bit D-Latch (74LS373) and the ALE signal of 8085, as shown in Fig. Q1.18.

At the beginning of every machine cycle, ALE is asserted **high** and then **low**. Also, the low byte of address is given out through  $AD_0 - AD_7$  lines. Since the ALE is connected to enable of latch, whenever ALE is asserted **high** and then **low**, the addresses are latched into the output lines of the latch then the lines  $AD_0 - AD_7$  are free for data transfer.



**Fig. Q1.18 :** Demultiplexing of address and data lines in 8085 processor.

*1.19 On what basis the computers are classified into micro, mini and large/mainframe?*

The classification of computers into micro, mini and mainframes are based on the following factors:

1. Speed of execution
2. Size and type of data
3. Memory capacity
4. IO devices and peripheral support devices
5. Application programs it can run.

*1.20 What is the difference between micro and minicomputers?*

The microcomputers are systems built using a single microprocessor and has single motherboard as CPU. The minicomputers will have multiple microprocessors connected in a particular configuration. The minicomputer can handle large data words, address more memory space, supports a variety of IO devices and can be used for multiuser applications. In today's technology, the features of microcomputers have exceeded the capability of minicomputers.

*1.21 What is mainframe?*

The largest and most powerful computers are called mainframes.

*1.22 What is a supercomputer?*

The computer built using very high speed devices (or devices with very low switching speeds) and can execute instructions at very high speeds are called supercomputers. The speed of supercomputers are measured in MIPS (Millions of Instructions Per Second) or Megaflops (Millions of floating point operations per second). A typical supercomputer can execute 3000 MIPS. The speed of Cray X-MP2 supercomputer is 500 Megaflops.

*1.23 List the applications of microcomputer.*

- |                           |                            |
|---------------------------|----------------------------|
| 1. Personal computing     | 4. Control applications    |
| 2. Calculators            | 5. Instrumentation systems |
| 3. Small business system. |                            |

*1.24 What are the advantages of microprocessor-based system?*

The advantages of microprocessor-based system are the following:

1. Computational or Processing speed is high.
2. Intelligence has been brought to systems.
3. Automation of industrial processes and office administration.
4. Both operation and maintenance are easier.

*1.25 What are the disadvantages of microprocessor-based system?*

The following are the disadvantages of microprocessor-based system:

1. It has limitations on the size of data.
2. The applications are limited by the physical address space.
3. The analog signals cannot be processed directly and digitizing the analog signals introduces errors.
4. Most of the microprocessors do not support floating point operations.

*1.26 What do you mean by 16 and 8-bit processors? Mention a few 8-bit and 16-bit processors.*

The processors are classified into 8-bit or 16-bit depending on the basic data size handled by the ALU of the processor.

8-bit microprocessors : 8085, Z80, Motorola 6800.

16-bit microprocessors : 8086, Z8000, MC68000.

1.27 *What is the fabrication technology used for 8085?*

The 8085A is fabricated using NMOS technology and 8085AH is fabricated using HMOS technology.

1.28 **What is the physical memory space in 8085?**

The 8085 uses 16-bit address to access memory locations. Hence, it can directly address 64 k memory locations ( $2^{16} = 65,536 = 64 \text{ k}$ ). Since 8085 has 8 data lines, it can read or write 8-data bits from a memory address. Therefore, the physical memory space is  $64 \text{ k} \times 1 \text{ byte} = 64 \text{ kilo bytes (64 kb)}$ .

1.29 *What is ALE?*

The ALE (Address Latch Enable) is a signal used to demultiplex the address and data lines using an external latch. It is used as enable signal for the external latch.

1.30 *Explain the function of  $IO/\overline{M}$  in 8085.*

The  $IO/\overline{M}$  is used to differentiate memory access and IO access. For IN and OUT instruction it is asserted **high**. For memory reference instructions it is asserted **low**.

1.31 *How the READY signal is used in microprocessor system?*

The READY is an input signal that can be used by slow peripherals to get extra time in order to communicate with 8085. The 8085 will work only when READY is tied to logic **high**. Whenever READY is tied to logic **low**, the 8085 will enter a wait state. When the system has slow peripheral devices, additional hardware is provided in the system to make the READY input **low** during the required extra time while executing a machine cycle, so that the processor will remain in wait state during this extra time.

1.32 *What is HOLD and HLDA? How is it used?*

The HOLD and HLDA signals are used for the **Direct Memory Access (DMA)** type of data transfer. These type of data transfers are achieved by employing a DMA controller in the system. When DMA is required, the DMA controller will place a **high** signal on the HOLD pin of 8085. When HOLD input is asserted **high**, the processor will enter a wait state and drive all its tristate pins to **high impedance** state and send an acknowledge signal to DMA controller through HLDA pin. Upon receiving the acknowledge signal, the DMA controller will take control of the bus and perform DMA transfer and at the end it asserts HOLD signal **low**. When HOLD is asserted **low**, the processor will resume its execution.

1.33 *How clock signals are generated in 8085 and what is the frequency of the internal clock?*

The 8085 has the clock generation circuit on the chip but an external quartz crystal or LC circuit or RC circuit should be connected at the pins  $X_1$  and  $X_2$  in order to generate a clock signal. The 8085 clock generation circuit, generate a clock whose frequency is double as compared to that of internal clock. The generated clock is divided by two and then used as internal clock. The maximum internal clock frequency of 8085A is 3.03 MHz.

1.34 *What happens to the 8085 processor when it is reset?*

When  $\overline{\text{RESET IN}}$  pin is asserted **low**, the program counter, instruction register, interrupt mask bits and all internal registers are cleared or reset. Also the RESET OUT signal is asserted **high** to clear or reset all the peripheral devices in the system. After a reset, the content of program counter will be  $0000_H$  and so the processor will start executing the program stored at  $0000_H$ .

1.35 *What are the operations performed by ALU of 8085?*

The operations performed by ALU of 8085 are addition, subtraction, logical AND, OR, Exclusive-OR, compare, complement, increment, decrement and left/right shift.



1.36 Mention the names of various registers in 8085 along with its size.

Register	Size (bits)	Register	Size (bits)
Accumulator (A)	- 8	Stack pointer	- 16
Temporary register	- 8	Program counter	- 16
Instruction register	- 8		
General purpose register	- 8		
(B, C, D, E, H and L)			

1.37 What is a flag?

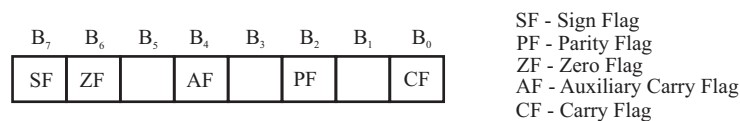
Flag is a flip- flop used to store the information about the status of the processor and the status of the instruction executed most recently.

1.38 List the flags of 8085.

There are five flags in 8085. They are sign flag, zero flag, auxiliary carry flag, parity flag and carry flag.

1.39 Show the bit positions of various flags in 8085 flag register.

The bit positions of various flags in the flag register of 8085 is shown in Fig. Q1.39.



**Fig. Q1.39** : Bit positions of various flags in the flag register of 8085.

1.40 What are the Hardware interrupts of 8085?

The hardware interrupts in 8085 are TRAP, RST 7.5, RST 6.5 and RST 5.5.

1.41 Which interrupt has highest priority in 8085? What is the priority of other interrupts?

The TRAP has the highest priority, followed by RST 7.5, RST 6.5, RST 5.5 and INTR.

1.42 Define stack.

Stack is a sequence of RAM memory locations defined by the programmer.

1.43 What is program counter? How is it useful in program execution?

The program counter keeps a track of program execution. To execute a program, the starting address of the program is loaded in program counter. The PC sends out an address to fetch a byte of instruction from memory and increment its content automatically.

1.44 How is the microprocessor synchronized with peripherals ?

The timing and control unit synchronizes all the microprocessor operations with clock and generates control signals necessary for communication between the microprocessor and peripherals.

1.45 What are the modes in which 8086 can operate?

The 8086 can operate in two modes and they are minimum (or uniprocessor) mode and maximum (or multiprocessor) mode.

1.46 What is the data and address size in 8086?

The 8086 can operate on either 8-bit or 16-bit data. The 8086 uses 20-bit address to access memory and 16-bit address to access IO devices.



1.47 What is the difference between 8086 and 8088?

The external data bus in 8086 is 16-bit and that of 8088 is 8-bit, i.e., the 8086 access memory in words but 8088 access memory in bytes.

1.48 Explain the function of  $M/\overline{IO}$  in 8086.

The signal  $M/\overline{IO}$  is used to differentiate memory address and IO address. When the processor is accessing memory locations  $M/\overline{IO}$  is asserted **high** and when it is accessing IO-mapped devices it is asserted **low**.

1.49 What are the hardware interrupts of 8086?

The hardware interrupts of 8086 are INTR and NMI. The INTR is general maskable interrupt and NMI is nonmaskable interrupt.

1.50 How is clock signal generated in 8086? What is the maximum internal clock frequency of 8086?

The 8086 does not have on-chip clock generation circuit. Hence the clock generator chip, 8284 is used to generate the required clock. The frequency of clock generated by 8284 is thrice that of internal clock frequency of 8086. The 8284 divides the generated clock by three and modifies the duty cycle to 33% and then supply as clock signal to 8086. The maximum internal clock frequency of 8086 is 5 MHz.

1.51 What is pipelined architecture?

In pipelined architecture, the processor will have the number of functional units and the execution time of functional units overlapped. Each functional unit works independently most of the time.

1.52 What are the functional units available in 8086 architecture?

The **Bus Interface Unit (BIU)** and **Execution Unit (EU)** are the two functional units available in 8086 architecture.

1.53 List the segment registers of 8086.

The segment registers of 8086 are **Code Segment (CS)**, **Data Segment (DS)**, **Stack Segment (SS)** and **Extra Segment (ES)** registers.

1.54 What is the difference between segment register and general purpose register?

The segment registers are used to store 16-bit segment base address of the four memory segments. The general purpose registers are used as the source or destination register during data transfer and computation, as pointers to memory and as counters.

1.55 What is queue? How is queue implemented in 8086?

A data structure which can be accessed on the basis of first in first out is called queue. The 8086 has six numbers of 8-bit FIFO registers, which are used as instruction queue.

1.56 Write the flags of 8086.

The 8086 has nine flags. They are:

- |                              |  |
|------------------------------|--|
| 1. Carry Flag (CF)           | 6. Overflow Flag (OF)                    |
| 2. Parity Flag (PF)          | 7. Trace Flag (TF) (or Single step trap) |
| 3. Auxiliary carry Flag (AF) | 8. Interrupt Flag (IF)                   |
| 4. Zero Flag (ZF)            | 9. Direction Flag (DF)                   |
| 5. Sign Flag (SF)            |  |

1.57 Write the special functions carried by the general purpose registers of 8086.

The special functions carried by the registers of 8086 are the following:

Register	Name of the register	Special function
AX	16-bit Accumulator	Stores the 16-bit result of certain arithmetic and logical operations.
AL	8-bit Accumulator	Stores the 8-bit result of certain arithmetic and logical operations.
BX	Base Register	Used to hold the base value in base addressing mode to access memory data.
CX	Count Register	Used to hold the count value in SHIFT, ROTATE and LOOP instructions.
DX	Data Register	Used to hold data for multiplication and division operations.
SP	Stack Pointer	Used to hold the offset address of top of stack memory.
BP	Base Pointer	Used to hold the base value in base addressing using stack segment register to access data from stack memory.
SI	Source Index	Used to hold the index value of source operand (data) for string instructions.
DI	Destination Index	Used to hold the index value of destination operand (data) for string instructions.

1.58 What are control bits?

The flags TF, IF and DF of 8086 are used to control the processor operation and so they are called control bits.

1.59 What are the additional features in Z80, when compared to 8085?

The Z80 has separate pins for data and address. The Z80 provides more register, extra addressing modes, a larger instruction set than 8085 and has a built-in logic to refresh dynamic RAM memories. The Z80 has indexed addressing mode.

1.60 What are shadow registers of Z80?

Each register of Z80 has an alternate register. The set of alternate registers are called shadow registers.

1.61 How are control signals classified in Z80?

The control signals of Z80 are classified into bus control, CPU control and system control signals.

1.62 List the register pairs of Z80.

The registers pairs of Z80 are BC, DE, HL, B'C', D'E' and H'L'.

1.63 List the flags of Z80.

The Z80 has six flags. They are as follows:

- |                          |   |
|--------------------------|---|
| 1. Sign flag (S and S')  | 4. Parity/Overflow flag (P/O and P'/O') |
| 2. Zero flag (Z and Z')  | 5. Half carry flag (H and H')           |
| 3. Carry flag (C and C') | 6. Subtract flag (N and N')             |

1.64 What are the common features of 8085 and Z80?

The common features of 8085 and Z80 are as follows:

1. It is fabricated using NMOS technology and has 40 pins.
2. Memory is accessed by 16-bit address and IO device by 8-bit address.
3. The 8085 is software compatible with Z80.

1.65 List the difference between 8085 and Z80.

The differences between 8085 and Z80 are tabulated in the following table.

8085	Z80
1. Low order address and data lines are multiplexed.	1. Separate lines are provided for address and data.
2. A single signal $\text{IO}/\overline{\text{M}}$ is used to differentiate IO access and memory access.	2. Separate signals are used to differentiate memory address and IO address.
3. The instruction size is one to three bytes.	3. The instruction size is one to four bytes.
4. The flag register has five flags.	4. The flag register has six flags.
5. It has five hardware interrupts.	5. The Z80 has two hardware interrupts.
6. It has 74 types of instruction	6. It has 156 types of instruction.

1.66 What is the data and address size in Motorola 6800?

In Motorola 6800, the data size is 8-bit and address size is 16-bit.

1.67 How IO devices are addressed in M6800?

The Motorola 6800 does not have separate address for memory and IO devices. Hence, some of the memory addresses are used to address IO devices.

1.68 How the control signals of M6800 are classified?

The control signals of M6800 are classified into control bus signals and CPU (microprocessor) supervisory signals.

1.69 What is the clock requirement of M6800?

The Motorola 6800 requires an external 2-phase clock whose maximum frequency can be 1MHz.

1.70 What is CCR? or What is the name of flag register in M6800?

The flag register in M6800 is called **Condition Code Register (CCR)**.

1.71 What are the flags of M6800?

The M6800 has six flags and they are:

1. Negative (N)
2. Zero (Z)
3. Overflow (O)
4. Carry (C)
5. Half Carry (H)
6. Interrupt enable/disable (I).

1.72 What are the addressing modes available in Motorola 6800?

The addressing modes of Motorola 6800 are immediate, direct, indexed and relative addressing.

1.73 What are the different types of instructions available in Motorola 6800?

Data handling, arithmetic, logic, control transfer, data test, condition codes, address maintenance and interrupt handling are the different types of instructions available in Motorola 6800.

1.74 List the differences between 8085 and M6800.

8085	M6800
1. Low order address and data lines are multiplexed.	1. Separate lines are provided for address and data.
2. It has 16-bit address for memory and 8-bit address for IO-mapped devices.	2. It does not have separate address for memory and IO-mapped devices.
3. The flag register has five flags.	3. The flag register has six flags.
4. It has five hardware interrupts.	4. It has two hardware interrupts.

1.75 What is the type of stack in 8085?

The stack in 8085 is LIFO stack (**L**ast-**I**n-**F**irst-**O**ut). In these type of stacks, the last stored informations can be retrieved first.

---

# INSTRUCTION SET OF 8085

## 2.1 ORIGIN OF SOFTWARE

The software is a set of instructions or commands needed for performing a specific task by a programmable device. The instructions needed for a programmable device should be written using the symbols 0 and 1. The software developed using 1s and 0s are called *machine language programs*. The machine language programs are the first programs developed for programmable devices and machines. The machine can understand only machine level programs. But it is highly tedious for a programmer to write a program in machine language.

Assembly level programs have been developed to simplify the programming task. The assembly level programs are written using mnemonics. The mnemonics are given by the manufacturers of the microprocessors. The assembly language programs are converted into machine language programs by a conversion software called assembler. Both the machine language and assembly language programs are machine dependent.

High level language programs have been developed to make the programming task independent of hardware of the machines. The examples of high level languages are BASIC, FORTRAN, COBOL, C, etc. The high level language programs are converted into machine language programs by a conversion software called Compiler or Interpreter.

## 2.2 PROCESSOR CYCLES

The sequence of operations that a processor has to carry out while executing the instruction is called instruction cycle. Each instruction cycle of a processor in turn consists of a number of machine cycles. The machine cycles are the basic operations performed by the processor. To execute an instruction, the processor executes one or more machine cycles in a particular sequence. The machine cycles of a processor are also called processor cycles. The manufacturers of microprocessors define the timings and status of various signals during the processor cycles.

In general, the instruction cycle of an instruction can be divided into two sub cycles : Fetch cycle and Execute cycle. The fetch cycle is executed to fetch the opcode from memory and the execute cycle is executed to decode the instruction and to perform the work specified by the instruction.

## 2.3 MACHINE CYCLES OF 8085

The 8085 microprocessor has seven basic machine cycles. They are as follows:

1. Opcode fetch cycle (4T or 6T)
2. Memory read cycle (3T)
3. Memory write cycle (3T)
4. IO read cycle (3T)
5. IO write cycle (3T)
6. Interrupt acknowledge cycle (6T or 12T)
7. Bus idle cycle (2T or 3T)

Each instruction of the 8085 processor consists of one to five machine cycles, i.e., when the 8085 processor executes an instruction, it will execute some of the machine cycles in a specific order. The processor takes a definite time to execute the machine cycles. The time taken by the processor to execute a machine cycle is expressed in T states. One T-state is equal to the time period of the internal clock signal of the processor. The T-state starts at the falling edge of a clock.

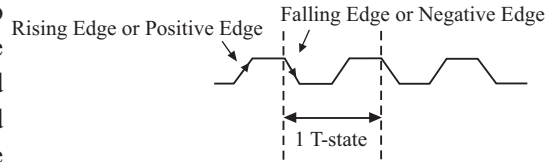


Fig. 2.1 : Clock Signal.

Note : Time period,  $T = 1/f$ ; where  $f$  = Internal clock frequency.

The T states required by the 8085 processor to execute each machine cycle are mentioned within brackets in the list of machine cycles given above.

### 2.3.1 Timing Diagram

The timing diagram provides information about the various condition (**high** state or **low** state or **high impedance** state) of the signals while a machine cycle is executed. The timing diagrams are supplied by the manufacturer of the microprocessor. The timing diagrams are essential for a system designer. Only from the knowledge of timing diagrams, the matched peripheral devices like memories, ports, etc., can be selected to form a system with microprocessor as CPU.

#### Opcode Fetch Machine Cycle of 8085

Each instruction of the processor has one-byte opcode. The opcodes are stored in memory. The opcode fetch machine cycle is executed by the processor to fetch the opcode from memory. Hence, every instruction starts with opcode fetch machine cycle.

The time taken by the processor to execute the opcode fetch cycle is either 4T or 6T. In this time, the first 3T states are used for fetching the opcode from memory and the remaining T states are used for internal operations by the processor. The timings of various signals during opcode fetch cycle are shown in Fig. 2.2.

1. At the falling edge of first T-state ( $T_1$ ), the microprocessor outputs the low byte address on  $AD_0-AD_7$  lines and high byte address on  $A_8$  to  $A_{15}$  lines. ALE is asserted **high** to enable the external address latch. The other control signals are asserted as follows.  
 $IO/\overline{M} = 0$ ,  $S_0 = 1$ ,  $S_1 = 1$ . ( $IO/\overline{M}$  is asserted **low** to indicate memory access.)
2. At the middle of  $T_1$ , the ALE is asserted **low** and this enables the external address latch to take low byte of the address and keep on its output lines.

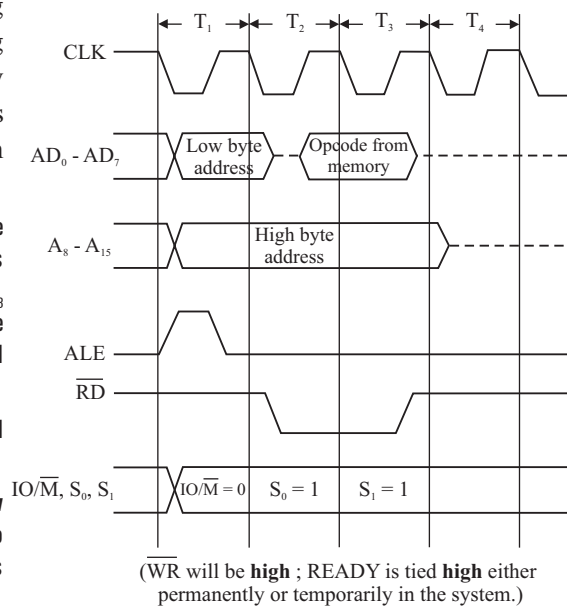


Fig. 2.2 : Opcode fetch machine cycle of 8085.

3. In the second T-state ( $T_2$ ), the memory is requested for read by asserting read line **low**. When read is asserted **low**, the memory is enabled for placing the opcode on the data bus. The time allowed for memory to output the opcode is the time during which read remains **low**.
4. In the third T-state ( $T_3$ ), the read signal is asserted **high**. On the rising edge of read signal, the opcode is latched into microprocessor. Other control signals remain in the same state until the next machine cycle.
5. The fourth T-state ( $T_4$ ) is used by the processor for internal operations to decode the instruction and encode into various machine cycles, and also for completing the task specified by 1-byte instruction. During this state ( $T_4$ ) the address and data bus will be in **high impedance** state.

### Memory Read Machine Cycle of 8085

The memory read machine cycle is executed by the processor to read a data byte from memory. The processor takes 3T states to execute this cycle. The timings of various signals during memory read cycle are shown in Fig. 2.3.

1. At the falling edge of  $T_1$ , the microprocessor outputs the low byte address on  $AD_0 - AD_7$  lines and high byte address on  $A_8$  to  $A_{15}$  lines. ALE is asserted **high** to enable the external address latch. The other control signals are asserted as follows.  
 $IO/\overline{M} = 0$ ,  $S_0 = 0$ ,  $S_1 = 1$ . ( $IO/\overline{M}$  is asserted **low** to indicate memory access.)
2. At the middle of  $T_1$ , the ALE is asserted **low** and this enables the external address latch to take low byte of address and keep on its output lines.
3. In the second T-state ( $T_2$ ), the memory is requested for read by asserting read line **low**. When read is asserted **low**, the memory is enabled for placing the data on the data bus. The time allowed for memory to output the data is the time during which read remains **low**.
4. At the end of  $T_3$ , the read signal is asserted **high**. On the rising edge of read signal, the data is latched into microprocessor. Other control signals remain in the same state until the next machine cycle.

### Memory Write Machine Cycle of 8085

The memory write machine cycle is executed by the processor to write a data byte in a memory location. The processor takes 3T states to execute this machine cycle. The timings of various signals during memory write cycle are shown in Fig. 2.4.

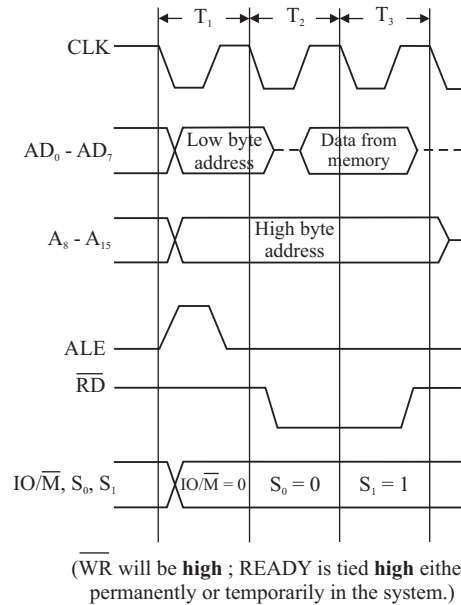


Fig. 2.3 : Memory read machine cycle of 8085.

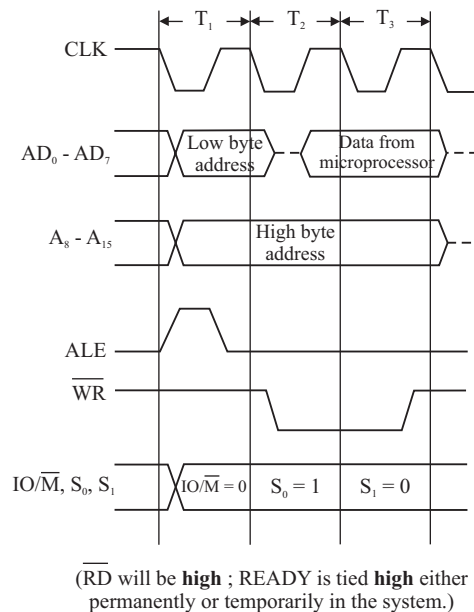


Fig. 2.4 : Memory write machine cycle of 8085.

1. At the falling edge of  $T_1$ , the microprocessor outputs the low byte address on  $AD_0 - AD_7$  lines and high byte address on  $A_8$  to  $A_{15}$  lines. ALE is asserted **high** to enable the external address latch. The other control signals are asserted as follows.  
 $IO/\overline{M} = 0$ ,  $S_0 = 1$ ,  $S_1 = 0$ . ( $IO/\overline{M}$  is asserted **low** to indicate memory access.)
2. At the middle of  $T_1$ , the ALE is asserted **low** and this enables the external address latch for latching the low byte address into its output lines.
3. In the falling edge of  $T_2$ , the processor output data on  $AD_0$  to  $AD_7$  lines and then request memory for write operation by asserting the write control signal  $\overline{WR}$  to **low**.
4. At the end of  $T_3$ , the processor asserts  $\overline{WR}$  **high**. This enables the memory to latch the data into it. The memory should prepare itself to accept the data within the time duration in which write control signal remains **low**. Other control signals remain in the same state until the next machine cycle.

### IO Read Cycle of 8085

The IO read cycle is executed by the processor to read a data byte from IO port or from the peripheral which is IO-mapped in the system. The processor takes 3T states to execute this machine cycle. The timings of various signals during this machine cycle are shown in Fig. 2.5.

1. At the falling edge of  $T_1$ , the microprocessor output the 8-bit port address on both the low order address lines ( $AD_0 - AD_7$ ) and high order address lines ( $A_8$  to  $A_{15}$ ). ALE is asserted **high** to enable the external address latch. The other control signals are asserted as follows.  
 $IO/\overline{M} = 1$ ,  $S_0 = 0$  and  $S_1 = 1$ . ( $IO/\overline{M}$  is asserted **high** to indicate IO access.)

2. At the middle of  $T_1$ , the ALE is asserted **low** and this enables the external address latch to take the port address and keep on its output lines.
3. In the second T-state ( $T_2$ ) the IO device is requested for read by asserting read line **low**. When read is asserted **low**, the IO port is enabled for placing the data on the data bus. The time allowed for IO port to output the data is the time during which read remains **low**.
4. At the end of  $T_3$ , the read signal is asserted **high**. On the rising edge of read signal the data is latched into microprocessor. Other control signals remains in the same state until the next machine cycle.

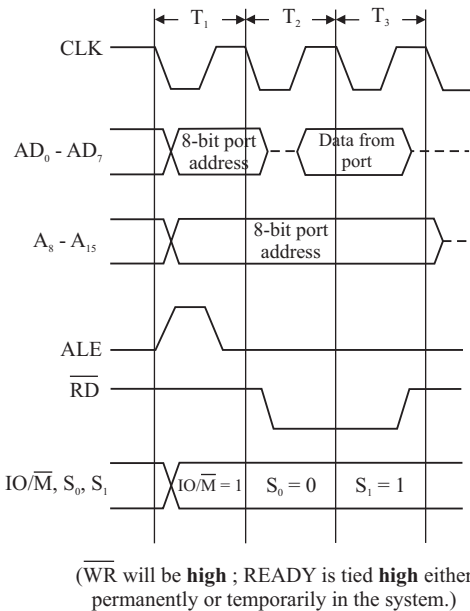


Fig. 2.5 : IO read machine cycle of 8085.

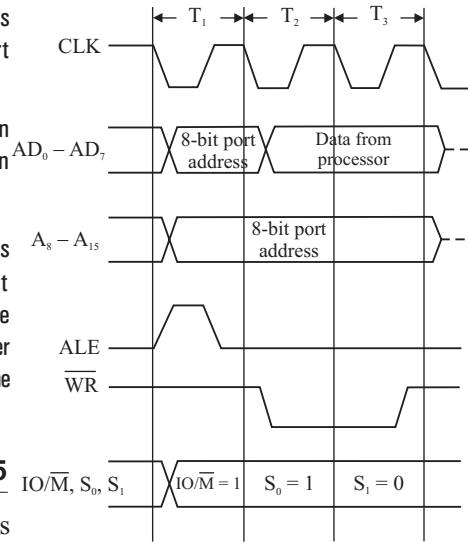
### IO Write Cycle of 8085

The IO write machine cycle is executed by the processor to write a data byte in an IO port or to a peripheral which is IO-mapped in the system. The processor takes 3T states to execute this machine cycle. The timings of the various signals of IO write cycle are shown in Fig. 2.6.

1. At the falling edge of  $T_1$ , the microprocessor outputs the 8-bit port address on low order address line ( $AD_0 - AD_7$ ) and high order address lines ( $A_8$  to  $A_{15}$ ). ALE is asserted **high** to enable the external address latch. The other control signals are asserted as follows:  
 $IO/\overline{M} = 1$ ,  $S_0 = 1$  and  $S_1 = 0$ . ( $IO/\overline{M}$  is asserted **high** to indicate IO access.)



- At the middle of  $T_1$ , the ALE is asserted **low** and this enables the external address latch for latching the port address into its output lines.
- In the falling edge of  $T_2$ , the processor output data on  $AD_0 - AD_7$  lines and then request IO port for write operation by asserting the write control signal  $\overline{WR}$  to **low**.
- At the end of  $T_3$ , the processor asserts  $\overline{WR}$  **high**. This enables the IO port to latch the data into it. The IO port should prepare itself to accept the data within the time duration in which write control signal remains **low**. Other control signals remains in the same state until the next machine cycle.



( $\overline{RD}$  will be **high** ; READY is tied **high** either permanently or temporarily in the system.)

Fig. 2.6 : IO write machine cycle of 8085.

### Interrupt Acknowledge Machine Cycle of 8085

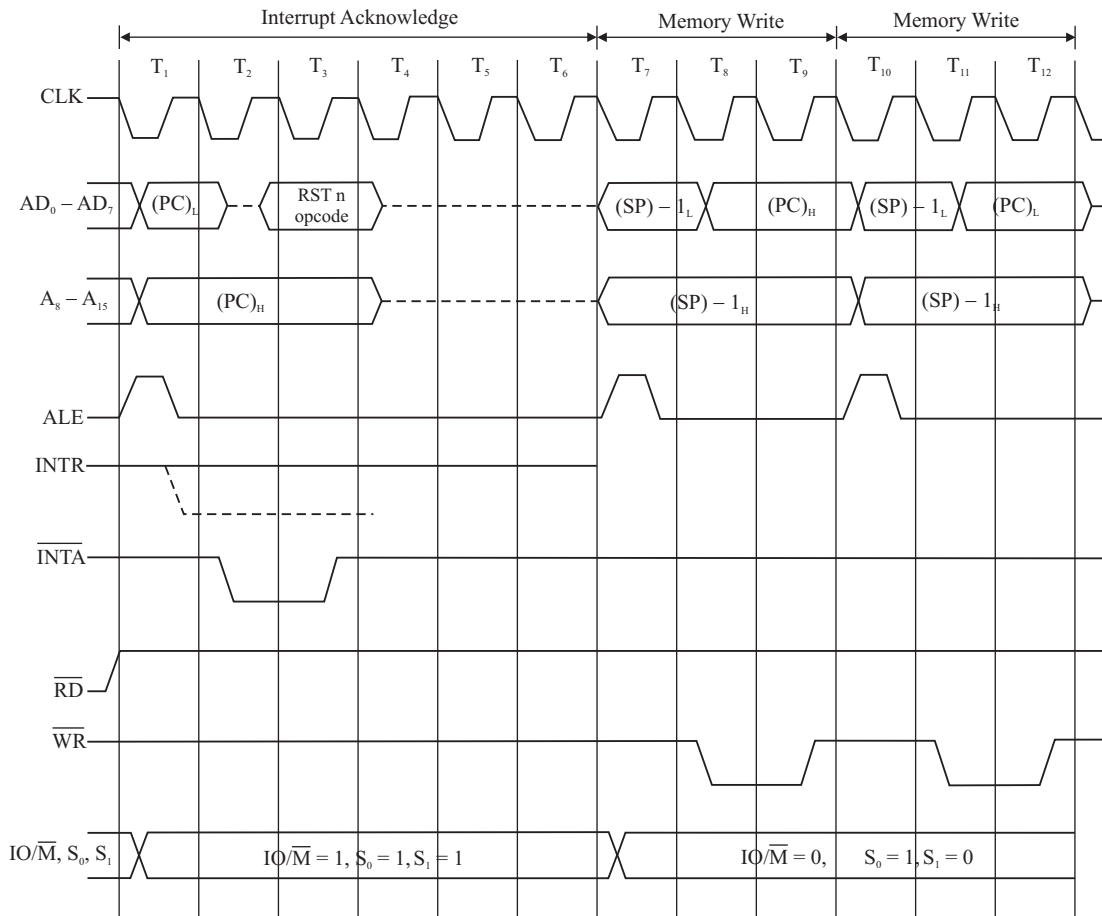
The interrupt acknowledge machine cycle is executed by the processor to service an interrupt when an interrupt request is made through INTR pin of the processor.

The 8085 processor checks for an interrupt at the second T-state of the last machine cycle of every instruction. If there is a valid interrupt request and if INTR is enabled then the processor completes the current instruction execution and then executes an interrupt acknowledge machine cycle. The interrupt acknowledge machine cycle is executed to get either a **RST n** instruction from the interrupting device or to get a CALL instruction with CALL address from the interrupting device. It also stores the content of program counter (return address) in stack.

### Interrupt acknowledge cycle of 8085 with RST n instruction

The timings of various signals during interrupt acknowledge cycle of 8085 when **RST n** instruction is supplied by the interrupting device are shown in Fig. 2.7.

- In the first T-state of interrupt acknowledge cycle, the address is placed on the  $AD_0 - AD_7$  and  $A_8 - A_{15}$  lines and ALE is asserted **high**. But the address is not used to read from memory. The other control signals are asserted as follows.  
 $IO/\overline{M} = 1$ ,  $S_0 = 1$  and  $S_1 = 1$ .  
In the middle of  $T_1$ , ALE is asserted **low**. The INTR signal can remain **high** or it can go **low** once the interrupt is accepted.
- In the second T-state ( $T_2$ ),  $\overline{INTA}$  is asserted **low**, and this enables the interrupting device to place the opcode of **RST n** instruction on the data bus.
- At the end of  $T_3$ , the  $\overline{INTA}$  is asserted **high** and the **RST n** opcode is latched into the processor. The time allowed for the external hardware to place the **RST n** opcode is the time during which  $\overline{INTA}$  remains **low**.
- The next three T states  $T_4$ ,  $T_5$  and  $T_6$  are used for internal operations. The internal operations performed are decoding the instruction and encoding into various machine cycles and generation of vector address for the **RST n** interrupt.



**Fig. 2.7 :** Interrupt acknowledge cycle with **RST n** opcode.

5. The T states T<sub>7</sub>, T<sub>8</sub> and T<sub>9</sub> are used to store the high byte of the **Program Counter (PC)** in stack (using the content of **Stack Pointer (SP)** as address).

In T<sub>7</sub>, the content of SP is decremented by one and placed on AD<sub>0</sub>-AD<sub>7</sub> and A<sub>8</sub>-A<sub>15</sub> lines. ALE is asserted **high** and then **low**, to latch the low byte of address into external latch. The status signals are asserted as  $\overline{\text{IO}/\overline{\text{M}}} = 0$ , S<sub>0</sub> = 1 and S<sub>1</sub> = 0.

In T<sub>8</sub>, the high byte of PC is placed on AD<sub>0</sub>-AD<sub>7</sub> lines and  $\overline{\text{WR}}$  is asserted **low** to enable the stack memory for write operation. At the end of T<sub>9</sub>,  $\overline{\text{WR}}$  is asserted **high**.

6. The T states T<sub>10</sub>, T<sub>11</sub> and T<sub>12</sub> are used to store the low byte of the program counter into stack.

In T<sub>10</sub>, the content of SP is again decremented by one and placed on AD<sub>0</sub>-AD<sub>7</sub> and A<sub>8</sub>-A<sub>15</sub> lines. ALE is asserted **high** and then **low**, to latch the low byte of address into external latch. The status signals are asserted as  $\overline{\text{IO}/\overline{\text{M}}} = 0$ , S<sub>0</sub> = 1 and S<sub>1</sub> = 0.

In T<sub>11</sub>, the low byte of PC is placed on AD<sub>0</sub>-AD<sub>7</sub> lines and  $\overline{\text{WR}}$  is asserted **low** to enable the stack memory for write operation. At the end of T<sub>12</sub>,  $\overline{\text{WR}}$  is asserted **high**.

After the interrupt acknowledge machine cycle, the PC will have the vector address of **RST n** instruction and so the processor starts servicing the interrupt by executing the interrupt service subroutine stored at this address.

### Interrupt acknowledge cycle of 8085 with CALL instruction

This cycle is executed by the machine to service an interrupt, when an interrupt request is made through 8259 (Interrupt Controller) to the INTR pin of 8085. The INTEL 8259 can accept 8 interrupt request and allow one by one to the INTR pin of the 8085 processor. It also supplies CALL opcode and CALL address, when it receives  $\overline{\text{INTA}}$  signal from the processor.

The processor checks for an interrupt at the second T-state of the last machine cycle of every instruction. If there is a valid interrupt request and if INTR is enabled then the processor completes the current instruction execution and then executes an interrupt acknowledge machine cycle.

The timings of various signals during interrupt acknowledge cycle when CALL instruction is supplied by the interrupting device are shown in Fig. 2.8.

1. At the falling edge of  $T_1$  the address is placed on  $\text{AD}_0 - \text{AD}_7$  and  $\text{A}_8 - \text{A}_{15}$  lines and ALE is asserted **high**. But the address is not used to read from memory. The other control signals are asserted as  $\text{IO}/\overline{\text{M}} = 1$ ,  $\text{S}_0 = 1$  and  $\text{S}_1 = 1$ .  
In the middle of  $T_1$ , ALE is asserted **low**. The INTR signal can remain **high** or it can go **low** once the interrupt is accepted by executing acknowledge cycle.
2. In  $T_2$ ,  $\overline{\text{INTA}}$  is asserted **low** and this enables the interrupt controller 8259 to place a CALL opcode on the data bus.
3. At the end of  $T_3$ , the  $\overline{\text{INTA}}$  is asserted **high** and the CALL opcode is latched into the processor.
4. The T states  $T_4$ ,  $T_5$  and  $T_6$  are used for internal operations. The internal operations performed are decoding the opcode and encoding into various machine cycles.
5. The T states  $T_7$ ,  $T_8$  and  $T_9$  are used to fetch the low byte of call address from 8259. In  $T_7$ , the content of **Program Counter (PC)** is placed on address bus but not used for memory operation. In  $T_8$  the  $\overline{\text{INTA}}$  is asserted **low** and this enables the interrupt controller 8259 to place the low byte of call address on data bus. At the end of  $T_9$  the  $\overline{\text{INTA}}$  is asserted **high** and the low byte call address on the data bus is latched into the processor.
6. The T states  $T_{10}$ ,  $T_{11}$  and  $T_{12}$  are used to fetch the high byte of call address from 8259. In  $T_{10}$  the content of PC is placed on address bus, but not used for memory operation. In  $T_{11}$  the  $\overline{\text{INTA}}$  is asserted **low** and 8259 is enabled for placing the high byte of call address on data bus. At the end of  $T_{12}$ , the  $\overline{\text{INTA}}$  is asserted **high** and the high byte call address on the data bus is latched into the processor.
7. The T states  $T_{13}$ ,  $T_{14}$  and  $T_{15}$  are used to store the high byte of the program counter in stack memory.  
In  $T_{13}$ , the content of **Stack Pointer (SP)** is decremented by one and placed on address bus. ALE is asserted **high** and then **low**, to latch the low byte of address into external latch. The other control signals are asserted as  $\text{IO}/\overline{\text{M}} = 0$ ,  $\text{S}_0 = 1$  and  $\text{S}_1 = 0$ . In  $T_{14}$ , the high byte of PC is placed on  $\text{AD}_0 - \text{AD}_7$  lines and  $\overline{\text{WR}}$  is asserted **low** to enable the stack memory for write operation. At the end of  $T_{15}$ ,  $\overline{\text{WR}}$  is asserted **high**.

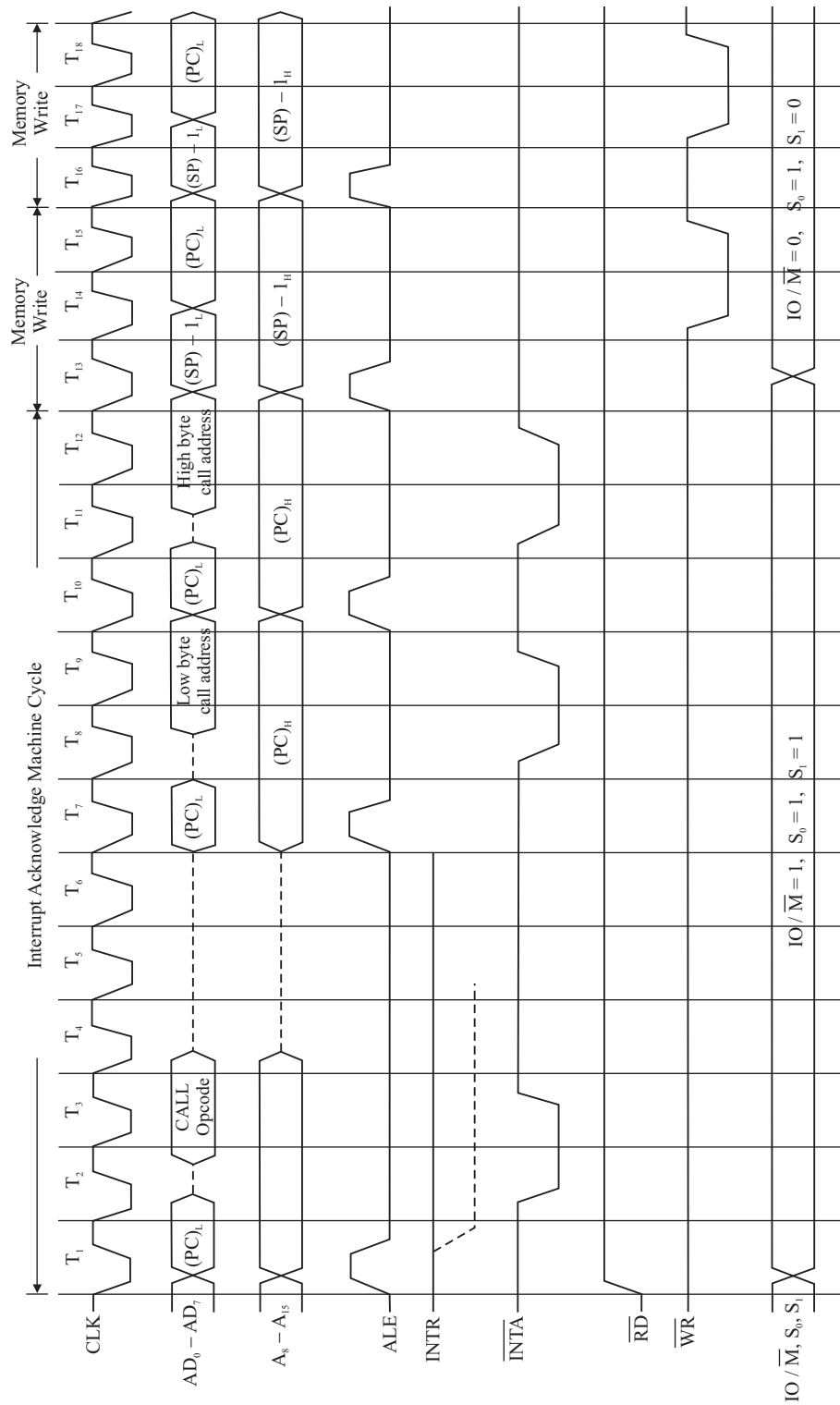


Fig. 2.8 : Interrupt acknowledge cycle with CALL opcode.

8. The T states  $T_{16}$ ,  $T_{17}$  and  $T_{18}$  are used to store the low byte of the program counter in stack memory. In  $T_{16}$ , the content of SP is again decremented by one and placed on address bus. ALE is asserted **high** and then **low**, to latch the low byte of address into external latch. The other control signals are asserted as  $\overline{IO/\overline{M}}=0$ ,  $S_0 = 1$  and  $S_1 = 0$ . In  $T_{17}$  the low byte of PC is placed on  $AD_0 - AD_7$  lines and  $\overline{WR}$  is asserted **low** to enable the stack memory for write operation. At the end of  $T_{15}$   $\overline{WR}$  is asserted **high**.

After the interrupt acknowledge machine cycle, the PC will have the call address and so the processor starts servicing the interrupt by executing the interrupt service subroutine stored at this address.

### Bus Idle Machine Cycle

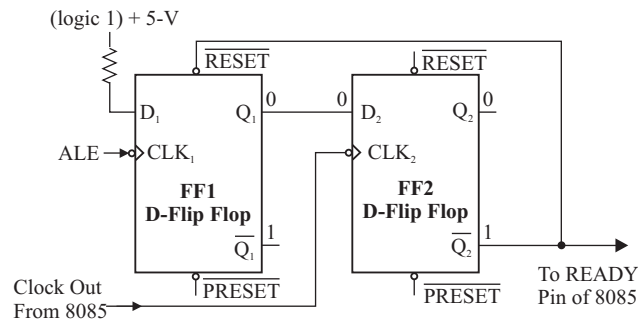
The bus idle machine cycle is executed, when extra time or more time is needed for an internal operation of the processor. During this cycle, the status signals  $S_0$  and  $S_1$  are asserted **low**. The data, address and control pins are driven to **high impedance** state. The READY signal will not be sampled by the processor during this cycle.

### Machine cycle with wait states

Wait states can be introduced in any machine cycle except bus idle cycle between  $T_2$  and  $T_3$ . The wait states are introduced in the machine cycle if READY pin is tied **low** at the second T-state of a machine cycle. The processor samples (or check) the READY signal at the second T-state of every machine cycle. If READY is tied **low** at this time, then the processor keeps on introducing wait state until the READY is again tied **high**. This facility is used by the slow memories, IO devices and peripherals to get extra time for read or write operations.

In the system when the peripheral timings are matched with processor timings, then the READY pin is permanently tied **high**. If the system peripherals require more time for read or write cycles, then using additional hardware the READY pin should be tied **low** for the required number of T states.

The circuit shown in Fig. 2.9 can be used to introduce one wait state in the machine cycles. The working of the circuit shown in Fig. 2.9 can be explained as follows:



(The values shown at the input and output of the flip-flops are initial conditions)

**Fig. 2.9** : Circuit to introduce one wait state in 8085 machine cycle.

- Initially  $Q_2 = 0$  and  $\bar{Q}_2 = 1$ . The input  $D_1$  is permanently tied **high**. The flip-flops are negative edge sensitive and so they are clocked (recognizes the clock) at the falling edges.
- In the beginning of every machine cycle (except bus idle), ALE is asserted **high** and then **low**. At the falling edge of ALE, FF1 is clocked and its output  $Q_1$  changes to 1. Also the input to FF2,  $D_2$  changes to 1.
- Now  $D_1 = 1, Q_1 = 1, D_2 = 1, Q_2 = 0, \bar{Q}_2 = 1$  and  $\overline{\text{RESET}} = 1$ .
- At the falling edge (beginning) of  $T_2$ , FF2 is clocked and so its output  $Q_2$  changes to 1 and  $\bar{Q}_2$  changes to 0.
- Now,  $D_1 = 1, Q_1 = 1, D_2 = 1, Q_2 = 1, \bar{Q}_2 = 0$  and  $\overline{\text{RESET}} = 0$ .
- Since  $\bar{Q}_2$  is connected to READY pin of 8085, the READY will be tied **low**. The  $\bar{Q}_2$  is also used to reset FF1 and so when  $\bar{Q}_2$  goes to 0 the FF1 is resetted or cleared. Now  $Q_1 = 0$  and since  $Q_1 = D_2$ , the  $D_2$  is also equal to 0.
- Now,  $D_1 = 1, Q_1 = 0, D_2 = 0, Q_2 = 1, \bar{Q}_2 = 0$  and  $\overline{\text{RESET}} = 0$ .
- At the falling edge of next T-state (i.e., in wait state) again FF2 is clocked and so the output of FF2 will change.
- Now,  $D_1 = 1, Q_1 = 0, D_2 = 0, Q_2 = 0, \bar{Q}_2 = 1$  and  $\overline{\text{RESET}} = 1$ .
- Since  $\bar{Q}_2 = 1$ , again READY is tied **high**. When the processor checks the READY at the falling edge of next cycle ( $T_3$ ), it will be **high** and it will continue the machine cycle.

Thus, the hardware shown in Fig. 2.9 introduces one wait state in the machine cycles. A machine cycle with one wait state is shown in Fig. 2.10.

Truth Table of D-flip-flop

Clock	Input	Output	
	D	$\bar{Q}$	
↓	1	1	0
↓	0	0	1

Preset and reset/clear facility in D-flip-flop

$\overline{\text{PRESET}}$	$\overline{\text{RESET}}$	Q	$\bar{Q}$
0	1	1	0
1	0	0	1
1	1	Clock and D input decide the output	
0	0	Should not occur	

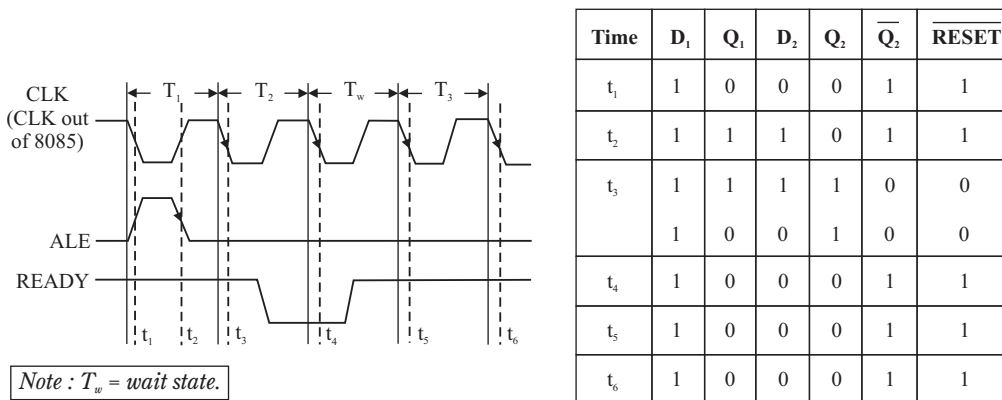


Fig. 2.10 : Machine cycle with one wait state.

## 2.4 INSTRUCTION FORMAT OF 8085

The 8085 has 74 basic instructions and 246 total instructions. The instruction set of 8085 is defined by the manufacturer INTEL Corporation. Each instruction of 8085 has one-byte opcode. With 8-bit binary code, we can generate 256 different binary codes. In this, 246 codes have been used for opcodes of 8085 instructions.

The size of 8085 instruction can be one-byte, two bytes or three bytes. The one-byte instruction has an opcode alone and the two-byte instruction has an opcode followed by an eight bit address or data. The three-byte instruction has an opcode followed by 16-bit address or data. While storing the three-byte instruction in memory, the sequence of storage is, opcode first followed by low byte of address or data and then high byte of address or data. The format of 8085 instructions are shown in Fig. 2.11.

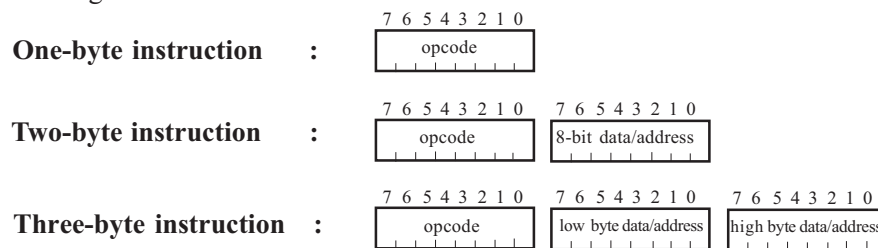


Fig. 2.11 : Format of 8085 instructions.

## 2.5 ADDRESSING MODES

Every instruction of a program has to operate on a data. The method of specifying the data to be operated by the instruction is called Addressing. The 8085 supports the following five addressing modes:

1. Immediate Addressing
2. Direct Addressing
3. Register Addressing
4. Register Indirect Addressing
5. Implied Addressing

### Immediate Addressing

*In immediate addressing mode, the data is specified in the instruction itself. The data will be a part of the program instruction.*

**Example :** MVI B, 3E<sub>H</sub>

*Move the data 3E<sub>H</sub> given in the instruction to B-register.*

### Direct Addressing

*In direct addressing mode, the address of the data is specified in the instruction. The data will be in memory. In this addressing mode, the program instructions and data can be stored in different memory blocks.*

**Example :** LDA 1050<sub>H</sub>

*Load the data available in memory location 1050<sub>H</sub> in accumulator.*

**Register Addressing**

*In register addressing mode, the instruction specifies the name of the register in which the data is available.*

**Example :** MOV A, B

*Move the content of B-register to A-register.*

**Register Indirect Addressing**

*In register indirect addressing mode, the instruction specifies the name of the register in which the address of the data is available. Here the data will be in memory and the address will be in a register pair.*

**Example :** MOV A, M

*The memory data addressed by HL pair is moved to A-register.*

**Implied Addressing**

*In implied addressing mode, the instruction itself specifies the data to be operated.*

**Example :** CMA

*Complement the content of accumulator.*

**2.6 INSTRUCTION SET**

The 8085 instructions can be classified into the following five functional groups.

- Group I      -    **Data Transfer Instructions :** Includes the instructions that moves (copies) data between registers or between memory location and register. In all data transfer operations, the content of source register or memory is not altered. Hence the data transfer is copying operation.
- Group II     -    **Arithmetic Instructions :** Includes the instructions which performs addition, subtraction, increment or decrement operations. The flag conditions are altered after execution of an instruction in this group.
- Group III    -    **Logical Instructions :** The instructions which performs the logical operations like AND, OR, EXCLUSIVE-OR, complement, compare and rotate instructions are grouped under this heading. The flag conditions are altered after execution of an instruction in this group.
- Group IV    -    **Branching Instructions :** The instructions that are used to transfer the program control from one memory location to another memory location are grouped under this heading.
- Group V     -    **Machine Control instructions :** Includes the instructions related to interrupts and the instruction used to halt program execution.

The 74 basic instructions of 8085 are listed in Table-2.1. The opcode of each instruction, size, machine cycles, number of T-state and the total number of instructions in each type are also shown in Table-2.1. The instructions affecting the status flag are listed in Table-2.2.



**TABLE - 2.1 : SUMMARY OF 8085 INSTRUCTION SET**

S.No.	Mnemonic	Opcode	No.of bytes	Machine cycles	No.of T states	Total no. of instructions
<b>Group I : Data transfer instructions</b>						
1.	MOV Rd, Rs	0 1 D D D S S S	1	F	4T	49
2.	MOV Rd, M	0 1 D D D 1 1 0	1	F, R	7T	7
3.	MOV M, Rs	0 1 1 1 0 S S S	1	F, W	7T	7
4.	MVI Rd, d8	0 0 D D D 1 1 0	2	F, R	7T	7
5.	MVI M, d8	0 0 1 1 0 1 1 0	2	F, R, W	10T	1
6.	LDA addr16	0 0 1 1 1 0 1 0	3	F, R, R, R	13T	1
7.	LDAX rp	0 0 R P 1 0 1 0	1	F, R	7T	2
8.	LXI rp, d16	0 0 R P 0 0 0 1	3	F, R, R	10T	4
9.	LHLD addr16	0 0 1 0 1 0 1 0	3	F, R, R, R, R	16T	1
10.	STA addr16	0 0 1 1 0 0 1 0	3	F, R, R, W	13T	1
11.	STAX rp	0 0 R P 0 0 1 0	1	F, W	7T	2
12.	SHLD addr16	0 0 1 0 0 0 1 0	3	F, R, R, W, W	16T	1
13.	SPHL	1 1 1 1 1 0 0 1	1	S	6T	1
14.	XCHG	1 1 1 0 1 0 1 1	1	F	4T	1
15.	XTHL	1 1 1 0 0 0 1 1	1	F, R, R, W, W	16 T	1
16.	PUSH rp	1 1 R P 0 1 0 1	1	S, W, W	12T	3
17.	PUSH PSW	1 1 1 1 0 1 0 1	1	S, W, W	12T	1
18.	POP rp	1 1 R P 0 0 0 1	1	F, R, R	10T	3
19.	POP PSW	1 1 1 1 0 0 0 1	1	F, R, R	10T	1
20.	IN addr8	1 1 0 1 1 0 1 1	2	F, R, I	10T	1
21.	OUT addr8	1 1 0 1 0 0 1 1	2	F, R, O	10T	1
<b>Group II : Arithmetic instructions</b>						
22.	ADD reg	1 0 0 0 0 S S S	1	F	4T	7
23.	ADDM	1 0 0 0 0 1 1 0	1	F, R	7T	1

Table - 2.1 continued...

S.No.	Mnemonic	Opcode	No.of bytes	Machine cycles	No.of T states	Total no. of instructions
24.	ADI d8	1 1 0 0 0 1 1 0	2	F, R	7T	1
25.	ADC reg	1 0 0 0 1 S S S	1	F	4T	7
26.	ADC M	1 0 0 0 1 1 1 0	1	F, R	7T	1
27.	ACI d8	1 1 0 0 1 1 1 0	2	F, R	7T	1
28.	DAA	0 0 1 0 0 1 1 1	1	F	4T	1
29.	DAD rp	0 0 R P 1 0 0 1	1	F, B, B	10T	4
30.	SUB reg	1 0 0 1 0 S S S	1	F	4T	7
31.	SUB M	1 0 0 1 0 1 1 0	1	F, R	7T	1
32.	SUI d8	1 1 0 1 0 1 1 0	2	F, R	7T	1
33.	SBB reg	1 0 0 1 1 S S S	1	F	4T	7
34.	SBB M	1 0 0 1 1 1 1 0	1	F, R	7T	1
35.	SBI d8	1 1 0 1 1 1 1 0	2	F, R	7T	1
36.	INR reg	0 0 S S S 1 0 0	1	F	4T	7
37.	INR M	0 0 1 1 0 1 0 0	1	F, R, W	10T	1
38.	INX rp	0 0 R P 0 0 1 1	1	S	6T	4
39.	DCR reg	0 0 S S S 1 0 1	1	F	4 T	7
40.	DCR M	0 0 1 1 0 1 0 1	1	F, R, W	10T	1
41.	DCX rp	0 0 R P 1 0 1 1	1	S	6T	4
<b>Group III : Logical instructions</b>						
42.	ANA reg	1 0 1 0 0 S S S	1	F	4T	7
43.	ANAM	1 0 1 0 0 1 1 0	1	F, R	7T	1
44.	ANI d8	1 1 1 0 0 1 1 0	2	F, R	7T	1
45.	ORA reg	1 0 1 1 0 S S S	1	F	4T	7
46.	ORAM	1 0 1 1 0 1 1 0	1	F, R	7T	1
47.	ORI d8	1 1 1 1 0 1 1 0	2	F, R	7T	1

Table - 2.1 continued...

S.No.	Mnemonic	Opcode	No.of bytes	Machine cycles	No.of T states	Total no. of instructions
48.	XRA reg	1 0 1 0 1 S S S	1	F	4T	7
49.	XRAM	1 0 1 0 1 1 1 0	1	F,R	7T	1
50.	XRI d8	1 1 1 0 1 1 1 0	2	F, R	7T	1
51.	CMP reg	1 0 1 1 1 S S S	1	F	4T	7
52.	CMP M	1 0 1 1 1 1 1 0	1	F, R	7T	1
53.	CPI d8	1 1 1 1 1 1 1 0	2	F, R	7T	1
54.	CMA	0 0 1 0 1 1 1 1	1	F	4T	1
55.	CMC	0 0 1 1 1 1 1 1	1	F	4T	1
56.	STC	0 0 1 1 0 1 1 1	1	F	4T	1
57.	RLC	0 0 0 0 0 1 1 1	1	F	4T	1
58.	RAL	0 0 0 1 0 1 1 1	1	F	4T	1
59.	RRC	0 0 0 0 1 1 1 1	1	F	4T	1
60.	RAR	0 0 0 1 1 1 1 1	1	F	4T	1
<b>Group IV : Branching instructions</b>						
61.	JMP addr16	1 1 0 0 0 0 1 1	3	F,R,R	10T	1
62.	J<condition> addr16	1 1 C C C 0 1 0	3	F,R/F,R,R	7T/10T	8
63.	CALL addr16	1 1 0 0 1 1 0 1	3	S,R,R,W,W	18T	1
64.	C<condition> addr16	1 1 C C C 1 0 0	3	S, R or S,R,R,W,W	9T/18T	8
65.	RET	1 1 0 0 1 0 0 1	1	F,R,R	10T	1
66.	R<condition>	1 1 C C C 0 0 0	1	S/S,R,R	6T/12T	8
67.	RST n	1 1 N N N 1 1 1	1	S,W,W	12T	8
68.	PCHL	1 1 1 0 1 0 0 1	1	S	6T	1

Table - 2.1 continued...

S.No.	Mnemonic	Opcode	No.of bytes	Machine cycles	No.of T states	Total no. of instructions
<b>Group V : Machine control instructions</b>						
69.	SIM	0 0 1 1 0 0 0 0	1	F	4T	1
70.	RIM	0 0 1 0 0 0 0 0	1	F	4T	1
71.	DI	1 1 1 1 0 0 1 1	1	F	4T	1
72.	EI	1 1 1 1 1 0 1 1	1	F	4T	1
73.	HLT	0 1 1 1 0 1 1 0	1	F,B	5T	1
74.	NOP	0 0 0 0 0 0 0 0	1	F	4T	1
						246

Meanings of various symbols used in Table - 2.1.

Symbol	Meaning
rp, RP	Register pair
Rs, SSS	Source register
Rd, DDD	Destination register
M	Memory
d8	8-bit data
d16	16-bit data
addr8	8-bit address
addr16	16-bit address
reg	Register
PSW	Program status word
n, NNN	Type number of restart instruction
<condition>, CCC	Flag condition
F	4T-Opcode fetch cycle
S	6T-Opcode fetch cycle
R	Memory read cycle
W	Memory write cycle
I	IO read cycle
O	IO write cycle
B	Bus idle cycle

Flag condition can be any one of the conditions given below :

Z → Zero flag = 1	M → Sign flag = 1
NZ → Zero flag = 0	P → Sign flag = 0
C → Carry flag = 1	PE → Parity flag = 1
NC → Carry flag = 0	PO → Parity flag = 0

The binary codes for the symbols used in opcode of 8085 instructions are given below :

Register	DDD or SSS
B	0 0 0
C	0 0 1
D	0 1 0
E	0 1 1
H	1 0 0
L	1 0 1
A	1 1 1

Register	RP
BC	0 0
DE	0 1
HL	1 0
SP	1 1

Flag condition	C C C
NZ	0 0 0
Z	0 0 1
NC	0 1 0
C	0 1 1
PO	1 0 0
PE	1 0 1
P	1 1 0
M	1 1 1

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

**TABLE - 2.2 : 8085 INSTRUCTIONS AFFECTING THE STATUS FLAGS**

Instructions	Status flags				
	CF	AF	ZF	SF	PF
ACI d8	+	+	+	+	+
ADC reg	+	+	+	+	+
ADC M	+	+	+	+	+
ADD reg	+	+	+	+	+
ADD M	+	+	+	+	+
ADI d8	+	+	+	+	+
ANA reg	0	1	+	+	+
ANAM	0	1	+	+	+
ANI d8	0	1	+	+	+
CMC	+				
CMP reg	+	+	+	+	+
CMP M	+	+	+	+	+
CPI d8	+	+	+	+	+
DAA	+	+	+	+	+
DAD rp	+				
DCR reg		+	+	+	+
DCR M		+	+	+	+
INR reg		+	+	+	+
INR M		+	+	+	+
ORA reg	0	0	+	+	+
ORAM	0	0	+	+	+

Table - 2.2 continued...

Instructions	Status flags				
	CF	AF	ZF	SF	PF
ORI d8	0	0	+	+	+
RAL	+				
RAR	+				
RLC	+				
RRC	+				
SBB reg	+	+	+	+	+
SBB M	+	+	+	+	+
SBI d8	+	+	+	+	+
STC	+				
SUB reg	+	+	+	+	+
SUB M	+	+	+	+	+
SUI d8	+	+	+	+	+
XRA reg	0	0	+	+	+
XRAM	0	0	+	+	+
XRI d8	0	0	+	+	+

**Note :**

- + → Indicates that the particular flag is affected.  
 0 → Indicates that the particular flag is always zero.  
 1 → Indicates that the particular flag is always one.

**TABLE - 2.3 : MEANING/EXPANSION OF MNEMONICS USED IN AN 8085 INSTRUCTION SET**

S.No.	Mnemonic	Meaning
1.	ACI	Add the immediate data and the carry to the accumulator.
2.	ADC	Add the register/memory and the carry to the accumulator.
3.	ADD	Add the register/memory to the accumulator.
4.	ADI	Add the immediate data to the accumulator.
5.	ANA	AND register/memory with the accumulator.
6.	ANI	AND immediate data with the accumulator.
7.	CALL	Call a subroutine/procedure.
8.	CC	Call on carry.
9.	CM	Call on minus.
10.	CMA	Complement accumulator.
11.	CMC	Complement carry.
12.	CMP	Compare register/memory with accumulator.
13.	CNC	Call on no carry.
14.	CNZ	Call on not zero.

Table - 2.3 continued...

S.No.	Mnemonic	Meaning
15.	CP	Call on positive.
16.	CPE	Call on parity even.
17.	CPI	Compare immediate data with the accumulator.
18.	CPO	Call on parity odd.
19.	CZ	Call on zero.
20.	DAA	Decimal adjust accumulator after addition.
21.	DAD	Double addition.
22.	DCR	Decrement the register/memory.
23.	DCX	Decrement the register pair.
24.	DI	Disable interrupt.
25.	EI	Enable interrupt.
26.	HLT	Halt program execution.
27.	IN	Input data from specified port to accumulator.
28.	INR	Increment the register/memory.
29.	INX	Increment the register pair.
30.	JC	Jump on carry.
31.	JM	Jump on minus.
32.	JMP	Jump to specified address to get the next instruction.
33.	JNC	Jump on no carry.
34.	JNZ	Jump on not zero.
35.	JP	Jump on positive.
36.	JPE	Jump on parity even.
37.	JPO	Jump on parity odd.
38.	JZ	Jump on zero.
39.	LDA	Load the accumulator.
40.	LDAX	Load accumulator indirectly using the address in the specified register pair.
41.	LHLD	Load HL direct.
42.	LXI	Load the immediate data in the register pair.
43.	MOV	Move (copy) the content of register/memory to another register/memory.
44.	MVI	Move the immediate data to register/memory.
45.	NOP	No operation.
46.	ORA	OR register/memory with accumulator.
47.	ORI	OR immediate data with accumulator.
48.	OUT	Output the content of accumulator to specified port.
49.	PCHL	Move the content of HL to PC.

Table - 2.3 continued...

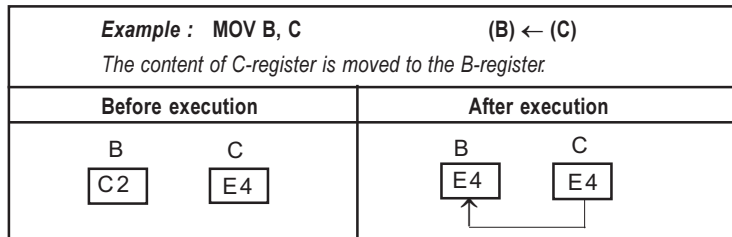
S.No.	Mnemonic	Meaning
50.	POP	Move the top of stack to the specified register pair.
51.	PUSH	Push the content of the specified register pair to top of stack.
52.	RAL	Rotate the accumulator left along with carry.
53.	RAR	Rotate the accumulator right along with carry.
54.	RC	Return on carry.
55.	RET	Return from subroutine/procedure to calling program.
56.	RIM	Read interrupt mask status.
57.	RLC	Rotate accumulator left to carry.
58.	RM	Return on minus.
59.	RNC	Return on no carry.
60.	RNZ	Return on not zero.
61.	RP	Return on positive.
62.	RPE	Return on parity even.
63.	RPO	Return on parity odd.
64.	RRC	Rotate accumulator right to carry.
65.	RST	Restart the program execution from the specified vector address.
66.	RZ	Return on zero.
67.	SBB	Subtract register/memory and the carry (borrow) from accumulator.
68.	SBI	Subtract the immediate data and the carry (borrow) from accumulator.
69.	SHLD	Store HL direct.
70.	SIM	Set interrupt mask.
71.	SPHL	Move HL to SP.
72.	STA	Store accumulator.
73.	STAX	Store accumulator indirectly by using the address in specified register pair.
74.	STC	Set carry.
75.	SUB	Subtract register/memory from accumulator.
76.	SUI	Subtract the immediate data from accumulator.
77.	XCHG	Exchange DE and HL.
78.	XRA	Exclusive-OR register/memory with accumulator.
79.	XRI	Exclusive-OR the immediate data with accumulator.
80.	XTHL	Exchange the top of stack and HL.



## 2.7 DATA TRANSFER INSTRUCTIONS

### 1. MOV Rd, Rs (Rd) ← (Rs)

The content of source register (Rs) is copied to the destination register (Rd). The registers Rd and Rs can be any one of the general purpose registers A, B, C, D, E, H or L. No flags are affected.



One byte instruction

One machine cycle : Opcode fetch - 4T

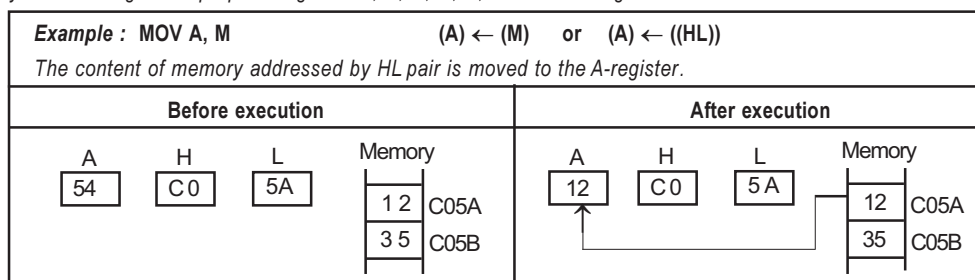
Register addressing

Total number of instructions = 49

MOV A, A	MOV B, A	MOV D, A	MOV H, A
MOV A, B	MOV B, B	MOV D, B	MOV H, B
MOV A, C	MOV B, C	MOV D, C	MOV H, C
MOV A, D	MOV B, D	MOV D, D	MOV H, D
MOV A, E	MOV B, E	MOV D, E	MOV H, E
MOV A, H	MOV B, H	MOV D, H	MOV H, H
MOV A, L	MOV B, L	MOV D, L	MOV H, L
	MOV C, A	MOV E, A	MOV L, A
	MOV C, B	MOV E, B	MOV L, B
	MOV C, C	MOV E, C	MOV L, C
	MOV C, D	MOV E, D	MOV L, D
	MOV C, E	MOV E, E	MOV L, E
	MOV C, H	MOV E, H	MOV L, H
	MOV C, L	MOV E, L	MOV L, L

### 2. MOV Rd, M (Rd) ← (M)    or    (Rd) ← ((HL))

The content of memory (M) addressed by the HL pair is moved to the destination register (Rd). The register Rd can be any one of the general purpose registers A, B, C, D, E, H or L. No flags are affected.



One byte instruction

Two machine cycles: Opcode fetch - 4T

Register indirect addressing

Memory read -  $\frac{3T}{7T}$

Total number of instructions = 7

MOV A, M    MOV B, M    MOV C, M    MOV D, M    MOV E, M    MOV H, M    MOV L, M

### 3. MOV M, Rs $(M) \leftarrow (Rs)$ or $((HL)) \leftarrow (Rs)$

The content of source register (Rs) is moved to the memory location addressed by HL pair. The register Rs can be any one of the general purpose registers A, B, C, D, E, H or L. No flags are affected.

<b>Example : MOV M, B</b> $(M) \leftarrow (B)$ or $((HL)) \leftarrow (B)$ The content of B-register is moved to memory location addressed by the HL pair.			
Before execution		After execution	
B 74	H C2 L 50	B 74 H C2 L 50	Memory 02 C250 15 C251 74 C250 15 C251

One byte instruction

Register indirect addressing

Two machine cycles : Opcode fetch - 4T

Memory write - 3T

7T

Total number of instructions = 7

MOV M, A    MOV M, B    MOV M, C    MOV M, D    MOV M, E    MOV M, H    MOV M, L

### 4. MVI Rd, d8 $(Rd) \leftarrow d8$

The 8-bit data (d8) given in the instruction is moved to the destination register (Rd). The register Rd can be any one of the general purpose registers A, B, C, D, E, H or L. No flags are affected.

<b>Example : MVI D, 09H</b> $(D) \leftarrow 09_H$ The 8-bit data 09 <sub>H</sub> given in the instruction is moved to the D-register.	
Before execution	After execution
D C2	D 09

Two byte instruction

Immediate addressing

Two machine cycles: Opcode fetch - 4T

Memory read - 3T

7T

Total number of instructions = 7

MVI A, d8    MVI B, d8    MVI C, d8    MVI D, d8    MVI E, d8    MVI H, d8    MVI L, d8

### 5. MVI M, d8 $(M) \leftarrow d8$ or $((HL)) \leftarrow d8$

The 8-bit data (d8) given in the instruction is moved to the memory location addressed by the HL pair. No flags are affected.

<b>Example : MVI M, E7H</b> $(M) \leftarrow E7_H$ or $((HL)) \leftarrow E7_H$ The 8-bit data E7 <sub>H</sub> given in the instruction is moved to the memory location addressed by the HL pair.			
Before execution		After execution	
H 20	L 5C	H 20 L 5C	Memory 28 205C 3A 205D E7 205C 3A 205D

Two byte instruction  
Register indirect addressing or  
Immediate addressing

Three machine cycles : Opcode fetch - 4T  
Memory read - 3T  
Memory write - 3T  
10T

Total number of instructions = 1

# 6. LDA addr16 (A) ← (M) or (A) ← (addr16)

The content of the memory location whose address is given in the instruction, is moved to accumulator. No flags are affected.

Before execution		After execution	
<div> <div>A</div> <div>C 2</div> </div>	Memory	<div> <div>A</div> <div>15</div> </div>	Memory
	<div> <div>15</div> <div>205D</div> </div> <div> <div>7 F</div> <div>205E</div> </div>		<div> <div>15</div> <div>205D</div> </div> <div> <div>7 F</div> <div>205E</div> </div>

Three byte instruction  
Direct addressing

Four machine cycles : Opcode fetch - 4T  
Memory read - 3T  
Memory read - 3T  
Memory read - 3T  
13T

Total number of instructions = 1

# 7. LHLD addr16 (L) ← (M) (L) ← (addr16) (H) ← (M) or (H) ← (addr16 + 01)

The content of the memory location whose address is given in the instruction, is moved to the L-register. The content of the next memory location is moved to the H-register. No flags are affected.

Before execution		After execution	
<div> <div>H</div> <div>05</div> </div> <div> <div>L</div> <div>72</div> </div>	Memory	<div> <div>H</div> <div>3 D</div> </div> <div> <div>L</div> <div>6 A</div> </div>	Memory
	<div> <div>6A</div> <div>1050</div> </div> <div> <div>3 D</div> <div>1051</div> </div> <div> <div>0 9</div> <div>1052</div> </div>		<div> <div>6A</div> <div>1050</div> </div> <div> <div>3 D</div> <div>1051</div> </div> <div> <div>0 9</div> <div>1052</div> </div>

Three byte instruction  
Direct addressing

Five machine cycles: Opcode fetch - 4T  
Memory read - 3T  
Memory read - 3T  
Memory read - 3T  
Memory read - 3T  
16T

Total number of instructions = 1

**8.**     **LXI rp, d16**                         (rp) ← d16

The 16-bit data given in the instruction is moved to the register pair (rp). The register pair can be BC, DE, HL or SP.

<b>Example :</b>	LXI H, 1050H	(L) ← 50 <sub>H</sub> (H) ← 10 <sub>H</sub>
<i>The 16-bit data 1050<sub>H</sub> given in the instruction is moved to the HL register pair.</i>		
Before execution		After execution
<div style="display: flex; justify-content: space-around;"> <div style="text-align: center;">H <div style="border: 1px solid black; padding: 2px 10px;">xx</div></div> <div style="text-align: center;">L <div style="border: 1px solid black; padding: 2px 10px;">yy</div></div> </div> <p>(some arbitrary value)</p>	<div style="display: flex; justify-content: space-around;"> <div style="text-align: center;">H <div style="border: 1px solid black; padding: 2px 10px;">10</div></div> <div style="text-align: center;">L <div style="border: 1px solid black; padding: 2px 10px;">50</div></div> </div>	

### Three byte instruction

### Immediate addressing

**Three machine cycles : Opcode fetch - 4T**

Memory read -  $3T$

Memory read -  $3T$

$$\overline{10T}$$

Total number of instructions = 4

LXI B, d16                  LXI D, d16                  LXI H, d16                  LXI SP, d16

9. LDAX rp (A) ← (M) or (A) ← ((rp))

The content of the memory addressed by the register pair (rp) is moved to the accumulator. (The content of the register pair is the memory address). The register pair can be either BC or DE.

<p><b>Example :</b> LDAX B                      (A) ← (M)    or    (A) ← ((BC))</p> <p>The content of the memory location addressed by the BC pair is moved to the A-register.</p>									
<p><b>Before execution</b></p> <div style="display: flex; align-items: center; justify-content: space-around;"> <div style="text-align: center;"> A  <div style="border: 1px solid black; padding: 2px;">02</div> </div> <div style="text-align: center;"> B  <div style="border: 1px solid black; padding: 2px;">20</div> </div> <div style="text-align: center;"> C  <div style="border: 1px solid black; padding: 2px;">5A</div> </div> <div style="text-align: center;"> Memory  <table border="1" style="border-collapse: collapse;"> <tr><td style="padding: 2px;">1E</td><td style="padding: 2px;">205A</td></tr> <tr><td style="padding: 2px;">3C</td><td style="padding: 2px;">205B</td></tr> </table> </div> </div>	1E	205A	3C	205B	<p><b>After execution</b></p> <div style="display: flex; align-items: center; justify-content: space-around;"> <div style="text-align: center;"> A  <div style="border: 1px solid black; padding: 2px;">1E</div> </div> <div style="text-align: center;"> B  <div style="border: 1px solid black; padding: 2px;">20</div> </div> <div style="text-align: center;"> C  <div style="border: 1px solid black; padding: 2px;">5A</div> </div> <div style="text-align: center;"> Memory  <table border="1" style="border-collapse: collapse;"> <tr><td style="padding: 2px;">1E</td><td style="padding: 2px;">205A</td></tr> <tr><td style="padding: 2px;">3C</td><td style="padding: 2px;">205B</td></tr> </table> </div> </div>	1E	205A	3C	205B
1E	205A								
3C	205B								
1E	205A								
3C	205B								

### One byte instruction

### Register indirect addressing

**Two machine cycles:** Opcode fetch - 4T

Memory read - 3T

7T

Total number of instructions = 2

LDAX B                      LDAX D

10. STA addr16                                     $(M) \leftarrow (A)$                     or                     $(\text{addr16}) \leftarrow (A)$

The content of the accumulator is moved to the memory. The address of the memory location is given in the instruction. No flags are affected.

<b>Example : STA 2050H</b> <span style="float: right;">(2050<sub>H</sub>) ← (A)</span>									
<i>The content of the accumulator is moved to memory location 2050<sub>H</sub>.</i>									
<b>Before execution</b>	<b>After execution</b>								
<div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;"> <b>A</b>  <div style="border: 1px solid black; padding: 5px; width: 40px; margin: 0 auto;">F4</div> </div> <div style="text-align: center;"> <b>Memory</b>  <table style="border-collapse: collapse; margin: 0 auto;"> <tr> <td style="border-right: 1px solid black; padding: 5px; text-align: center;">06</td> <td style="padding: 5px; text-align: left;">2050</td> </tr> <tr> <td style="border-right: 1px solid black; padding: 5px; text-align: center;">7A</td> <td style="padding: 5px; text-align: left;">2051</td> </tr> </table> </div> </div>	06	2050	7A	2051	<div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;"> <b>A</b>  <div style="border: 1px solid black; padding: 5px; width: 40px; margin: 0 auto;">F4</div> </div> <div style="text-align: center;"> <b>Memory</b>  <table style="border-collapse: collapse; margin: 0 auto;"> <tr> <td style="border-right: 1px solid black; padding: 5px; text-align: center;">F4</td> <td style="padding: 5px; text-align: left;">2050</td> </tr> <tr> <td style="border-right: 1px solid black; padding: 5px; text-align: center;">7A</td> <td style="padding: 5px; text-align: left;">2051</td> </tr> </table> </div> </div>	F4	2050	7A	2051
06	2050								
7A	2051								
F4	2050								
7A	2051								

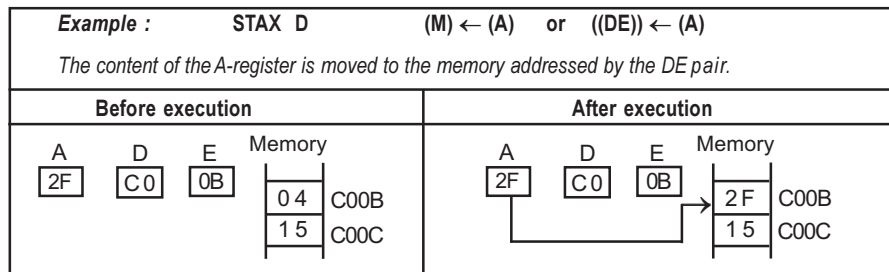
Three byte instruction  
Direct addressing

**Four machine cycles:** Opcode fetch - 4T  
Memory read - 3T  
Memory read - 3T  
Memory write - 3T  
13T

Total number of instructions = 1

# 11. STAX rp (M) ← (A) or ((rp)) ← (A)

The content of the accumulator is moved to the memory addressed by the register pair (rp). (The content of the register pair is the memory address.) The register pair can be either BC or DE.



One byte instruction  
Register indirect addressing

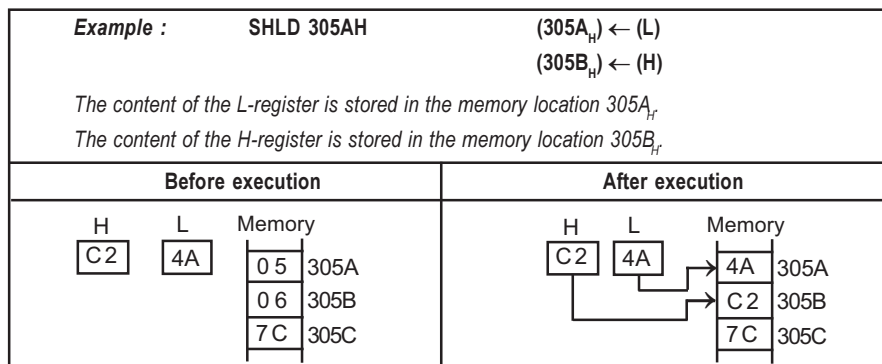
**Two machine cycles:** Opcode fetch - 4T  
Memory write - 3T  
7T

Total number of instructions = 2

STAX B STAX D

# 12. SHLD addr16 (M) ← (L) (addr16) ← (L) (M) ← (H) or (addr16+1) ← (H)

The content of the L-register is stored in the memory location, whose address is given in the instruction. The content of the H-register is stored in the next memory location. No flags are affected.



Three byte instruction  
Direct addressing

**Five machine cycles:** Opcode fetch - 4T  
Memory read - 3T  
Memory read - 3T  
Memory write - 3T  
Memory write - 3T  
16T

Total number of instructions = 1

### 13. SPHL (SP) ← (HL)

The content of the HL pair is moved to the Stack Pointer (SP). No flags are affected.

Example : SPHL (SP) ← (HL)		
The content of the HL pair is copied to the Stack Pointer (SP).		
Before execution		
SP 1016	H C0	L 15
After execution		
SP C015	H C0	L 15

One byte instruction

One machine cycle: Opcode fetch - 6T

Implied addressing

Total number of instructions = 1

### 14. XCHG (E) ↔ (L) (D) ↔ (H)

The content of the HL pair is exchanged with the DE pair. No flags are affected.

Example : XCHG (E) ↔ (L) and (D) ↔ (H)

The content of the E-register is exchanged with the L-register and the content of the D-register is exchanged with the H-register.

Before execution			
D 24	E C7	H A3	L 49
After execution			
D A3	E 49	H 24	L C7

One byte instruction

One machine cycle: Opcode fetch - 4T

Implied addressing

Total number of instructions = 1

### 15. XTHL (HL) ↔ (M) or (HL) ↔ ((SP))

The content of the top of stack is exchanged with the HL pair. Stack is a portion of memory (RAM memory). The content of the Stack Pointer (SP) is the address of the top of the stack. No flags are affected.

Example : XTHL (L) ↔ ((SP)) and (H) ↔ ((SP) + 01)			
The content of memory addressed by the stack pointer is exchanged with the L-register and the content of the next memory location is exchanged with the H-register.			
Before execution			
SP 2000	H 42	L C4	Stack Memory 15 2000 ← Top of Stack 67 2001 AD 2002 D2 2003
After execution			
SP 2000	H 67	L 15	Stack Memory C4 2000 ← Top of Stack 42 2001 AD 2002 D2 2003

One byte instruction

Implied addressing

**Five machine cycles:** Opcode fetch - 4T

Memory read - 3T

Memory read - 3T

Memory write - 3T

Memory write - 3T

16T

Total number of instructions = 1

16. **PUSH rp**                       $(SP) \leftarrow (SP) - 1$  ;       $((SP)) \leftarrow (rp)_H$   
     $(SP) \leftarrow (SP) - 1$  ;       $((SP)) \leftarrow (rp)_L$

The content of the register pair (rp) is pushed to the stack. After execution of this instruction, the content of the Stack Pointer (SP) will be 02 less than the earlier value. The register pairs can be BC, DE, HL and PSW. No flags are affected.

[PSW (Program Status Word) : Accumulator and Flag register together called PSW. Accumulator is high order register and Flag register is low order register.]

**The instruction is executed as follows:**

- The content of the SP is decremented by one.
- The content of the high order register is moved to memory addressed by SP.
- The content of the SP is decremented by one.
- The content of the low order register is moved to memory addressed by SP.

One byte instruction

Register indirect addressing

**Three machine cycles:** Opcode fetch - 6T

Memory write - 3T

Memory write - 3T

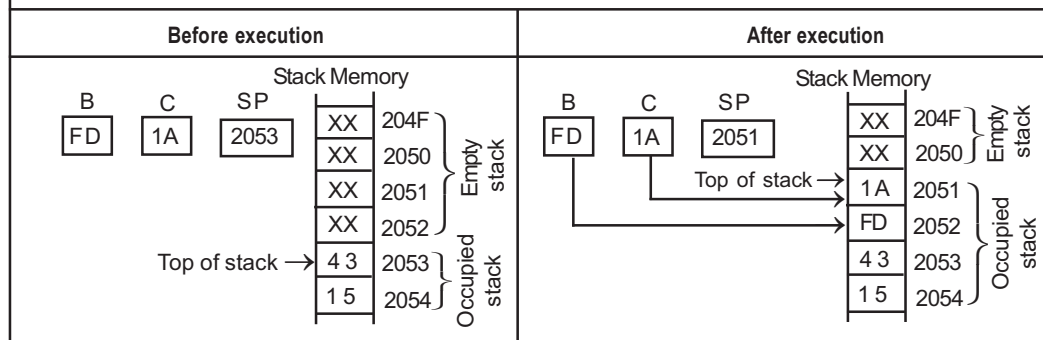
12T

Total number of instructions = 4

**PUSH PSW      PUSH B      PUSH D      PUSH H**

**Example : PUSH B**                       $(SP) \leftarrow (SP) - 01$   
     $((SP)) \leftarrow (B)$   
     $(SP) \leftarrow (SP) - 01$   
     $((SP)) \leftarrow (C)$

- The content of the SP is decremented by one.
- The content of the B-register is moved to the memory addressed by the Stack Pointer (SP).
- Again the content of SP is decremented by one.
- The content of the C-register is moved to the memory addressed by SP.



17. **POP rp**                       $(rp)_L \leftarrow ((SP))$  ;       $(SP) \leftarrow (SP) + 1$   
     $(rp)_H \leftarrow ((SP))$  ;       $(SP) \leftarrow (SP) + 1$

The content of top of stack memory is moved to the register pair. After execution of this instruction the content of the Stack Pointer (SP) will be 02 greater than the earlier value. The register pairs can be BC, DE, HL and PSW. No flags are affected. [PSW (Program Status Word) : Accumulator and Flag register are together called PSW. The accumulator is a high order register and the flag register is a low order register.]

**The pop instruction is executed as follows:**

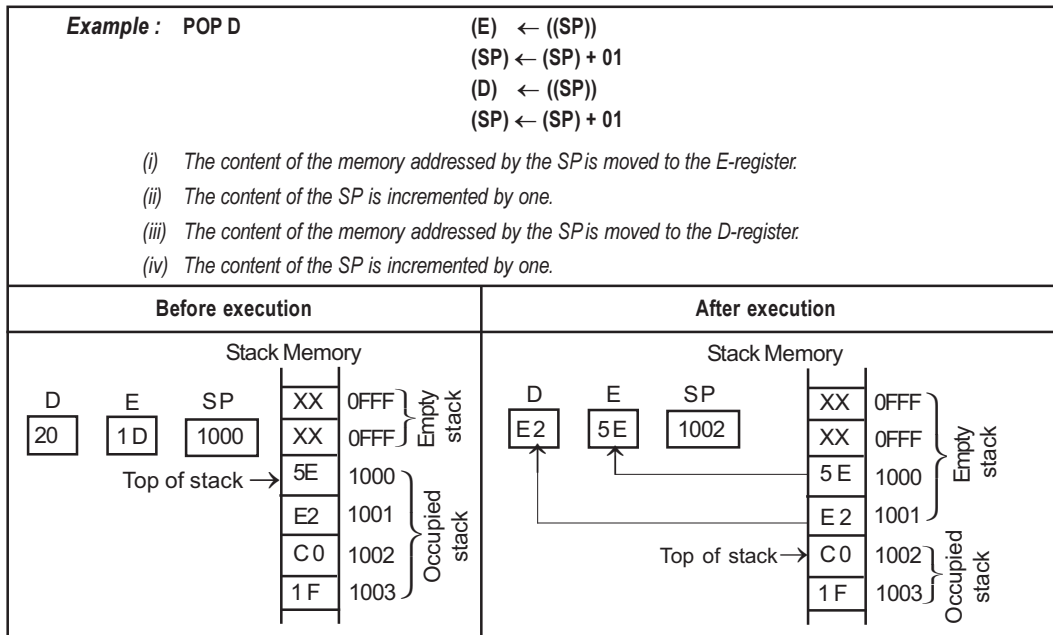
- (i) The content of the memory addressed by the SP is moved to the low order register.
- (ii) The content of the SP is incremented by one.
- (iii) The content of the memory addressed by the SP is moved to the high order register.
- (iv) The content of the SP is incremented by one.

One byte instruction  
 Register indirect addressing

**Three machine cycles:** Opcode fetch - 4T  
 Memory read - 3T  
 Memory read - 3T  
10T

Total number of instructions = 4

**POP PSW      POP B      POP D      POP H**



18. **IN addr8**                       $(A) \leftarrow (addr8)$

The content of the port is moved to the A-register. The 8-bit port address will be given in the instruction. No flags are affected.

Two byte instruction  
 Direct addressing

**Three machine cycles:** Opcode fetch - 4T  
 Memory read - 3T  
 IO read - 3T  
10T

Total number of instructions = 1



**19. OUT addr8 (addr8) ← (A)**

The content of the A-register is moved to the port. The 8-bit port address will be given in the instruction. No flags are affected.

Two byte instruction

Direct addressing

**Three machine cycles:** Opcode fetch - 4T

Memory read - 3T

IO write - 3T

10T

Total number of instructions = 1

**Note :** In an 8085 processor-based system when the IO devices are mapped by IO mapping then the processor can communicate with these IO devices only by using IN and OUT instructions. The processor uses an 8-bit address to select IO-mapped IO devices. With 8-bit address the processor can generate  $2^8 = 256_{10}$  IO addresses.

**2.8 ARITHMETIC INSTRUCTIONS****1. ADD reg (A) ← (A) + (reg)**

The content of the register is added to the content of the accumulator (A-register). After addition the result is stored in the accumulator. All flags are affected. The register can be any one of the general purpose register A, B, C, D, E, H or L.

Before execution		Addition	After execution	
A	E		A	E
C2	B8		7A	B8
CF = 0		$C2_H = 1100\ 0010$	CF = 1	
PF = 0		$B8_H = 1011\ 1000$	PF = 0	
AF = 0		<u>10111 1010</u>	AF = 0	
ZF = 0		Sum = $0111\ 1010 = 7A_H$	ZF = 0	
SF = 0		Carry = 1	SF = 0	
		(Addition is performed in ALU)		

One byte instruction

Register addressing

**One machine cycle:** Opcode fetch - 4T

Total number of instructions = 7

**ADD A    ADD B    ADD C    ADD D    ADD E    ADD H    ADD L**

**2. ADI d8 (A) ← (A) + d8**

The 8-bit data given in the instruction is added to the content of the A-register (Accumulator). After addition, the result is stored in the accumulator. All flags are affected.

Two byte instruction

Immediate addressing

**Two machine cycles:** Opcode fetch - 4T

Memory read - 3T

7T

Total number of instructions = 1

**3. ADD M (A) ← (A) + (M)    or    (A) ← (A) + ((HL))**

The content of memory addressed by HL pair is added to the content of the A-register. After addition, the result is stored in the A-register. All flags are affected.

**Example : ADD M**  $(A) \leftarrow (A) + (M)$  or  $(A) \leftarrow (A) + ((HL))$

Let the content of A be  $44_H$

Let the content of memory location  $C00A_H$  be  $73_H$

The content of the memory location  $C00A_H$  is added to the content of the A-register. The result is put back in the A-register.

Before execution			Addition		After execution		
A	HL	Memory			A	HL	Memory
<div>44</div>	<div>C00A</div>	<div>73</div> C00A	$44_H = 0100\ 0100$		<div>B7</div>	<div>C00A</div>	<div>73</div> C00A
CF = 0		<div>14</div> C00B	$73_H = 0111\ 0011$		CF = 0		<div>14</div> C00B
PF = 0		<div>27</div> C00C	$\begin{array}{r} 0111\ 0011 \\ \hline \end{array}$		PF = 1		<div>27</div> C00C
AF = 0			Sum = B7		AF = 0		
ZF = 0			Carry = 0		ZF = 0		
SF = 0			(Addition is performed in ALU)		SF = 1		

One byte instruction

Register indirect addressing

**Two machine cycles:** Opcode fetch -  $4T$

Memory read -  $3T$

$\underline{7T}$

Total number of instructions = 1

#### 4. ACI d8 $(A) \leftarrow (A) + d8 + CF$

The 8-bit data given in the instruction and the carry flag (the value of carry flag before executing this instruction) are added to the content of the A-register (Accumulator). After addition, the result is stored in the accumulator. All flags are affected.

Two byte instruction

Immediate addressing

**Two machine cycles :** Opcode fetch -  $4T$

Memory read -  $3T$

$\underline{7T}$

Total number of instructions = 1

#### 5. ADC reg $(A) \leftarrow (A) + (reg) + CF$

The content of the register and the carry flag are added to the content of the A-register. After addition, the result is stored in the A-register. All flags are affected. The register can be any one of the general purpose register A, B, C, D, E, H or L.

**Example : ADC H**  $(A) \leftarrow (A) + (H) + CF$

The content of the H-register and the value of the carry flag (before executing this instruction) are added to the content of the A-register. After addition, the result will be in the A-register.

Before execution		Addition		After execution	
A	H			A	H
<div>43</div>	<div>7A</div>	$43_H = 0100\ 0011$		<div>BE</div>	<div>7A</div>
CF = 1		$7A_H = 0111\ 1010$		CF = 0	
PF = 0		CF = 1		PF = 1	
AF = 0		$\begin{array}{r} 0111\ 1110 \\ \hline \end{array}$		AF = 0	
ZF = 0		Sum = $BE_H$		ZF = 0	
SF = 1		Carry = 0		SF = 1	
		(Addition is performed in the ALU)			

**One machine cycle : Opcode fetch - 4T**

### Register addressing

Total number of instructions = 7

ADC A

ADC B

ADC C

ADC D

ADC E

ADC H

ADC L

6. ADC M

$$(A) \leftarrow (A) + (M) + CF \quad \text{or} \quad (A) \leftarrow (A) + ((HL)) + CF$$

The content of the memory addressed by the HL pair and the value of the carry flag (before executing this instruction) are added to the content of A-register. After addition, the result is stored in the A-register. All flags are affected.

*One byte instruction*

**Two machine cycles:** Opcode fetch - 4T

### Register indirect addressing

Memory read - 3T

Total number of instructions = 1

7. SUB req

$$(A) \leftarrow (A) - (\text{reg})$$

The content of the register is subtracted from the content of the accumulator(A-register). After subtraction the result is stored in the A-register. All flags are affected. The register can be any one of the general purpose registerA, B, C, D, E, H or L.

Case ii	
Before execution	Subtraction
<div style="display: flex; justify-content: space-around; align-items: center;"> <div style="border: 1px solid black; padding: 2px;">A</div> <div style="border: 1px solid black; padding: 2px;">C</div> </div> <div style="display: flex; justify-content: space-around; align-items: center; margin-top: 5px;"> <div style="border: 1px solid black; padding: 2px;">89</div> <div style="border: 1px solid black; padding: 2px;">C4</div> </div> <div style="margin-top: 10px;"> <p>CF = 0</p> <p>PF = 0</p> <p>AF = 0</p> <p>ZF = 1</p> <p>SF = 1</p> </div>	<p><math>89_H = 1000\ 1001</math></p> <p><math>C4_H = 1100\ 0100</math></p> <p>1's complement of <math>C4_H = 0011\ 1011</math></p> <p>2's complement of <math>C4_H = 0011\ 1011 + 1</math></p> <p style="text-align: right;"><math>= 0011\ 1100 = 3C_H</math></p>

Case ii continued ...	
After execution	Subtraction
<div style="display: flex; justify-content: space-around;"> <div style="border: 1px solid black; padding: 2px;">A</div> <div style="border: 1px solid black; padding: 2px;">C</div> </div> <div style="display: flex; justify-content: space-around;"> <div style="border: 1px solid black; padding: 2px;">C5</div> <div style="border: 1px solid black; padding: 2px;">C4</div> </div> <div style="margin-top: 10px;"> CF = 1  PF = 1  AF = 1  ZF = 0  SF = 1 </div>	$89_H = 1000\ 1001$ $+3C_H = 0011\ 1100$ <hr/> <div style="display: flex; align-items: center;"> <div style="border: 1px solid black; padding: 2px;">0</div> <div style="margin: 0 5px;">1100 0101</div> </div> <div style="margin-top: 5px;"> Complement  Carry </div> <div style="margin-top: 5px;"> <div style="border: 1px solid black; padding: 2px;">1</div> <div style="margin-left: 10px;">C 5</div> </div> <div style="margin-top: 10px;"> Result = <math>C5_H</math>  CF = 1 </div> <div style="border: 1px solid black; padding: 2px; margin-top: 5px;"> Note : 2's complement of <math>C5_H = 3B_H</math> </div>

**Note :** The 8085 microprocessor performs 2's complement subtraction. But after subtraction, it will complement the carry alone. In 2's complement subtraction, if  $CF = 1$ , then the result is positive and if  $CF = 0$ , then the result is negative. Since, the 8085 processor complements the carry after subtraction, here if  $CF = 0$ , then the result is positive and if  $CF = 1$ , then the result is negative. If the result is negative, then it will be in 2's complement form.

One byte instruction

One machine cycle: Opcode fetch - 4T

Register addressing

Total number of instructions = 7

SUB A      SUB B      SUB C      SUB D      SUB E      SUB H      SUB L

#### 8. SUI d8 (A) ← (A) - d8

The 8-bit data given in the instruction is subtracted from the A-register (accumulator). After subtraction, the result is stored in the A-register. All flags are affected.

Two byte instruction

Two machine cycles : Opcode fetch - 4T

Immediate addressing

Memory read - 3T

---

7T

Total number of instructions = 1

#### 9. SUB M (A) ← (A) - (M) or (A) ← (A) - ((HL))

The content of the memory addressed by the HLpair is subtracted from the A-register. After subtraction, the result is stored in the A-register. All flags are affected.

One byte instruction

Two machine cycles : Opcode fetch - 4T

Register indirect addressing

Memory read - 3T

---

7T

Total number of instructions = 1

#### 10. SBB reg (A) ← (A) - (reg) - CF

The content of the register and the value of carry (before executing this instruction) are subtracted from the accumulator (A-register). After subtraction, the result is stored in the accumulator. All flags are affected. The register can be any one of the general purpose register A, B, C, D, E, H or L.

One byte instruction

One machine cycle : Opcode fetch - 4T

Register addressing

Total number of instructions = 7

SBB A      SBB B      SBB C      SBB D      SBB E      SBB H      SBB L

**11. SBI d8  $(A) \leftarrow (A) - d8 - CF$** 

The 8-bit data given in the instruction and the value of carry (before executing this instruction) are subtracted from accumulator. After subtraction, the result is stored in the accumulator. All flags are affected.

Two byte instruction	<b>Two machine cycles :</b> Opcode fetch - 4T
Immediate addressing	Memory read - 3T
	<u>7T</u>

Total number of instructions = 1

**12. SBB M  $(A) \leftarrow (A) - (M) - CF$  or  $(A) \leftarrow (A) - ((HL)) - CF$** 

The content of the memory addressed by HL and the value of carry (before executing this instruction) are subtracted from accumulator (A-register). After subtraction, the result is stored in the A-register. All flags are affected.

One byte instruction	<b>Two machine cycles:</b> Opcode fetch - 4T
Register indirect addressing	Memory read - 3T
	<u>7T</u>

Total number of instructions = 1

**13. DAA  
(DAA - Decimal Adjust Accumulator)**

After BCD addition, the DAA instruction is executed to get the result in BCD. When DAA instruction is executed, the content of the accumulator is altered or adjusted as explained below :

- i) If the sum of the lower nibbles exceeds  $09_{16}$  or auxiliary carry is set, then a correction  $06_{16}$  (0110) is added to sum of lower nibbles.
- ii) If the sum of the upper nibbles exceeds  $09_{16}$  or carry is set, then a correction  $06_{16}$  (0110) is added to sum of upper nibble.

After executing this instruction all flags are modified to indicate the status of the result.

One byte instruction	<b>One machine cycle:</b> Opcode fetch - 4T
Implied addressing	

Total number of instructions = 1

**14. DAD rp  $(HL) \leftarrow (HL) + (rp)$   
(DAD - Double Addition)**

The content of the register pair is added to the content of the HL pair. After addition, the result is stored in the HL pair. Only the carry flag is affected. The register pair can be BC, DE, HL or SP.

One byte instruction	<b>Three machine cycles:</b> Opcode fetch - 4T
Register addressing	Bus idle - 3T
	Bus idle - 3T
	<u>10T</u>

Total number of instructions = 4

**DAD B      DAD D      DAD H      DAD SP**

**15. INR reg**  $(reg) \leftarrow (reg) + 01$

The content of the register is incremented by one. Except carry flag, all other flags are affected. The register can be any one of the general purpose register A, B, C, D, E, H or L.

<b>Example : INR B</b> $(B) \leftarrow (B) + 01$		
The content of the B-register is incremented by one. The increment operation is performed by adding $01_H$ to the content of B-register.		
Before execution	Increment Operation	After execution
<div>B</div> <div>CF = 0</div> <div>PF = 0</div> <div>AF = 0</div> <div>ZF = 0</div> <div>SF = 0</div> <div>4A</div>	<div><math>4A_H = 0100\ 1010</math></div> <div><math>+ 01_H = 0000\ 0001</math></div> <div><hr/></div> <div><math>0100\ 1011</math></div> <div><hr/></div> <div>4 B</div>	<div>B</div> <div>CF = 0</div> <div>PF = 1</div> <div>AF = 0</div> <div>ZF = 0</div> <div>SF = 0</div> <div>4B</div>

One byte instruction

One machine cycle: Opcode fetch - 4T

Register addressing

Total number of instructions = 7

INR A    INR B    INR C    INR D    INR E    INR H    INR L

**16. INR M**  $(M) \leftarrow (M) + 01$     or     $((HL)) \leftarrow ((HL)) + 01$

The content of the memory addressed by the HL pair is incremented by one. Except carry, all other flags are affected.

<b>Example : INR M</b> $(M) \leftarrow (M) + 01$		
Let the content of the HL pair be $C00A_H$ . Let the content of memory location $C00A_H$ be $C5_H$ . The content of the memory location $C00A_H$ is incremented by one. The increment operation is performed by adding $01_H$ to the content of the memory.		
Before execution	Increment Operation	After execution
<div>HL</div> <div>C00A</div> <div>CF = 0</div> <div>PF = 0</div> <div>AF = 0</div> <div>ZF = 0</div> <div>SF = 0</div> <div>Memory</div> <div>C5 C00A</div> <div>A2 C00B</div> <div>07 C00C</div>	<div><math>C5_H = 1100\ 0101</math></div> <div><math>+ 01_H = 0000\ 0001</math></div> <div><hr/></div> <div><math>1100\ 0110</math></div> <div><hr/></div> <div>C 6</div>	<div>HL</div> <div>C00A</div> <div>CF = 0</div> <div>PF = 1</div> <div>AF = 0</div> <div>ZF = 0</div> <div>SF = 1</div> <div>Memory</div> <div>C6 C00A</div> <div>A2 C00B</div> <div>07 C00C</div>

One byte instruction

Three machine cycles : Opcode fetch - 4T

Register indirect addressing

Memory read - 3T

Memory write - 3T

---

10T

Total number of instructions = 1

**17. DCR reg**  $(reg) \leftarrow (reg) - 01$

The content of the register is decremented by one. Except carry, all other flags are affected. The register can be A, B, C, D, E, H or L.

<b>Example : DCR D</b> $(D) \leftarrow (D) - 01$	
The content of the D-register is decremented by one. The decrement operation is performed by subtracting $01_H$ from the content of the D-register.	

Before execution	Decrement operation
D      CF = 0 <div style="border: 1px solid black; padding: 2px; display: inline-block;">60</div> PF = 0 AF = 0 ZF = 0 SF = 0	$01_H = 0000\ 0001$ 1's complement of $01_H = 1111\ 1110$ 2's complement of $01_H = 1111\ 1110 + 1$ $= 1111\ 1111 = FF_H$
After execution	
D      CF = 0 <div style="border: 1px solid black; padding: 2px; display: inline-block;">5F</div> PF = 1 AF = 0 ZF = 0 SF = 0	$60_H = 0110\ 0000$ $+ FF_H = 1111\ 1111$ <hr/> <div style="display: inline-block; border: 1px solid black; padding: 2px;">1</div> 0101 1111 <hr/> 5      F Carry is discarded

One byte instruction

One machine cycle : Opcode fetch - 4T

Register addressing

Total number of instructions = 7

DCR A      DCR B      DCR C      DCR D      DCR E      DCR H      DCR L

### 18. DCR M      $(M) \leftarrow (M) - 01$      or      $((HL)) \leftarrow ((HL)) - 01$

The content of memory addressed by the HL pair is decremented by one. Except carry, all other flags are affected.

Before execution		Decrement operation
HL <div style="border: 1px solid black; padding: 2px; display: inline-block;">2010</div> CF = 0 PF = 0 AF = 0 ZF = 0 SF = 0	Memory <div style="border: 1px solid black; padding: 2px; display: inline-block;">FA</div> 2010 <div style="border: 1px solid black; padding: 2px; display: inline-block;">02</div> 2011	$01_H = 0000\ 0001$ 1's complement of $01_H = 1111\ 1110$ 2's complement of $01_H = 1111\ 1110 + 1$ $= 1111\ 1111 = FF_H$
After execution		
HL <div style="border: 1px solid black; padding: 2px; display: inline-block;">2010</div> CF = 0 PF = 1 AF = 1 ZF = 0 SF = 1	Memory <div style="border: 1px solid black; padding: 2px; display: inline-block;">F9</div> 2010 <div style="border: 1px solid black; padding: 2px; display: inline-block;">02</div> 2011	$FA_H = 1111\ 1010$ $+ FF_H = 1111\ 1111$ <hr/> <div style="display: inline-block; border: 1px solid black; padding: 2px;">1</div> 1111 1001 <hr/> F      9 Carry is discarded

One byte instruction

Three machine cycles : Opcode fetch - 4T

Register indirect addressing

Memory read - 3T

Memory write - 3T

---

10T

Total number of instructions = 1

**19. INX rp**  $(rp) \leftarrow (rp) + 01$ 

The content of the register pair is incremented by one. The register pair can be BC, DE, HL or SP. No flags are affected.

<b>Example : INX H</b> $(HL) \leftarrow (HL) + 01$ The content of the HL pair is incremented by one.	
Before execution	After execution
H L <div style="border: 1px solid black; padding: 2px; display: inline-block;">00FF</div>	H L <div style="border: 1px solid black; padding: 2px; display: inline-block;">0100</div>

One byte instruction

One machine cycle : Opcode fetch - 6T

Register addressing

Total number of instructions = 4

INX B      INX D      INX H      INX SP

**20. DCX rp**  $(rp) \leftarrow (rp) - 01$ 

The content of the register pair is decremented by one. The register pair can be BC, DE, HL or SP. No flags are affected.

<b>Example : DCX SP</b> $(SP) \leftarrow (SP) - 01$ The content of the stack pointer is decremented by one.	
Before execution	After execution
SP <div style="border: 1px solid black; padding: 2px; display: inline-block;">1000</div>	SP <div style="border: 1px solid black; padding: 2px; display: inline-block;">0FFF</div>

One byte instruction

One machine cycle : Opcode fetch - 6T

Register addressing

Total number of instructions = 4

DCX B      DCX D      DCX H      DCX SP

**2.9 LOGICAL INSTRUCTIONS****1. ANA reg**  $(A) \leftarrow (A) \& (reg)$ 

(& is the symbol used for logical AND operation)

The content of the register is logically ANDed bit by bit with the content of the accumulator. In bit by bit AND operation, the bit  $D_0$  of register is ANDed with the bit  $D_0$  of A-register, the bit  $D_1$  of register is ANDed with bit  $D_1$  of A-register, and so on. The register can be any one of the general purpose register A, B, C, D, E, H or L. After execution of the instruction, carry flag is always reset and auxiliary carry flag is always set. Other flags are altered (according to the results). After AND operation, result is stored in accumulator.

<b>Example :</b> ANA E										(A) ← (A) & (E)									
The content of E-register is logically ANDed bit by bit with the content of accumulator.																			
Before execution						AND operation						After execution							
A		E		CF = 0		15 <sub>H</sub> = 0001 0101						A		E		CF = 0			
<div>15</div>		<div>E2</div>		PF = 0		E2 <sub>H</sub> = 1110 0010						<div>00</div>		<div>E2</div>		PF = 1			
				AF = 0		<div>0000 0000</div>										AF = 1			
				ZF = 0		<div>0 0</div>										ZF = 1			
				SF = 0												SF = 0			



One byte instruction

**One machine cycle:** Opcode fetch - 4T

Register addressing

Total number of instructions = 7

**ANA A**
**ANA B**
**ANA C**
**ANA D**
**ANA E**
**ANA H**
**ANA L**
**2. ANI d8**
**(A) ← (A) & d8**

The 8-bit data given in the instruction is logically ANDed bit by bit with the content of the accumulator. The result is stored in the accumulator. After execution of this instruction, CF = 0 and AF = 1. Other flags are affected.

Two byte instruction

**Two machine cycles :** Opcode fetch - 4T

Immediate addressing

Memory read - 3T

7T

Total number of instructions = 1

**3. ANA M**
**(A) ← (A) & (M) or (A) ← (A) & ((HL))**

The content of the memory addressed by the HL pair is logically ANDed bit by bit with the content of the accumulator. The result is stored in the accumulator. After execution, CF = 0 and AF = 1. Other flags are affected.

**Example : ANA M**
**(A) ← (A) & (M)**

Let the content of HL be 105A<sub>H</sub>. Let the content of the memory location 105A<sub>H</sub> be 4C<sub>H</sub>. The content of the memory location 105A<sub>H</sub> is logically ANDed bit by bit with the content of the accumulator. The result is stored in the accumulator.

Before execution			AND operation	After execution		
A	HL	Memory		A	HL	Memory
27	105A	14 1059	27 <sub>H</sub> = 0010 0111	04	105A	14 1059
CF = 0		4C 105A	4C <sub>H</sub> = 0100 1100	CF = 0		
PF = 0			0000 0100	PF = 0		
AF = 0			0 4	AF = 1		
ZF = 0				ZF = 0		
SF = 0				SF = 0		

One byte instruction

**Two machine cycles:** Opcode fetch - 4T

Register indirect addressing

Memory read - 3T

7T

Total number of instructions = 1

**4. ORA reg**
**(A) ← (A) | (reg)**

(| is the symbol used for logical OR operation)

The content of the register is logically ORed bit by bit with the content of the accumulator. In bit by bit OR operation, the bit D<sub>0</sub> of the register is ORed with bit D<sub>0</sub> of the A-register, the bit D<sub>1</sub> of the register is ORed with bit D<sub>1</sub> of the A-register, and so on. The register can be any one of the general purpose register A, B, C, D, E, H or L. After execution of the instruction, both the carry and auxiliary flags are always reset (AF = 0, CF = 0). Other flags are modified (according to the result). After OR operation, the result is stored in the accumulator.

One byte instruction

**One machine cycle:** Opcode fetch - 4T

Register addressing

<b>Example : ORA B</b>			<b>(A) ← (A)   (B)</b>			
The content of the B-register is logically ORed bit by bit with the content of the accumulator.						
<b>Before execution</b>			<b>OR operation</b>		<b>After execution</b>	
A	B	CF = 0	04 <sub>H</sub> = 0000 0100	A	B	CF = 0
04	7A	PF = 0	7A <sub>H</sub> = 0111 1010	7E	7A	PF = 1
		AF = 0	0111 1110			AF = 0
		ZF = 0	7 E			ZF = 0
		SF = 0				SF = 0

Total number of instructions = 7

ORA A      ORA B      ORA C      ORA D      ORA E      ORA H      ORA L

5. **ORA M**      (A) ← (A) | (M)      or      (A) ← (A) | ((HL))

The content of the memory addressed by the HL pair is logically ORed bit by bit with the content of the accumulator. The result is stored in the accumulator. After execution, CF = AF = 0. Other flags are affected.

Example : ORA M			(A) ← (A)   (M)			
Let the content of the HL pair be 2050 <sub>H</sub> . Let the content of memory location 2050 <sub>H</sub> be 1B <sub>H</sub> . The content of the memory location 2050 <sub>H</sub> is logically ORed bit by bit with the content of the accumulator. The result is stored in the accumulator.						
Before execution			OR operation	After execution		
A	HL	Memory	45 <sub>H</sub> = 0100 0101	A	HL	Memory
45	2050	1B 2050	1B <sub>H</sub> = 0001 1011	5F	2050	1B 2050
		07 2051	<div>0101 1111</div>			07 2051
CF = 0			5 F	CF = 0		
PF = 0				PF = 1		
AF = 0				AF = 0		
ZF = 0				ZF = 0		
SF = 0				SF = 0		

One byte instruction

Two machine cycles: Opcode fetch - 4T

Register indirect addressing

Memory read - 3T

7T

Total number of instructions = 1

6. **ORI d8**      (A) ← (A) | d8

The 8-bit data given in the instruction is logically ORed bit by bit with the content of the accumulator. The result is stored in the accumulator. After execution of this instruction, CF = AF = 0. Other flags are affected.

Two byte instruction

Two machine cycles : Opcode fetch - 4T

Immediate addressing

Memory read - 3T

7T

Total number of instructions = 1

7. **XRA reg**      (A) ← (A) ^ (reg)

( ^ is the symbol used for logical EXCLUSIVE-OR operation).

The content of the register is logically EXCLUSIVE-ORed bit by bit with the content of the accumulator. In bit by bit EXCLUSIVE-OR operation, the bit D<sub>0</sub> of register is EXCLUSIVE-ORed with bit D<sub>0</sub> of A-register, the bit D<sub>1</sub> of register is EXCLUSIVE-ORed with bit D<sub>1</sub> of A-register, and so on. The result is stored in the accumulator. The register can be any one of the general purpose register A, B, C, D, E, H or L. After execution AF = CF = 0. Other flags are modified (according to the result).

<b>Example : XRA A</b> $(A) \leftarrow (A) \wedge (A)$ The content of the A-register is EXCLUSIVE-ORed bit by bit with the content of the A-register itself.		
Before execution	EXCLUSIVE-OR operation	After execution
A    CF = 1 74   PF = 0 AF = 1 ZF = 0 SF = 1	$74_H = 0111\ 0100$ $74_H = \begin{array}{r} 0111\ 0100 \\ 0000\ 0000 \\ \hline \end{array}$	A    CF = 0 00   PF = 1 AF = 0 ZF = 1 SF = 0

One byte instruction

One machine cycle: Opcode fetch - 4T

Register addressing

Total number of instructions = 7

XRA A    XRA B    XRA C    XRA D    XRA E    XRA H    XRA L

### 8. XRI d8      $(A) \leftarrow (A) \wedge d8$      or      $(A) \leftarrow (A) \wedge d8$

The 8-bit data given in the instruction is logically EXCLUSIVE-ORed bit by bit with the content of the accumulator. The result is stored in the accumulator. After execution of this instruction, CF = AF = 0. Other flags are affected.

Two byte instruction

Two machine cycles : Opcode fetch - 4T

Immediate addressing

Memory read -  $\frac{3T}{7T}$ 

Total number of instructions = 1

### 9. XRA M      $(A) \leftarrow (A) \wedge (M)$      or      $(A) \leftarrow (A) \wedge (HL)$

The content of the memory addressed by the HL pair is logically EXCLUSIVE-ORed bit by bit with the content of accumulator. The result is stored in accumulator. After execution, CF = AF = 0. Other flags are affected.

<b>Example : XRA M</b> $(A) \leftarrow (A) \wedge (M)$ Let the content of the HL pair be 805A <sub>H</sub> . Let the content of memory location 805A <sub>H</sub> be C4 <sub>H</sub> . The content of the memory location 805A <sub>H</sub> is logically EXCLUSIVE-ORed bit by bit with the content of the accumulator. The result will be in the accumulator.		
Before execution	EXCLUSIVE-OR operation	After execution
A    HL    Memory B7   805A   1C 8059 CF = 1    C4 805A PF = 1    20 805B AF = 1    51 805C ZF = 0 SF = 1	$B7_H = 1011\ 0111$ $C4_H = \begin{array}{r} 1100\ 0100 \\ 0111\ 0011 \\ \hline 7\ 3 \end{array}$	A    HL    Memory 73   805A   1C 8059 CF = 0    C4 805A PF = 0    20 805B AF = 0    51 805C ZF = 0 SF = 0

One byte instruction

Two machine cycles : Opcode fetch - 4T

Register indirect addressing

Memory read -  $\frac{3T}{7T}$ 

Total number of instructions = 1

**10. CMP reg****(A) – (reg) ⇒ Modify flags**

The content of the register is compared with the accumulator. The comparison is performed by subtracting the content of register from the A-register. The subtraction is performed in the ALU, and the result is used to modify flags and then the result is discarded (i.e., it is not stored in any register). After execution of this instruction, the content of accumulator and the register are not altered. All flags are affected by this instruction. The register can be any one of the general purpose register A, B, C, D, E, H or L.

The status of carry and zero flag after comparison are given below :

- i) If (A) < (reg) then the carry flag is set (i.e., CF = 1)
- ii) If (A) > (reg) then the carry flag is reset or cleared (i.e., CF = 0)
- iii) If (A) = (reg) then the zero flag is set (i.e., ZF = 1).

**Example : CMP B****(A) – (B) ⇒ Modify flags.**

The content of the B-register is compared with the accumulator. The comparison is performed by subtracting the content of the B-register from the content of the accumulator. The subtraction is performed in the ALU and the result is used to modify the flags and then discarded. The content of the accumulator and the B-register are not altered.

Before execution	Comparison	After execution
<div style="display: flex; justify-content: space-around;"> <div>A</div> <div>B</div> </div> <div style="display: flex; justify-content: space-around;"> <div>15</div> <div>C2</div> </div> <div style="margin-top: 10px;">           CF = 0            PF = 0            AF = 0            ZF = 0            SF = 0         </div>	<div style="text-align: center; margin-top: 10px;"> <math>C2_H = 1100\ 0010</math>            1's complement of <math>C2_H = 0011\ 1101</math>            2's complement of <math>C2_H = 0011\ 1101 + 1</math>  <math>= 0011\ 1110 = 3E_H</math>  <math>15_H = 0001\ 0101</math>  <math>+ 3E_H = 0011\ 1110</math>  <hr/> <math>0101\ 0011</math>            Complement            Carry  <div style="display: flex; align-items: center;"> <div style="border: 1px solid black; padding: 2px 5px;">1</div> <div style="margin: 0 10px;">5</div> <div style="margin: 0 10px;">3</div> </div> </div>	<div style="display: flex; justify-content: space-around;"> <div>A</div> <div>B</div> </div> <div style="display: flex; justify-content: space-around;"> <div>15</div> <div>C2</div> </div> <div style="margin-top: 10px;">           CF = 1            PF = 1            AF = 1            ZF = 0            SF = 0         </div>

One byte instruction

One machine cycle: Opcode fetch - 4T

Register addressing

Total number of instructions = 7

<b>CMP A</b>	<b>CMP B</b>	<b>CMP C</b>	<b>CMP D</b>	<b>CMP E</b>	<b>CMP H</b>	<b>CMP L</b>
--------------	--------------	--------------	--------------	--------------	--------------	--------------

**11. CPI d8****(A) – d8 ⇒ Modify flags.**

The 8-bit data given in the instruction is compared with the accumulator. The comparison is performed by subtracting the 8-bit data from the A-register. The subtraction is performed in ALU and the result is used to modify flags and then discarded. After execution of the instruction, the content of the accumulator is not altered. All flags are affected.

The status of carry and zero flag after comparison are given below :

- i) If (A) < d8 then the carry flag is set (i.e., CF = 1)
- ii) If (A) > d8 then the carry flag is reset or cleared (i.e., CF = 0)
- iii) If (A) = d8 then the zero flag is set (i.e., ZF = 1).

Two byte instruction

Two machine cycles : Opcode fetch - 4T

Immediate addressing

Memory read - 3T

7T

Total number of instructions = 1

**12. CMP M                    (A) – (M)  $\Rightarrow$  Modify flags or (A) – ((HL))  $\Rightarrow$  Modify flags.**

The content of the memory addressed by HL pair is compared with the accumulator. The comparison is performed by subtracting the content of memory from the A-register. The subtraction is performed in the ALU and the result is used to modify flags and then discarded. After execution of the instruction, the content of the accumulator and the memory are not altered. All flags are affected by this instruction.

The status of carry and zero flag after comparison are given below:

- i) If (A) < (M) then the carry flag is set (i.e., CF = 1).
- ii) If (A) > (M) then the carry flag is reset or cleared (i.e., CF = 0).
- iii) If (A) = (M) then the zero flag is set (i.e., ZF = 1).

**Example : CMP M**

Let the content of the HL pair be C050<sub>H</sub>. Let the content of the memory location C050<sub>H</sub> be 7A<sub>H</sub>. The content of the memory location C050<sub>H</sub> is compared with the content of the accumulator. Only flags are altered. The content of the accumulator and the memory remains the same.

Before execution	Comparison	After execution
<div style="display: flex; justify-content: space-around;"> <div>A <div style="border: 1px solid black; padding: 2px;">25</div></div> <div>HL <div style="border: 1px solid black; padding: 2px;">C050</div></div> </div> <div style="margin-top: 10px;">           Memory  <div style="display: flex; justify-content: space-between;"> <div style="border: 1px solid black; padding: 2px;">7A</div> <div>C050</div> </div> <div style="display: flex; justify-content: space-between;"> <div style="border: 1px solid black; padding: 2px;">10</div> <div>C051</div> </div> </div> <div style="margin-top: 10px;">           CF = 0            PF = 0            AF = 0            ZF = 0            SF = 0         </div>	<div style="text-align: center;"> <math>25_H = 0010\ 0101</math>  <math>7A_H = 0111\ 1010</math>            1's complement of <math>7A_H = 1000\ 0101</math>            2's complement of <math>7A_H = 1000\ 0101 + 1</math>  <math>= 1000\ 0110 = 86_H</math>  <math>25_H = 0010\ 0101</math>  <math>+ 86_H = 1000\ 0110</math>  <div style="border-top: 1px solid black; display: inline-block; padding-top: 2px;"><math>01010\ 1011</math></div>            Complement            Carry <math>\downarrow</math>  <div style="border: 1px solid black; padding: 2px;">1</div> </div> <div style="text-align: center; margin-top: 10px;"> <math>A \quad B</math> </div>	<div style="display: flex; justify-content: space-around;"> <div>A <div style="border: 1px solid black; padding: 2px;">25</div></div> <div>HL <div style="border: 1px solid black; padding: 2px;">C050</div></div> </div> <div style="margin-top: 10px;">           Memory  <div style="display: flex; justify-content: space-between;"> <div style="border: 1px solid black; padding: 2px;">7A</div> <div>C050</div> </div> <div style="display: flex; justify-content: space-between;"> <div style="border: 1px solid black; padding: 2px;">10</div> <div>C051</div> </div> </div> <div style="margin-top: 10px;">           CF = 1            PF = 0            AF = 0            ZF = 0            SF = 1         </div>

One byte instruction

Register indirect addressing

**Two machine cycles:** Opcode fetch - 4T

Memory read - 3T

7T

Total number of instructions = 1

**13. CMA                                    (A)  $\leftarrow$  ( $\bar{A}$ )**

**(CMA - Complement Accumulator)**

The content of the accumulator is complemented. No flags are affected.

One byte Instruction

Implied addressing

**One machine cycle:** Opcode fetch - 4T

**14. STC                                    (CF)  $\leftarrow$  1**

**(STC - Set Carry)**

The carry flag is set to 1. Only carry flag is affected by this instruction.

One byte instruction

Implied addressing

**One machine cycle:** Opcode fetch - 4T

15. CMC  $(CF) \leftarrow \overline{(CF)}$

(CMC - Complement Carry)

The carry flag is complemented. Only the carry flag is affected by this instruction.

One byte instruction

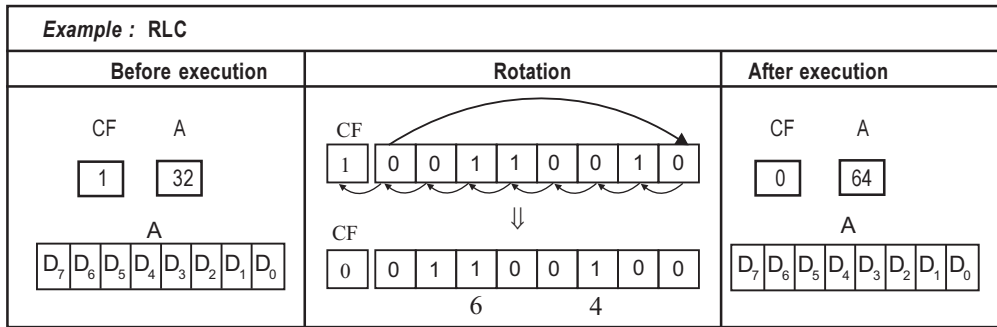
One machine cycle: Opcode fetch - 4T

Implied addressing

16. RLC  $D_{n+1} \leftarrow D_n$  ;  $D_0 \leftarrow D_7$  and  $(CF) \leftarrow D_7$

(RLC - Rotate Accumulator Left to carry)

The content of the A-register is rotated left by one bit and the left most bit of A-register is rotated to the carry. [The left most bit is most significant bit.] Only the carry flag is affected.



One byte instruction

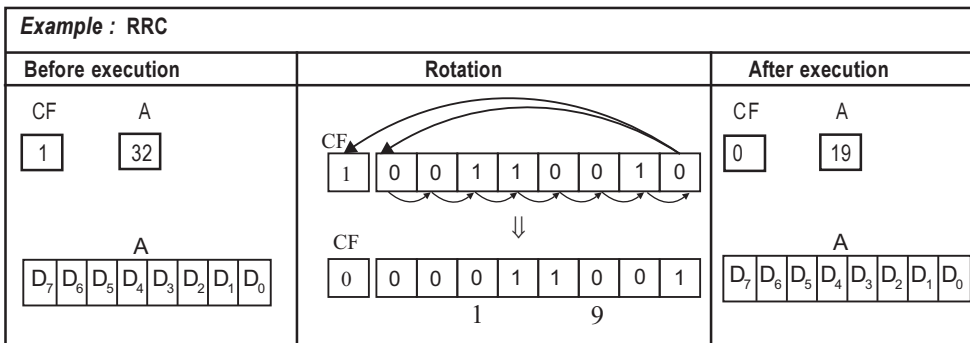
One machine cycle: Opcode fetch - 4T

Implied addressing

17. RRC  $D_n \leftarrow D_{n+1}$  ;  $D_7 \leftarrow D_0$  and  $(CF) \leftarrow D_0$

(RRC - Rotate Accumulator Right to Carry)

The content of A-register is rotated right by one bit and the right most bit of A-register is rotated to carry. [The right most bit is least significant bit.] Only carry flag is affected.



One byte instruction

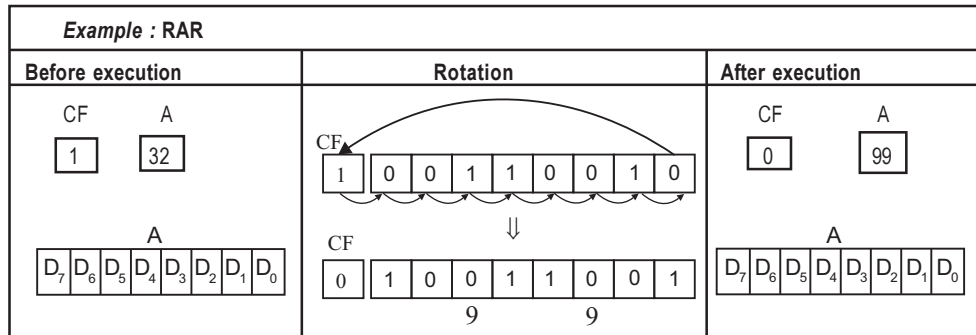
One machine cycle: Opcode fetch - 4T

Implied addressing

18. RAR  $D_n \leftarrow D_{n+1}$  ;  $D_7 \leftarrow (CF)$  and  $(CF) \leftarrow D_0$

(RAR - Rotate Accumulator Right through carry)

The content of the A-register along with the carry is rotated right by one bit. Here the carry is moved to the most significant bit position (D<sub>7</sub>) and the least significant bit (D<sub>0</sub>) is moved to the carry. Only the carry flag is affected.

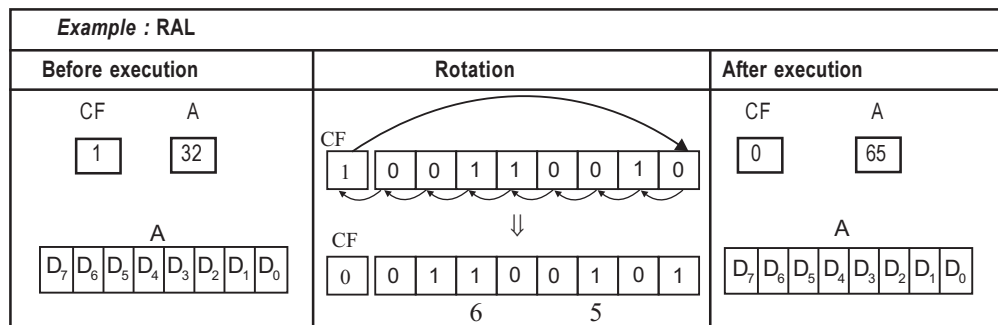


One byte instruction  
Implied addressing

One machine cycle: Opcode fetch - 4T

19. **RAL**  $D_{n+1} \leftarrow D_n$  ;  $D_0 \leftarrow (CF)$  and  $(CF) \leftarrow D_7$   
(RAL - Rotate Accumulator Left through carry)

The content of the A-register along with the carry is rotated left by one bit. Here the carry is moved to the least significant bit position ( $D_0$ ) and the most significant bit ( $D_7$ ) is moved to the carry. Only the carry flag is affected.



One byte instruction  
Implied addressing

One machine cycle: Opcode fetch - 4T

## 2.10 BRANCHING INSTRUCTIONS

1. **JMP addr16** (PC)  $\leftarrow$  addr16

It is unconditional jump instruction. When this instruction is executed, the address given in the instruction is moved to the program counter. Now, the processor starts executing the instructions stored in this address.

Three byte instruction  
Immediate addressing

Three machine cycles: Opcode fetch - 4T

Memory read - 3T

Memory read - 3T  
10T

2. **J <condition> addr16**  
If <condition> is TRUE then,

(PC)  $\leftarrow$  addr16

It is conditional jump instruction. The conditional jump instruction will check a flag condition. If the flag condition is true, then the address given in the instruction is moved to the program counter. Thus the program control is branched to the jump address. If the flag condition is false, then the next instruction is executed.

There are eight conditional jump instructions.

- i) **JZ addr16** ;Jump on Zero - Jump if zero flag = 1.
- ii) **JNZ addr16** ;Jump on Not Zero - Jump if zero flag = 0.
- iii) **JC addr16** ;Jump on Carry - Jump if carry flag = 1.
- iv) **JNC addr16** ;Jump on No Carry - Jump if carry flag = 0.
- v) **JM addr16** ;Jump on Minus - Jump if sign flag = 1.
- vi) **JP addr16** ;Jump on Positive - Jump if sign flag = 0.
- vii) **JPE addr16** ;Jump on Parity Even - Jump if parity flag = 1.
- viii) **JPO addr16** ;Jump on Parity Odd - Jump if parity flag = 0.

Three byte instruction

Two or three machine cycles:

Condition False

Condition True

Immediate addressing

Opcode fetch - 4T

Opcode fetch - 4T

Memory read - 3T

Memory read - 3T

Memory read - 3T

7T

10T

### 3. **CALL addr16**

$(SP) \leftarrow (SP) - 1$  ;  $((SP)) \leftarrow (PC)_H$

$(SP) \leftarrow (SP) - 1$  ;  $((SP)) \leftarrow (PC)_L$

$(PC) \leftarrow \text{addr16}$

It is unconditional CALL used to call a subroutine program. When this instruction is executed, the address of the next instruction in the program counter is pushed to the stack. The 16-bit address (which is the address of the subroutine program) given in the instruction is loaded in the program counter. Now, the processor will start executing the instructions stored in this call address.

Three byte instruction

Five machine cycles:

Opcode fetch - 6T

Memory read - 3T

Memory read - 3T

Memory write - 3T

Memory write - 3T

18T

### 4. **C<condition> addr16**

If <condition> is TRUE then,

$(SP) \leftarrow (SP) - 1$  ;  $((SP)) \leftarrow (PC)_H$

$(SP) \leftarrow (SP) - 1$  ;  $((SP)) \leftarrow (PC)_L$

$(PC) \leftarrow \text{addr16}$

It is conditional subroutine call instruction. The conditional CALL instruction will check for a flag condition. If the flag condition is true, then the address of the next instruction is pushed to the stack and the call address (address given in the instruction) is loaded in the program counter. Now, the processor will start executing the instructions stored in this address. If the flag condition is false, then the next instruction is executed.

There are eight conditional CALL instructions. These are:

- i) **CZ addr16** ;Call on Zero - Call if zero flag = 1.
- ii) **CNZ addr16** ;Call on Not Zero - Call if zero flag = 0.
- iii) **CC addr16** ;Call on Carry - Call if carry flag = 1.
- iv) **CNC addr16** ;Call on No Carry - Call if carry flag = 0.
- v) **CM addr16** ;Call on Minus - Call if sign flag = 1.
- vi) **CP addr16** ;Call on Positive - Call if sign flag = 0.
- vii) **CPE addr16** ;Call on Parity Even - Call if parity flag = 1.
- viii) **CPO addr16** ;Call on Parity Odd - Call if parity flag = 0.



Three byte instruction	Two or five machine cycles:	<b>Condition False</b>	<b>Condition True</b>
Immediate addressing		Opcode fetch - 6T	Opcode fetch - 6T
		Memory read - 3T	Memory read - 3T
		<u>9T</u>	Memory read - 3T
			Memory write - 3T
			Memory write - 3T
			<u>18T</u>

5. **RET**  $(PC)_L \leftarrow ((SP))$  ;  $(SP) \leftarrow (SP) + 1$   
 $(PC)_H \leftarrow ((SP))$  ;  $(SP) \leftarrow (SP) + 1$

(RET - Return to the main program)

It is an unconditional return instruction. This instruction is placed at the end of the subroutine program, in order to return to the main program. When this instruction is executed, the top of the stack is popped to (loaded in) the program counter.

**Note :** While calling the subroutine using CALL instruction, the return address of the main program is pushed to the stack. The return instruction, (RET) pops that to the program counter. Thus the processor resumes the execution of main program.

One byte instruction	Three machine cycles:	Opcode fetch - 4 T
Register indirect addressing		Memory read - 3 T
		Memory read - 3 T
		<u>10 T</u>

6. **R<condition>**  
 If <condition> is TRUE then,  
 $(PC)_L \leftarrow ((SP))$  ;  $(SP) \leftarrow (SP) + 1$   
 $(PC)_H \leftarrow ((SP))$  ;  $(SP) \leftarrow (SP) + 1$

It is conditional return instruction.

In a conditional return instruction a flag condition is tested. If the flag condition is true, then the program control return to main program by popping the top of the stack to the program counter. If the flag condition is false, then the next instruction is executed.

There are eight conditional return instructions:

- i) **RZ** ;Return on Zero - Return if zero flag = 1.
- ii) **RNZ** ;Return on Not Zero - Return if zero flag = 0.
- iii) **RC** ;Return on Carry - Return if carry flag = 1.
- iv) **RNC** ;Return on No Carry - Return if carry flag = 0.
- v) **RM** ;Return on Minus - Return if sign flag = 1.
- vi) **RP** ;Return on Positive - Return if sign flag = 0.
- vii) **RPE** ;Return on Parity Even - Return if parity flag = 1.
- viii) **RPO** ;Return on Parity Odd - Return if parity flag = 0.

One byte instruction	One or three machine cycles:	<b>Condition False</b>	<b>Condition True</b>
Register indirect addressing		Opcode fetch - 6T	Opcode fetch - 6T
			Memory read - 3T
			Memory read - 3T
			<u>12T</u>

**7. RST n**

*It is a restart instruction. The restart instructions are also called software interrupts. Each restart instruction has a vector address. The vector address is fixed by the manufacturer (INTEL).*

When a restart instruction is executed, the content of the program counter is pushed to the stack and the vector address is loaded in the program counter. The vector address is internally generated (computed) by the processor. The vector address for **RST n** is obtained by multiplying **n** by 8. Thus the program control is branched to a subroutine program stored in this vector address.

One byte instruction  
Register indirect addressing

**Three machine cycles:** Opcode fetch - 6 T  
Memory write - 3 T  
Memory write - 3 T  
12T

*There are eight restart instructions.*

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

The vector addresses for the restart instructions are listed in the table given below :

Restart instruction	Vector address	Computation of vector address
RST 0	0000 <sub>H</sub>	$0 \times 8 = 0_{10} = 0_H$
RST 1	0008 <sub>H</sub>	$1 \times 8 = 8_{10} = 8_H$
RST 2	0010 <sub>H</sub>	$2 \times 8 = 16_{10} = 10_H$
RST 3	0018 <sub>H</sub>	$3 \times 8 = 24_{10} = 18_H$
RST 4	0020 <sub>H</sub>	$4 \times 8 = 32_{10} = 20_H$
RST 5	0028 <sub>H</sub>	$5 \times 8 = 40_{10} = 28_H$
RST 6	0030 <sub>H</sub>	$6 \times 8 = 48_{10} = 30_H$
RST 7	0038 <sub>H</sub>	$7 \times 8 = 56_{10} = 38_H$

8. PCHL (PC) ← (HL)

*The content of the HL register pair is moved to the program counter. Since this instruction alters the content of the program counter, the program control is transferred to a new address. This instruction is used by the system designer to implement the system subroutine to execute a program.*

One byte instruction  
Implied addressing

## 2.11 MACHINE CONTROL INSTRUCTIONS

1. DI

(DI - Disable Interrupts)

*When this instruction is executed, all the interrupts except TRAP are disabled. [When the interrupts are disabled the processor will not accept or recognize the interrupt request made by the external devices through the interrupt pins. When the processor is doing an emergency work, it can execute DI instruction to prevent the interrupts from interrupting the processor.]*

*One byte instruction* **One machine cycle:** Opcode fetch - 4T

## 2. EI

### (EI - Enable Interrupts)

This instruction is used (or executed) to allow the interrupts after disabling. (The interrupts except TRAP are disabled after processor reset or after execution of DI instruction. When we want to allow the interrupts, we have to execute EI instructions.)

One byte instruction

**One machine cycle:** Opcode fetch - 4T

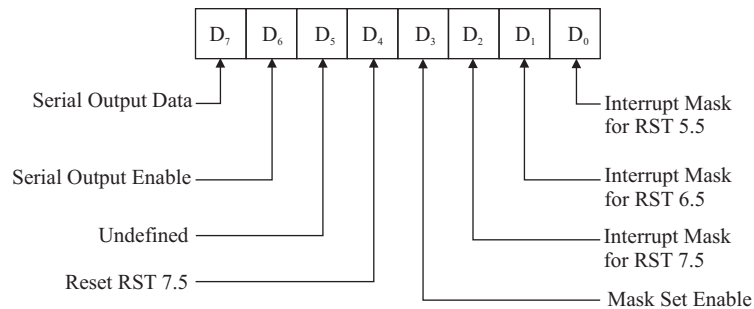
## 3. SIM

### (SIM - Set Interrupt Mask)

The SIM instruction is used to mask the hardware interrupts RST 7.5, RST 6.5 and RST 5.5. It is also used to send data through the SOD line. (SOD: Serial Output Data pin of the 8085 processor.) The execution of SIM instruction uses the content of the accumulator to perform the following functions:

- Program the interrupt mask for the hardware interrupts RST 5.5, RST 6.5 and RST 7.5.
- Reset the edge-triggered RST 7.5 input latch.
- Load the SOD output latch.

The bits in the accumulator before execution of the SIM instruction are defined as shown in the Fig. 2.12.



**Fig. 2.12 :** Accumulator content before execution of SIM instruction.

If the mask set enable bit is set to "1" then the interrupt mask bits for RST 7.5, RST 6.5 and RST 5.5 ( $D_0$ ,  $D_1$  and  $D_2$ ) are recognized and if it is "0" then these bits are not recognized by the processor. The interrupt mask bits  $D_0$ ,  $D_1$  and  $D_2$  can be independently set to "1" to mask the particular interrupt and reset to "0" to unmask the particular interrupt.

If the bit  $D_4$  is set to "1", then an internal flip-flop is reset to "0" in order to disable the RST 7.5 interrupt. If the serial output enable is "1", the serial output data is sent to the SOD pin.

One byte Instruction

**One machine cycle:** Opcode fetch - 4T

## 4. RIM

### (RIM - Read Interrupt Mask)

The RIM instruction is used to check whether an interrupt is masked or not. It is also used to read data from the SID line. (SID: Serial Input Data pin of 8085 processor).

When a RIM instruction is executed, the accumulator is loaded with 8-bit data. The 8-bit data in the accumulator (content of accumulator) can be interpreted as shown in Fig. 2.13.

Bits  $D_0$ ,  $D_1$  and  $D_2$  provide the mask status of the RST 5.5, RST 6.5 and RST 7.5 interrupts respectively. If the mask bit corresponding to a particular RST is "1", then the interrupt is masked and if the mask bit is "0" then the interrupt is unmasked.

If the interrupt enable bit ( $D_3$ ) is "0", the 8085's maskable interrupts are disabled. The interrupts are enabled if this bit is "1".

A "1" in a particular interrupt pending bit indicates that an interrupt is being requested on the identified RST line. When this bit is "0", no interrupt is waiting to be serviced. The serial input data (bit  $D_7$ ) indicate the value of the signal at the SID pin.

One byte instruction

**One machine cycle:** Opcode fetch - 4T

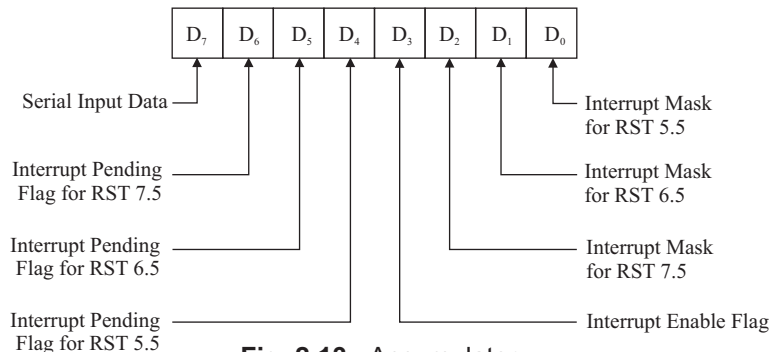


Fig. 2.13 : Accumulator.

**5. HLT**  
(HLT - Halt program Execution)

This instruction is placed at the end of the program. When this instruction is executed, the processor suspends program execution and bus will be in idle state.

One byte instruction

Two machine cycle: Opcode fetch -  $3T$   
Bus idle -  $2T$   
 $5T$

**6. NOP**  
(NOP - No operation)

The NOP is a dummy instruction, it neither achieves any result nor affects any CPU registers. This is an useful instruction for producing software delay and reserve memory spaces for future software modifications.

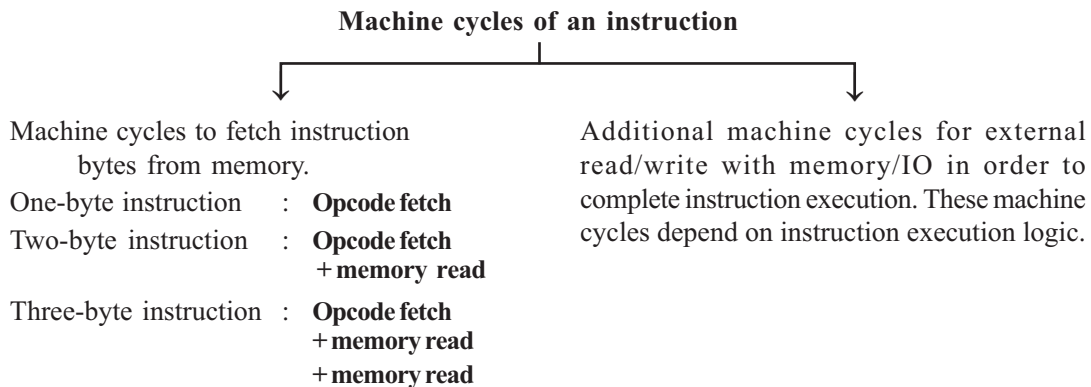
One byte instruction

One machine cycle : Opcode fetch -  $4T$

## 2.12 TIMING DIAGRAM OF 8085 INSTRUCTIONS

The 8085 instructions is one to five machine cycles. (Refer Table-2.1 for the machine cycles of instructions.) Actually the execution of an instruction is the execution of the machine cycles of that instruction in a predefined order. Therefore, from the knowledge of the timing diagrams of machine cycles, the timing diagram of an instruction can be obtained.

The machine cycles of an 8085 instruction can be divided into two parts as shown below:



Based on the execution of the machine cycles, the instructions can be classified as shown below:

- Case(i)* : 1-byte, 1-cycle - Opcode fetch.
- Case(ii)* : 1-byte, 2-cycle - Opcode fetch + memory read/write.
- Case(iii)* : 1-byte, 3-cycle - Opcode fetch + memory read/write (or Bus idle)  
+ memory read/write (or Bus idle).
- Case(iv)* : 1-byte, 5-cycle - Opcode fetch + memory read + memory read  
+ memory write + memory write.
- Case(v)* : 2-byte, 2-cycle - Opcode fetch + memory read (read second byte of instruction.)
- Case(vi)* : 2-byte, 3-cycle - Opcode fetch + memory read (read second byte of instruction)  
+ memory read/write/or IO read/write.
- Case(vii)* : 3-byte, 3-cycle - Opcode fetch + memory read (read second byte of instruction)  
+ memory read (read third byte of instruction.)
- Case(viii)* : 3-byte, 4-cycle - Opcode fetch + memory read (read second byte of instruction)  
+ memory read (read third byte of instruction)  
+ memory read/write.
- Case(ix)* : 3-byte, 5-cycle - Opcode fetch + memory read (read second byte of instruction)  
+ memory read (read third byte of instruction)  
+ memory read/write + memory read/write.

The timing diagram of an instruction is obtained by drawing the timing diagrams of the machine cycles of that instruction one by one in the order of execution. The timing diagrams of few instructions are presented from Figs. 2.14 to 2.20.

### **Timing Diagram of STA Instruction**

The "STA addr16" instruction is used to store the content of the accumulator to a memory location. This instruction employs direct addressing. Let the content of the accumulator be  $C7_H$  and it is desired to store the content of the accumulator to a memory location  $526A_H$ .

The STA addr16 instruction is a three byte instruction. The first byte is the opcode of the instruction  $32_H$ . The second byte is low byte address  $6A_H$  and the third byte is high byte address  $52_H$ . Let the three bytes of the instructions be stored in memory locations  $41FF_H$ ,  $4200_H$  and  $4201_H$ .

In order to execute this instruction, the 8085 microprocessor will first execute opcode fetch machine cycle to get the opcode, followed by two memory read cycles to read the address of data (i.e., to read second and third byte of instruction). Then, the processor executes the memory write cycle to store the content of the accumulator in the memory. The status of various signals during execution of this instruction are shown in Fig. 2.14. (Readers are advised to refer to Section 2.2 for explanation of each machine cycle.)

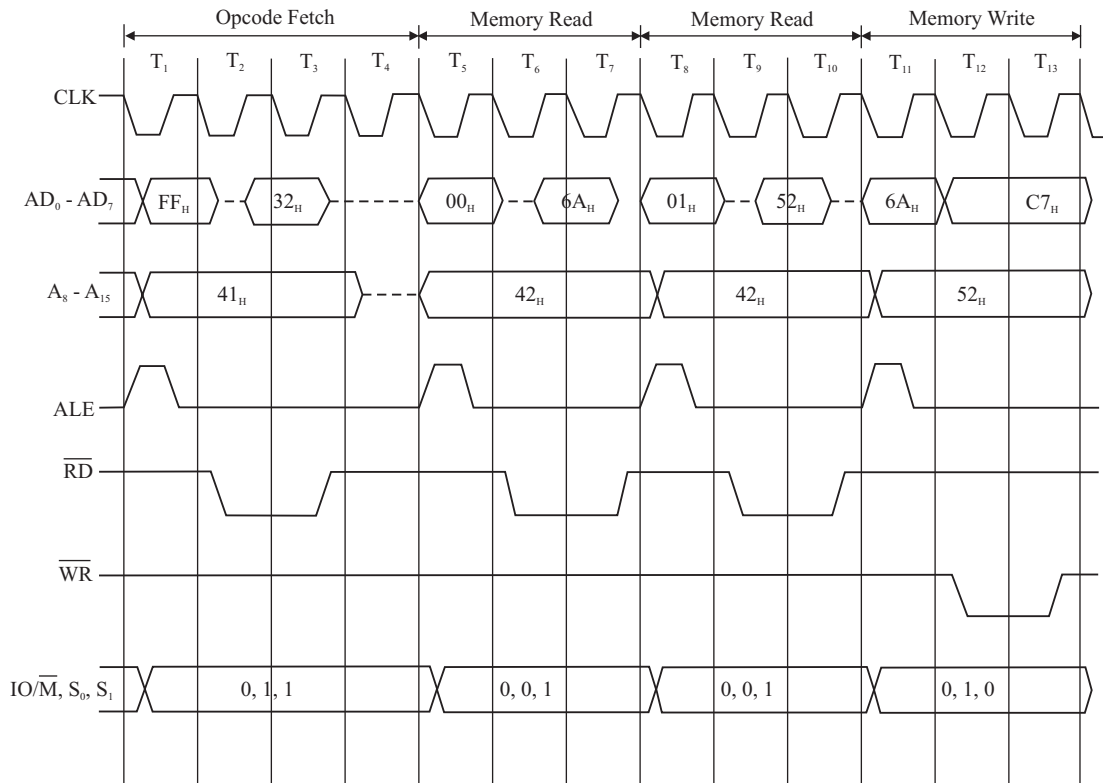


Fig. 2.14 : Timing diagram of STA 526AH instruction.

### Timing Diagram of PUSH Instruction

The "PUSH rp" instruction is used to store the content of a register pair in the stack memory. This instruction employs register indirect addressing using Stack Pointer (SP). Let us consider PUSH B instruction. On execution of this instruction, the content of the BC pair is pushed to the stack. Let the content of the BC pair be  $E25D_H$  and the content of SP be  $A100_H$ .

The PUSH rp is one-byte instruction and it is the opcode of the instruction. The opcode of PUSH B instruction is  $C5_H$  and let it be stored in memory location  $C010_H$ . In order to execute this instruction, the processor will first execute the opcode fetch cycle to get the opcode  $C5_H$ . Then the processor executes two memory write cycles to store the content of the BC pair in the stack memory. The status of various signals during execution of this instruction are shown in Fig. 2.15.

During the memory write cycles in PUSH rp instruction, the content of the SP is used as the memory address. In the first write cycle, the content of the SP is decremented by one ( $A100_H - 1 = A0FF_H$ ) and output on the address lines and in this address, the content of B-register ( $E2_H$ ) is stored. In the second write cycle, the content of the SP is again decremented by one ( $A0FF_H - 1 = A0FE_H$ ) and output on the address lines and in this address, the content of C-register ( $5D_H$ ) is stored.

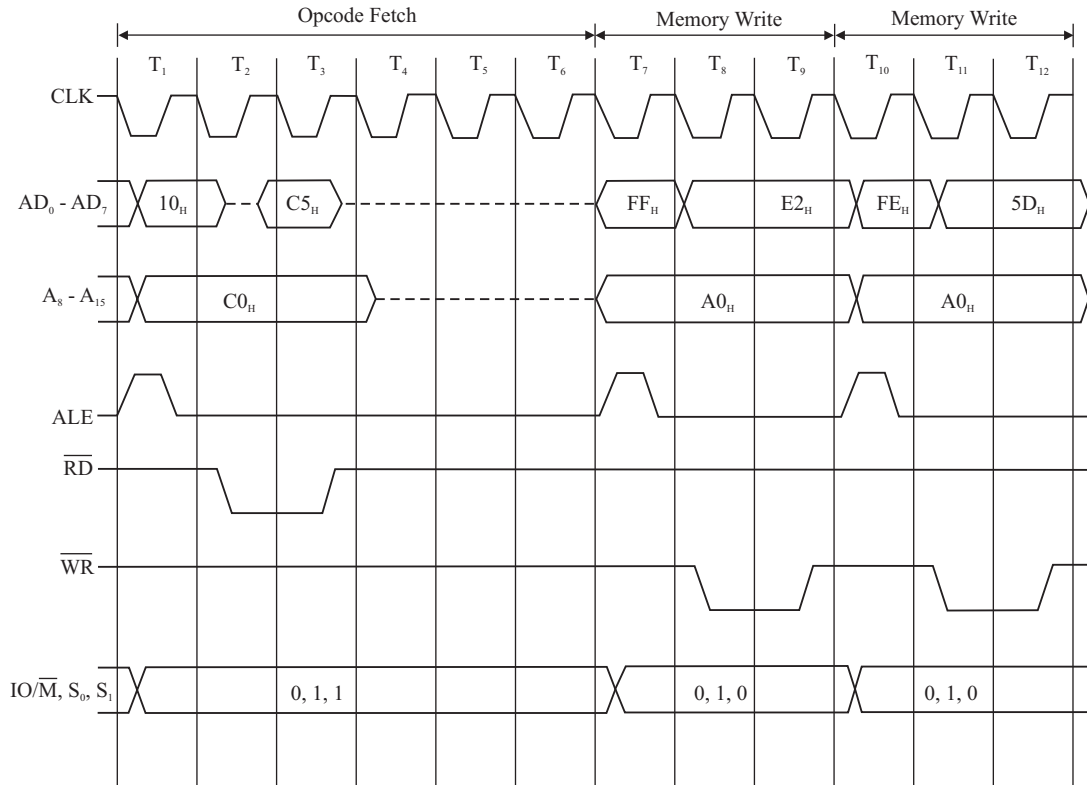


Fig. 2.15 : Timing diagram of PUSH B instruction.

### Timing Diagram of IN Instruction

The "IN addr8" instruction is used to read the content of an IO-mapped device/port and store in the accumulator. For addressing IO-mapped devices the 8085 microprocessor employs 8-bit address. Let the 8-bit address of the IO port be C0<sub>H</sub>, and the content of IO port be 5E<sub>H</sub>.

The IN addr8 instruction, is a two-byte instruction. The first byte is the opcode of the instruction DB<sub>H</sub> and the second byte is the IO port address C0<sub>H</sub>. Let the two bytes of the instruction be stored in memory locations 4125<sub>H</sub> and 4126<sub>H</sub>.

In order to execute this instruction, the 8085 microprocessor will first execute the opcode fetch machine cycle to get the opcode, followed by the memory read cycle to read the IO port address (i.e., to read the second byte of the instruction.) Then, the processor executes IO read cycle to read the content of the IO port. The status of various signals during execution of this instruction are shown in Fig. 2.16.

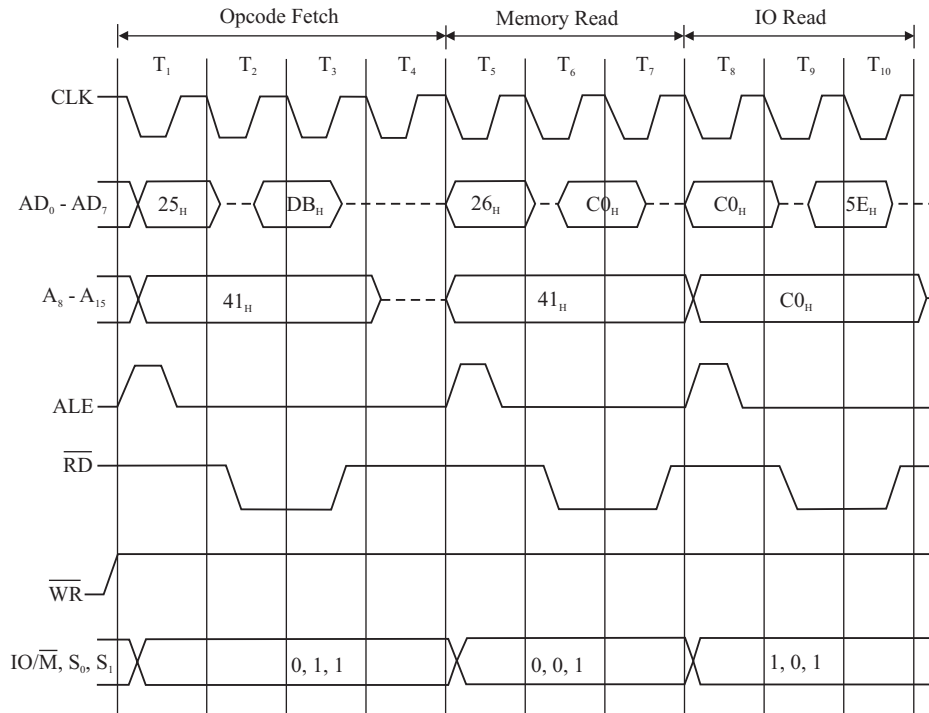


Fig. 2.16 : Timing diagram of IN C0H instruction.

### Timing Diagram of OUT Instruction

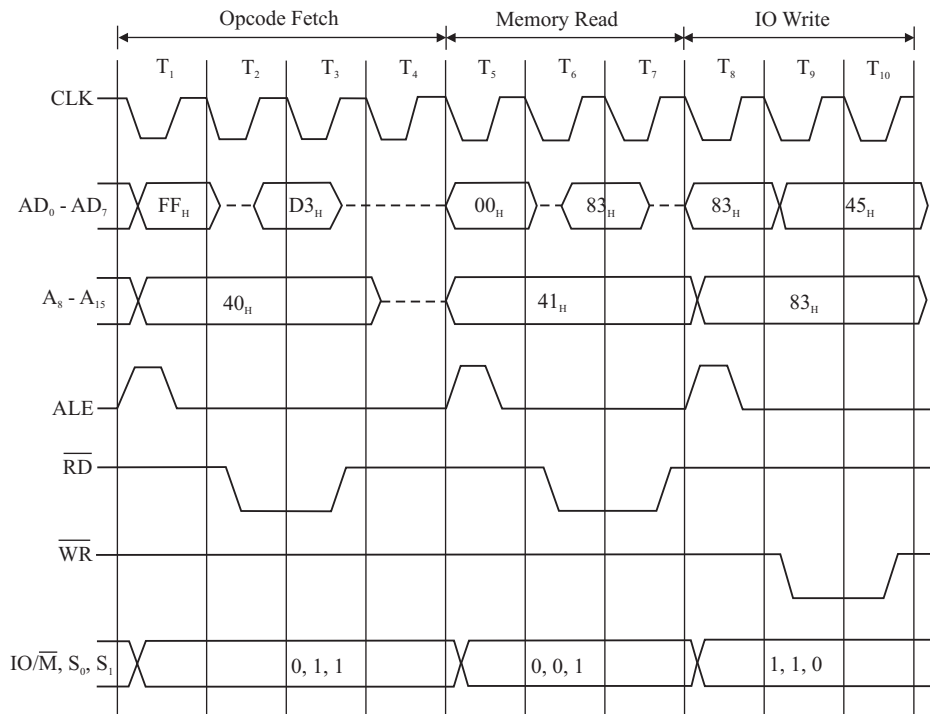


Fig. 2.17 : Timing diagram of OUT 83H instruction.



The "OUT addr8" instruction is used to output the content of the accumulator to the IO-mapped device/port. For addressing IO-mapped devices, the 8085 microprocessor employs 8-bit address. Let the 8-bit address of the IO port be  $83_H$  and the content of the accumulator be  $45_H$ .

The OUT addr8 instruction is a two-byte instruction. The first byte is the opcode of the instruction  $D3_H$ , and the second byte is the IO port address  $83_H$ . Let the two bytes of instruction be stored in memory locations  $40FF_H$  and  $4100_H$ .

In order to execute this instruction, the 8085 microprocessor will first execute the opcode fetch machine cycle to get the opcode, followed by the memory read cycle to read the IO port address. (i.e., to read the second byte of the instruction.) Then the processor executes the IO write cycle to write the content of the accumulator to the IO port. The status of various signals during execution of this instruction are shown in Fig. 2.17.

### Timing Diagram of INR M Instruction

The "INR M" instruction is used to increment the content of a memory location. This instruction employs register indirect addressing using an HL pair. Let the content of the HL pair be  $4250_H$  and let the content of the memory location  $4250_H$  be  $12_H$ .

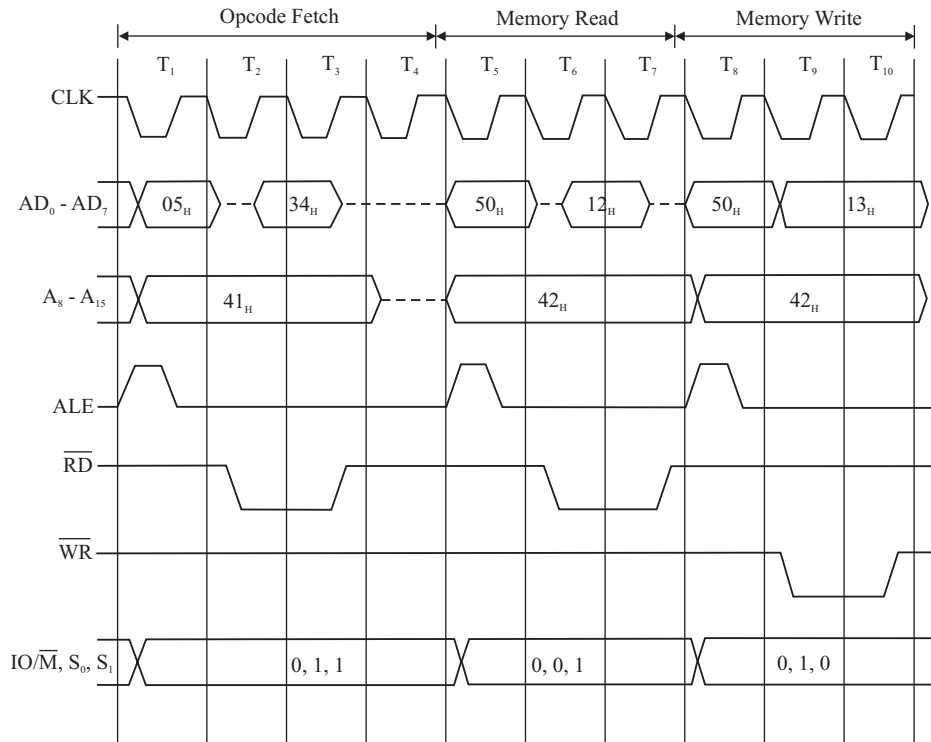


Fig. 2.18 : Timing diagram of INR M instruction.

The INR M is one-byte instruction and it is the opcode of instruction  $34_H$ . Let this instruction be stored in the memory location  $4105_H$ . In order to execute the instruction, the 8085 microprocessor will first execute the opcode fetch cycle to get the opcode  $34_H$ . Then, it executes the memory read cycle to read the content of the memory location  $4250_H$ .

The content ( $12_H$ ) of the memory location is incremented by one in the ALU and then, the processor executes the memory write cycle to store the result ( $13_H$ ) of the ALU operation in the same memory location  $4250_H$ . The status of various signals during execution of this instruction are shown in Fig. 2.18.

### **Timing Diagram of CALL Instruction**

The "CALL addr16" instruction is used to execute a subroutine/procedure stored at addr16, after saving the address of the next instruction in the stack memory. On execution of this instruction, the addr16 is loaded in the Program Counter (PC) and the previous value of the PC is stored in the stack memory pointed by the Stack Pointer (SP). Let the address of the subroutine be  $4F50_H$  and the content of SP be  $4100_H$ .

The CALL addr16 is a three-byte instruction. The first byte is the opcode of the instruction  $CD_H$ . The second byte is the low byte of address  $50_H$  and the third byte is the high byte of address  $4F_H$ . Let the three bytes of the instructions be stored in memory locations  $4200_H$ ,  $4201_H$  and  $4202_H$ .

In order to execute this instruction, the 8085 microprocessor will first execute the opcode fetch machine cycle to get the opcode of the instruction  $CD_H$ , followed by two memory read cycles to get the address of the subroutine (i.e., to read the second and third byte of instruction).

At the end of the second memory read cycle, the content of the PC will be  $4203_H$ . After the read cycles, the processor executes two memory write cycles to store this content ( $4203_H$ ) of PC in the stack. During the memory write cycles, the content of the SP is used as the memory address. In the first write cycle, the content of the SP is decremented by one ( $4100_H - 1 = 40FF_H$ ) and output on the address lines and in this address, the high byte of PC ( $42_H$ ) is stored. In the second write cycle, the content of the SP is again decremented by one ( $40FF_H - 1 = 40FE_H$ ) and output on the address lines and in this address the low byte of PC ( $03_H$ ) is stored. The status of various signals during execution of this instruction are shown in Fig. 2.19.

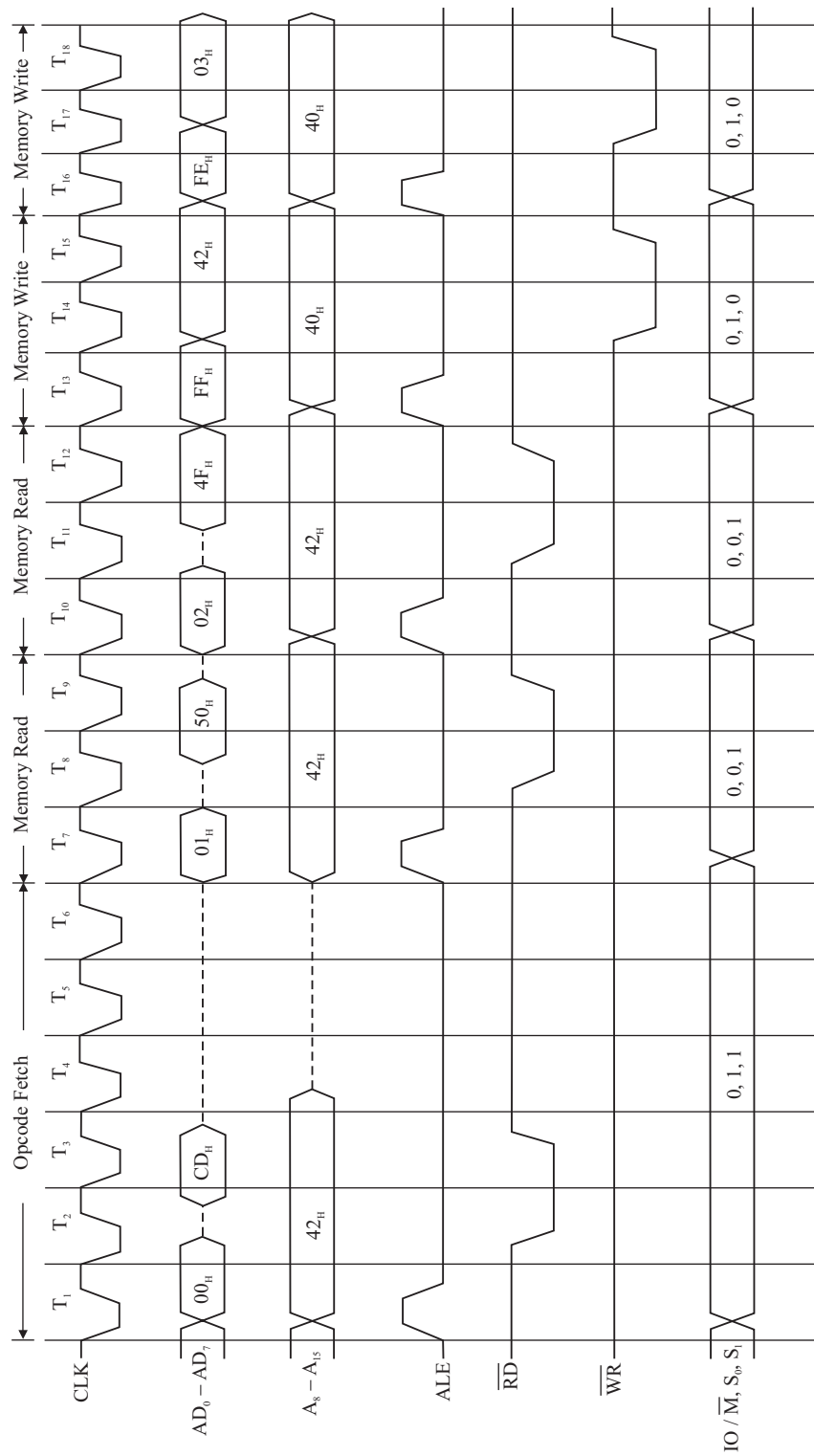


Fig. 2.19 : Timing diagram of CALL 4F50H instruction.

### Timing Diagram of RET Instruction

In a program while calling a subroutine/procedure using CALL instruction, the address of next instruction of the program is stored in top of stack. The RET instruction is usually placed at the end of subroutine/procedure. On execution of RET instruction, the top of stack is loaded in Program Counter (PC). For stack operation, the content of Stack Pointer (SP) is used as address. Let the content of SP be  $40FE_H$  and the content of stack memory locations  $40FE_H$  and  $40FF_H$  be  $03_H$  and  $42_H$ , respectively.

The RET instruction is one-byte instruction and it is the opcode of instruction  $C9_H$ . Let this instruction be stored in memory location  $4F80_H$ . In order to execute this instruction, the 8085 processor will first execute the opcode fetch machine cycle to get the opcode  $C9_H$ . Then it executes two memory read cycle to read the top of stack memory.

During the memory read cycles the content of SP is used as memory address. In the first read cycle the content of SP ( $40FE_H$ ) is output on address lines and the memory content ( $03_H$ ) in this address is read and stored as low byte of PC. In the second read cycle the content of SP is incremented by one ( $40FE_H + 1 = 40FF_H$ ) and output on address lines and the memory content ( $42_H$ ) in this address is read and stored as high byte of PC. The status of various signals during execution of this instruction are shown in Fig. 2.20.

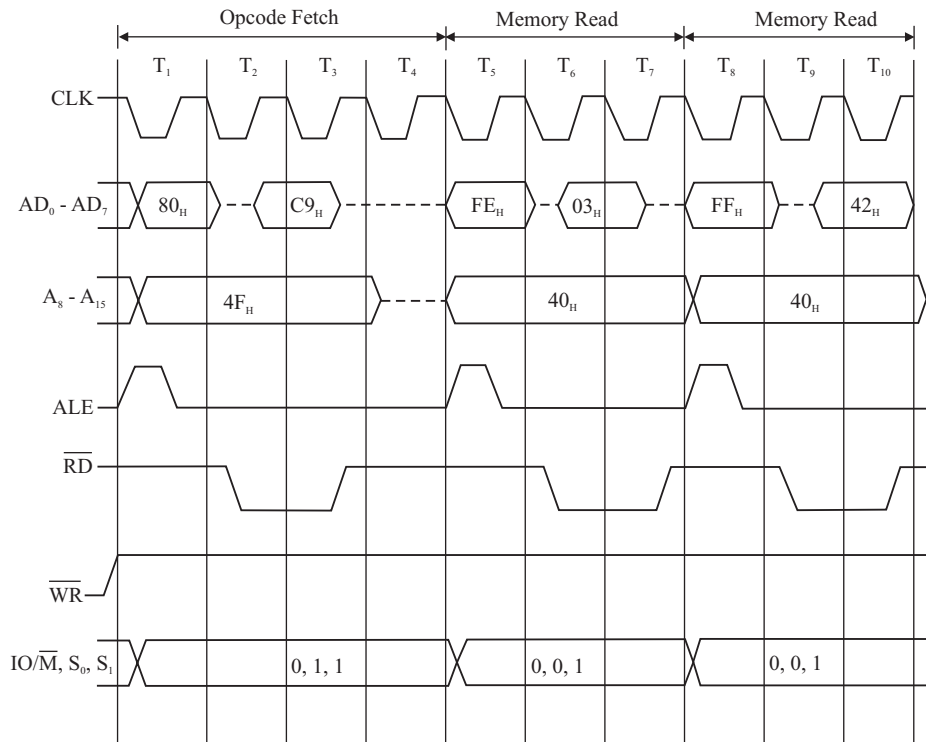


Fig. 2.20 : Timing diagram of RET instruction.

## 2.13 SUMMARY

---

- The software developed using 1s and 0s is called machine language program.
- The machine can understand only machine language programs.
- The software developed using mnemonics is called assembly language program.
- Assembler is a conversion software, which can convert assembly language programs to machine language programs.
- The machine language and assembly language programs are machine (processor) dependent.
- The language which can be used to develop software independent of the hardware (processor) in the machine are called High Level Languages.
- The compiler or interpreter is a software which can convert high level language programs to machine language programs.
- The sequence of operations that a processor has to carry out while executing the instruction is called Instruction cycle.
- The basic operations performed by the processor are called machine cycles or processor cycles.
- The 8085 microprocessor has seven basic machine cycles.
- In 8085, one T-state is equal to the time period of the internal clock signal of the processor.
- Each instruction of the 8085 processor consists of one to five machine cycles.
- The timing diagram provides the status of various signals when a machine cycle is executed.
- The first T-state of every machine cycle in 8085 processor is used to demultiplex the low order address and data lines.
- The 8085 processor checks for an interrupt at the second T-state of the last machine cycle of every instruction.
- In 8085, wait states can be introduced in any machine cycle except bus idle cycle between  $T_2$  and  $T_3$ .
- The 8085 has 74 basic instructions and 246 total instructions.
- The method of specifying the data to be operated by the instruction is called addressing.
- In 8085, the flag register and the accumulator together are called PSW (Program Status Word).
- The flags are altered after execution of arithmetic and logical instructions.
- The 8-bit increment and decrement instructions does not affect carry flag.
- The 16-bit increment/decrement instructions will not affect any flag.
- The data transfer, branching and machine control instructions will not alter the flags.
- The memory address residing in SP (Stack Pointer) indicates the top of stack.

## 2.14 SHORT QUESTIONS AND ANSWERS

---

### 2.1 What is Software and Hardware?

The software is a set of instructions or commands needed for performing a specific task by a programmable device or a computing machine.

The hardware refers to the components or devices used to form computing machine in which the software can be run and tested. Without software the hardware is an idle machine.

2.2 *What is machine language?*

The language that can be understood by a programmable machine is called machine language. The machine language program are developed using 1s and 0 s.

2.3 *What is assembly language?*

The language in which the mnemonics (short-hand form of instructions) are used to write a program is called assembly language. The mnemonics are given by the manufacturers of microprocessor.

2.4 *What are machine language and assembly language programs?*

The software developed using 1s and 0 s are called machine language programs. The software developed using mnemonics are called assembly language programs.

2.5 *What is the drawback in machine language and assembly language programs?*

The machine language and assembly language programs are machine dependent. The programs developed using these languages for a particular machine cannot be directly run on another machine. (But after conversion using suitable conversion software it can be run on another machine.)

2.6 *Define mnemonics.*

The short-hand form of describing the instructions are called mnemonics. The mnemonics are given by the manufacturers of microprocessors and programmable devices.

2.7 *What is processor cycle (machine cycle)?*

The processor cycle or machine cycle is the basic external operation performed by the processor. To execute an instruction, the processor will run one or more machine cycles in a particular order.

2.8 *What is instruction cycle?*

The sequence of operations that a processor has to carry out while executing an instruction is called instruction cycle. Each instruction cycle of a processor in turn consists of a number of machine cycles.

2.9 *What is fetch and execute cycle?*

In general, the instruction cycle of an instruction can be divided into fetch and execute cycles. The fetch cycle is executed to fetch the opcode from memory. The execute cycle is executed to decode the instruction and to perform the work instructed by the instruction.

2.10 *List the various machine cycles of 8085.*

The various machine cycles of 8085 are as follows:

- (i) Opcode fetch cycle
- (ii) Memory read cycle
- (iii) Memory write cycle
- (iv) IO read cycle
- (v) IO write cycle
- (vi) Interrupt acknowledge cycle
- (vii) Bus idle cycle.

2.11 *What is the need for timing diagram?*

The timing diagram provides information regarding the status of various signals, when a machine cycle is executed. The knowledge of timing diagram is essential for system designer to select matched peripheral devices like memories, latches, ports, etc., to form a microprocessor system.

2.12 *What is T-state?*

The T-state is the time period of the internal clock signal of the processor. The time taken by the processor to execute a machine cycle is expressed in T-state.

2.13 *How many machine cycles constitute one instruction cycle in 8085?*

Each instruction of the 8085 processor consist of one to five machine cycles.

2.14 *Define opcode and operand.*

Opcode (**Operation Code**) is the part of an instruction/directive that identifies a specific operation. Operand is a part of an instruction/directive that represents a value on which the instruction acts.

2.15 *What is opcode fetch cycle?*

The opcode fetch cycle is a machine cycle executed to fetch the opcode of an instruction stored in memory. The first machine cycle of every instruction is opcode fetch machine cycle.

2.16 *What operation is performed during first T-state of every machine cycle in 8085?*

In 8085, during the first T-state of every machine cycle the low byte address is latched into an external latch using ALE signal.

2.17 *Why status signals are provided in microprocessor?*

The status signals can be used by the system designer to track the internal operations of the processor. Also, it can be used for memory expansion (by providing separate memory banks for program and data, and selecting the banks using status signals).

2.18 *How the 8085 processor differentiates memory access (read/write) and IO access (read/write)?*

The memory access and IO access is differentiated using IO/M signal. The 8085 processor asserts IO/M **low** for memory read/write operation and IO/M is asserted **high** for IO read/write operation.

2.19 *In which lines the 8085 processor gives the output of IO port address during IO read/write operation?*

When the processor executes an IO read or write cycle, 8-bit port address is sent out both on low order address bus and high order address bus. This facility offers a flexibility for system designer to use either low-order address lines or high-order address lines for addressing ports and generating chip select signals for IO devices.

2.20 *When the 8085 processor checks for an interrupt?*

In the second T-state of the last machine cycle of every instruction, the 8085 processor checks whether an interrupt request is made or not.

2.21 *What is interrupt acknowledge cycle?*

The interrupt acknowledge cycle is a machine cycle executed by 8085 processor after acceptance of the interrupt to get the address of the interrupt service routine in-order to service the interrupting device.

2.22 *What will be the status of the processor during bus idle cycle?*

During bus idle cycle, the status signals  $S_0$  and  $S_1$  are both asserted **low** and data, address and control pins are driven to **high impedance** state. Also, the processor will not sample the READY signal.

2.23 *How the slow peripherals are interfaced with 8085 processor?*

The slow peripherals require longer read/write time than allowed by the processor. Hence to interface slow peripherals, an extra hardware should be designed so that it introduces required number of wait states in machine cycles between  $T_1$  and  $T_2$ . An alternate solution is to interface the slow peripherals using ports.

## 2.24 When is the READY signal sampled by the processor?

The 8085 processor samples or checks the READY signal at the second T-state of every machine cycle.

## 2.25 What are wait states?

The T states introduced between  $T_2$  and  $T_3$  of a machine cycle by the slow peripherals (to get extra time for read/write operation) are called wait states.

## 2.26 When the 8085 processor will enter wait state?

The 8085 processor will check the READY signal at the second T-state of a machine cycle. If the READY is tied **low** at this time, then it will enter into wait state (i.e., after second T-state). The processor will come out of wait state only when READY is again made **high**.

## 2.27 What is the difference between wait state and bus idle condition?

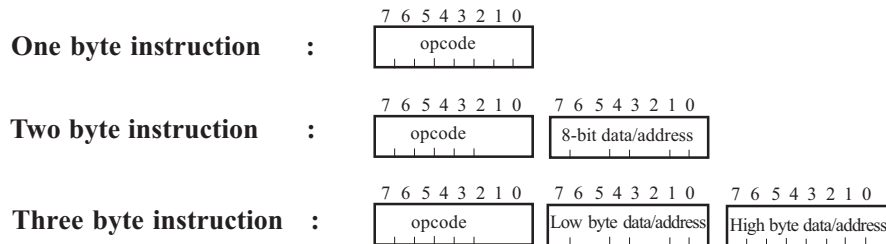
During bus idle condition, the tristate pins of the processor are driven to **high impedance** state, but during wait state they are in normal states (either **low** or **high**). The READY is not sampled during bus idle condition but it is sampled during wait state.

## 2.28 How many instructions are available in 8085 instruction set?

The 8085 instruction set consists of 74 basic instructions and 246 total instructions.

## 2.29 What is the instruction format of 8085?

The size of 8085 instruction is 1 to 3 bytes. Each instruction has one-byte opcode. The remaining bytes are either data or address. The format of 8085 instructions are shown below :



## 2.30 What is addressing?

The method of specifying the data to be operated (operand) by the instruction is called addressing.

## 2.31 What are the addressing modes available in 8085?

The 8085 has the following five different modes of addressing.

- i) Immediate addressing
- ii) Direct addressing
- iii) Register addressing
- iv) Register indirect addressing
- v) Implied addressing.

## 2.32 Explain the immediate addressing with an example.

In immediate addressing mode, the data is specified in the instruction itself. The data will be a part of the program instruction.

**Example :** MVI B, 3EH - Move the data 3EH given in the instruction to B-register.



2.33 What is direct addressing? Give an example.

If the address of the data is directly specified in the instruction then the addressing mode is called direct addressing.

**Example :** LDA 1050H - Load the data available in memory location 1050<sub>H</sub> in accumulator.

2.34 Explain register addressing with an example.

In register addressing mode, the instruction specifies the name of the register in which the data is available.

**Example :** MOV A, B - Move the content of B-register to A-register.

2.35 Explain register indirect addressing with an example.

In register indirect addressing mode, the instruction specifies the name of the register in which the address of the data is available. Here the data will be in memory and the address will be in the register pair.

**Example :** MOV A, M - The content of memory whose address is available in HL pair is moved to A-register.

2.36 What is implied or implicit addressing mode?

If the instruction operates on a data available in the register defined by the opcode then the addressing mode is called implied or implicit addressing mode.

**Example :** CMA - Complement the content of accumulator.

2.37 What are the functions performed by data transfer instruction? Give an example and explain.

The data transfer instructions can copy the content of one register to another and copy the content of register to memory or vice versa.

**Example :** MOV B, C - The content of C-register is moved (copied) to B-register

2.38 What are the functions performed by arithmetic instructions? Give an example and explain.

The functions performed by arithmetic instructions are Addition, Subtraction, Increment and Decrement.

**Example :** ADD E - The content of E-register is added to accumulator.

2.39 What are the operations performed by logical instructions? Give an example and explain.

The operations performed by logical instructions are AND, OR, EXCLUSIVE-OR, Complement, Compare and Shift (Rotate).

**Example :** ANA D - The content of D-register is logically ANDed with accumulator.

2.40 In which unit the arithmetic and logical operations are performed. Which unit is the destination of result.

The arithmetic and logical operations are performed in ALU. After the operation, the result will be stored in accumulator.

2.41 Which group of instruction affects the flags?

The flags are altered after execution of arithmetic and logical instructions.

2.42 What are the arithmetic instructions that do not affect the flag?

The 16-bit increment and decrement instructions (INX rp and DCX rp) will not affect any flags.

2.43 What are the flags affected by 8-bit increment and decrement instructions?

Except carry, all other flags are affected by 8-bit increment and decrement instructions.

2.44 What will be condition of flags after logical AND and OR operations?

After logical AND operation the carry flag is RESET (0), auxiliary carry flag is SET (1) and depending on the result of AND operation other flags are altered.

After logical OR operation the carry flag and auxiliary carry flag are RESET (0). Depending on the result of OR operation other flags are altered.

2.45 List the instructions that affect only carry flag.

The instructions that affect only carry flag are the following :

CMC	RAR	STC
DAD rp	RLC	
RAL	RRC	

2.46 What is DAA ?

DAA - **D**ecimal **A**djust **A**ccumulator.

After BCD addition, this instruction is executed to get the result in BCD. When DAA instruction is executed, the content of the accumulator is altered or adjusted as explained below :

- (i) If the sum of lower nibbles exceeds 09<sub>h</sub> or auxiliary carry is set, a correction 06<sub>h</sub> (0110) is added to lower nibble.
- (ii) If the sum of upper nibbles exceeds 09<sub>h</sub> or carry is set, a correction 06<sub>h</sub> (0110) is added to upper nibble.

2.47 What is DAD and what are the flags affected by this instruction?

DAD refers to Double Addition. This instruction is used to perform addition of two 16-bit data.

*Syntax : DAD rp*

The content of register pair (rp) is added to the content of HL pair. After addition, the result will be in HL pair. The register pair can be BC, DE, HL, or SP. On execution of this instruction, only carry flag is affected.

2.48 List the various instructions that can be used to clear accumulator ?

The accumulator can be cleared by the following instructions:

1. MVI A,00<sub>h</sub>
2. SUB A
3. ANI 00<sub>h</sub>
4. XRA A.

2.49 What is the similarity and difference between subtract and compare instruction?

**Similarity** : Both the subtraction and comparison are performed by subtracting two data in ALU and flags are altered depending upon the result.

**Difference** : After subtract instruction is executed, the result is stored in accumulator, but after the execution of compare instruction the result is discarded ( i.e., the subtract instruction alters the content of destination register (accumulator), but the compare instruction will not alter the content of any register or memory).

2.50 List the IO instruction in 8085.

- The IO instruction of 8085 are IN addr8 and OUT addr8.
- The IN instruction is used to input a data byte from the IO-mapped device or port. The OUT instruction used to output data byte to IO-mapped device or port.

2.51 State the difference between LDA and LDAX.

The LDA instruction uses direct addressing mode to load a data byte from memory to accumulator, but LDAX instruction uses register indirect addressing for the same operation.

In LDA instruction, the content of memory location whose address is given in the instruction is moved to accumulator.

In LDAX instruction, a register pair contains the address of memory location. The content of memory location whose address is available in register pair is moved to accumulator.

2.52 *Explain DI and EI.*

**DI - Disable Interrupt** . When this instruction is executed all the interrupts except TRAP are disabled. When the interrupts are disabled the processor will not accept or recognize the interrupt.

**EI - Enable Interrupt**. This instruction is used or executed to allow the interrupts after disabling.

2.53 *What is the function performed by SIM instruction?*

**SIM - Set Interrupt Mask**. The SIM instruction is used to mask the hardware interrupts RST 7.5, RST 6.5 and RST 5.5. The execution of SIM instruction output and the content of the accumulator to program interrupt mask bits are also used to output serial data on the SOD line.

2.54 *What is the function performed by RIM instruction?*

**RIM - Read Interrupt Mask**. The RIM instruction is used to check whether an interrupt is masked or not. It is also used to read data from SID line.

2.55 *What will be the state of the processor after executing HLT instruction?*

When the HLT instruction is executed, the processor suspends program execution and the bus will be in idle state (i.e., the processor keeps on executing bus idle cycle until a reset or interrupt).

2.56 *What is NOP? State its importance.*

The NOP is a dummy instruction, it neither achieves any result nor affects any CPU register. This is used for producing software delay and reserve memory spaces for future software modifications.

2.57 *What is PSW?*

**PSW - Program Status Word**. The flag register and accumulator together is called PSW. Flag register is low order register. Accumulator is high order register.

2.58 *Explain RET instruction.*

**RET - Return to main program**. This instruction is placed at the end of subroutine program in order to return to the main program. When this instruction is executed, the top of stack is popped to program counter.

2.59 *Explain the difference between the conditional and unconditional return instructions.*

In a conditional return instruction a flag condition is tested. If the flag condition is true, then the program control returns to main program. If the flag condition is false, then the next instruction is executed. In unconditional return instruction, the program control returns to the main program irrespective of the condition of the flag.

2.60 *State the difference between STA and STAX instructions.*

The STA instruction uses direct addressing mode to store the content of accumulator to a memory location, but the STAX instruction uses indirect addressing mode for the same operation.

2.61 *What will be the content of SP (Stack Pointer) after execution of PUSH and POP instructions?*

- After execution of PUSH instruction, the content of **Stack Pointer (SP)** will be 02 less than the earlier value.
- After execution of POP instruction, the content of **Stack Pointer (SP)** will be 02 greater than the earlier value.

2.62 *What is the difference between ADD and ADC instruction?*

The ADD instruction will not consider the value of carry flag for addition, but the ADC instruction will consider the value of carry flag (before executing this instruction) for addition. In ADC instruction the content of register or memory and the carry flag are added to the content of accumulator.

2.63 *How is the subtraction performed in 8085?*

The 8085 processor performs 2's complement subtraction and after subtraction, it complements the carry flag.

2.64 *How the result of subtract operation can be interpreted?*

- After subtract operation, if the carry flag is SET (1), then the result is negative will be in 2's complement form.
- After subtract operation, if the carry flag is RESET (0), then the result is positive.

2.65 *What is the difference in 2's complement subtraction and 8085 subtraction?*

In 2's complement subtraction, the result is positive if carry is equal to one (1) and negative if carry is equal to zero (0). But in 8085, the result is negative if carry is equal to one (1) and positive if carry is equal to (0).

2.66 *What is the difference between CALL and JUMP instruction?*

In CALL instruction, the address of next instruction is pushed to stack (i.e., stored in stack memory) before transferring the program control to call address. But in JUMP instruction, the address of next instruction is not saved.

2.67 *What is the difference between conditional and unconditional branch instructions?*

In unconditional branch instructions, the program control is transferred to branch address without checking any flag condition. But in conditional branch instructions, a flag condition is checked and only if the flag condition is true, program control is transferred to branch address, otherwise the next instruction is executed.

---

## CHAPTER 3

---

# MEMORY AND IO INTERFACING

---

## 3.1 INTRODUCTION TO MEMORY

---

A memory unit is an integral part of any microcomputer system and its primary purpose is to store programs and data. In a broad sense, a microcomputer memory system can be logically divided into three groups. They are as follows:

- Processor memory
- Primary or main memory
- Secondary memory

Processor memory refers to registers inside the microprocessor. These registers are used to hold data and results temporarily when computation is in progress. Since the registers of the processor are fabricated using the same technology as that of a microprocessor, there is no speed disparity between these registers and a microprocessor. However, the cost involved in this approach forces a manufacturer to include only a few registers in the microprocessor.

Primary or main memory refers to the storage area which can be directly accessed by the microprocessor. Therefore, all programs and data must be stored only in primary memory prior to execution. In primary memory the access time should be compatible with the read/write time of the processor. Therefore, only semiconductor memories are used as primary memories and they (the latest versions) are fabricated using CMOS technology. Primary memory normally includes ROM, EPROM, static RAM, DRAM and NVRAM.

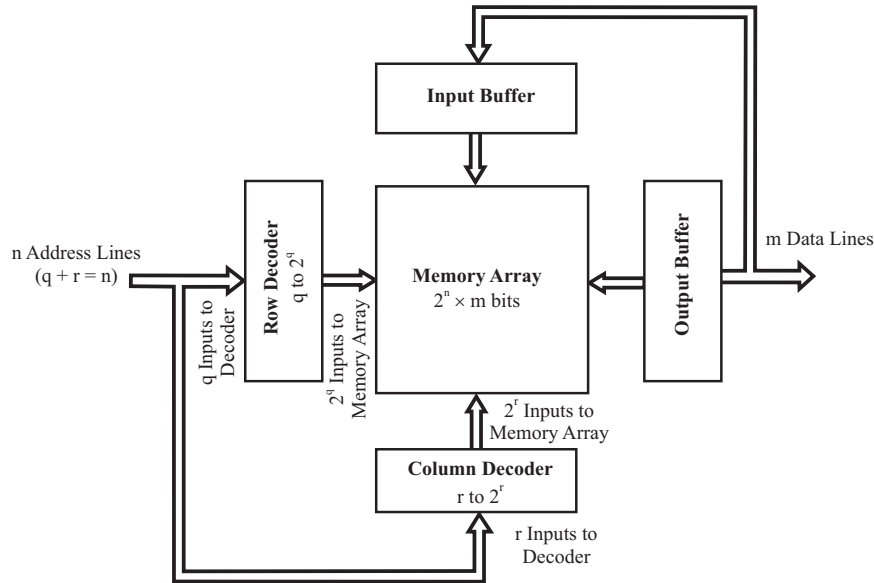
Secondary memory refers to the storage medium which comprises of slow devices such as magnetic tapes and disks (hard disk, floppy disc and Compact Disc (CD)). They are called as auxiliary or backup storage devices. These devices are used to hold large data files and huge programs such as operating systems, compilers, data bases, permanent programs, etc. The microcomputer system copies the required programs and data from secondary memory to main memory and work directly with main memory only.

## 3.2 SEMICONDUCTOR MEMORY

---

The main or primary memory elements are semiconductor devices, because the semiconductor devices alone can work at high speeds and consume less power. Moreover, they can be fabricated as ICs and also occupy less space.

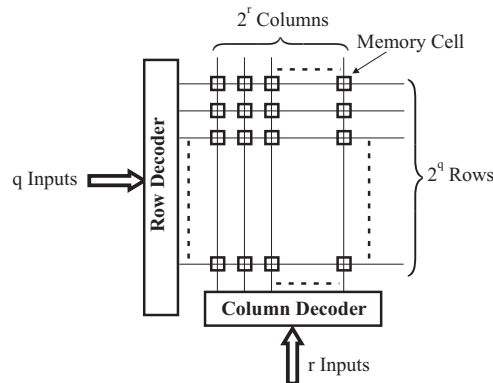
A typical semiconductor memory IC will have **n** address pins (lines) and **m** data pins (lines). The capacity of the memory will be  $2^n \times m$  bits. Figure 3.1 shows a simplified functional block diagram of a semiconductor memory. The functional blocks of a semiconductor memory are row address decoder, column address decoder, memory array, input buffer and output buffer.



**Fig. 3.1 :** Simplified functional block diagram of a typical semiconductor memory.

Input and output buffers are used to hold the data until a valid time and also takes care of signal current level matching (or Impedance matching). The  $n$  address lines are split into 'q' lines and 'r' lines, such that  $q + r = n$ . The 'q' address lines are applied as input to row decoder and 'r' address lines are applied as input to column decoder.

The output lines of the row and column decoder are used to form a matrix array of size,  $2^q \times 2^r$  consisting of  $2^n$  crossing points as shown in Fig. 3.2. Each crossing point is called memory cell and can store one-bit of binary information. A typical memory array will consist of  $m$  number of layers of matrix array as shown in Fig. 3.2 and all of them are wired parallelly. When an address is sent to memory IC, the row and column decoder will select one line each, which in turn will select one memory cell in each layer. Thus  $m$  memory cells are selected by an address. Then, using the read or write control signals, the data can be read or stored in the selected memory cells.



**Fig. 3.2 :** One layer of memory array.

In the first version of semiconductor memory, the memory cells were made of passive elements like resistors and capacitors. Later, diodes were used instead of passive elements. With advancement in semiconductor technology, Bipolar and MOS transistors were used to form memory cells. The latest technology used for fabricating memory cells are CMOS and HMOS which offers very low power and high speeds in operation.

The different types of semiconductor memories are ROM, PROM, EPROM, static RAM, DRAM and NVRAM. These semiconductor memories can be classified into volatile and non-volatile memories. If the information stored in a semiconductor memory is lost when the power supply to that IC is switched OFF, then the memory is called volatile. On the other hand if the stored information is retained even when the power supply is switched OFF, then the memory is called non-volatile. ROM, PROM, EPROM and NVRAM are non-volatile memories. Static RAM and DRAM are volatile memories.

Semiconductor memories can also be classified into read only and read/write memories. In read only memories, information is stored permanently either during manufacturing or after manufacturing and then interfaced to microcomputer system. The processor can only read the stored informations from these memories and cannot write into it. But in read/write memory, the processor can store (write) information as well as read from it. The ROM, PROM and EPROM are read only memories. The NVRAM, static RAM and DRAM are read/write memories.

Semiconductor memories also have random access and non-destructive readout features. In random access memory, the memory access time is independent of the memory location being accessed (i.e., the access time will be same for the first or the last location). All semiconductor memories are random access memories. In semiconductor memories, a read operation by the processor will not destroy the stored information and for this reason the semiconductor memory is also called NDRO memory (**Non-Destructive Read-Out** memory).

### 3.3 ROM AND PROM

---

ROM is a semiconductor memory which permits only a read access. The ROM functions as a memory array whose contents, once programmed, are permanently fixed and cannot be altered by the microprocessor to which the memory is interfaced. Other names for this type of memory are dead memory, fixed memory, permanent memory and **Read-Only Store (ROS)**. In ROM, the memory cell (storage unit) will have a MOS transistor either with an open gate or a closed gate. Transistors with closed gate represent **1's** and the ones with open gate represent **0's**. Since the configuration is fixed, they permanently store **1's** and **0's**.

The ROM is non-volatile memory, i.e., loss of power or system malfunction does not change the contents of the memory. Also, ROM memory has the feature of random access, which means that the access time for a given memory location is same as that for all other locations. The process of storing information in ROM is called programming. The technique employed for storing information in the ROM provides a convenient method for classifying ROMs into one of the following three categories. They are as follows:

- Custom programmed or Mask programmed ROM (ROM)
- Programmable or Field programmable ROM (PROM)
- Reprogrammable or Erasable-Programmable ROM (EPROM)

Custom programmed ROMs are programmed by the manufacturer as specified by the user during fabrication and the contents cannot be changed after packaging. Programmable ROM's are one time programmable by the user. Reprogrammable ROMs have facilities for programming as well as for erasing its content and reprogramming the memory. They are erased either by passing Electrical current or Ultraviolet light.

The programming of ROMs can be carried using ROM (EPROM) programmer. Usually the ROM programmer is a digital system interfaced to a Personal Computer (PC). The information to be programmed is first stored as a file in the PC and then converted in to the required binary format using a conversion software. Then the information is transferred from the PC to a ROM programmer.

### 3.4 EPROM

---

The **Read Only Memory (ROM)** which has a reprogrammable feature is called EPROM (**Erasable-Programmable Read Only Memory**). The EPROM memory is non-volatile and also has the feature of random access. In an EPROM, the binary information is entered using electrical impulses and the stored information is erased using ultraviolet rays. The typical erase time varies between 10 to 30 minutes.

In EPROM the memory cell (storage location of a bit) consists of a MOS transistor with an isolated gate. The isolated gate is located between the normal control gate and the source/drain region of an MOS transistor. This gate may be charged with electrons during the programming operation and when charged with electrons, the transistor is permanently turned OFF. The state of the floating gate, charged or uncharged, is permanent because the gate is isolated in an extremely pure oxide.

The charge on the isolated gate may be removed if the device is irradiated with ultraviolet light. The ultraviolet light allows the electrons to recombine and discharge through the control gate. The process of charging and discharging are repeatable.

The EPROM is programmed by inserting the EPROM chip into the socket of a PROM programmer and providing addresses and voltage pulses at the appropriate pins of the chip. Usually the PROM programmer is interfaced to a Personal Computer (PC) and the information to be programmed is downloaded from the PC.

EPROMs are manufactured by many semiconductor industries like INTEL, Hitachi, Toshiba, Cypress, etc. The manufacturers follow a common industry standard, so that a product from a different industry will be pin to pin compatible and differ slightly in electrical and switching characteristics. The various features of the 2764 (8 kb EPROM) manufactured by CYPRESS semiconductor Corporation are discussed in this section.

#### **CY27C64 (Cypress Make CMOS 2764)**

---

The CY27C64 is a high performance 8192 byte (8 kb) CMOS EPROM. It has power down mode, in which the device will enter a low-power standby mode when it is not enabled (or deselected).

The logic block diagram of CY27C64 is shown in Fig. 3.3 and the pin configuration during read mode is shown in Fig. 3.4. [The pin configuration of CY27C64 will be different to that of the configuration shown in Fig. 3.4 during programming or write mode.] The chip has thirteen



address inputs denoted as  $A_0$ - $A_{12}$ . The address is used to access any one of the 8 kilo (8192) locations within the chip. The eight output lines,  $O_0$  to  $O_7$  are used to output data from the chip. The chip will be in standby mode when  $\overline{CE}$  is inactive. The  $\overline{CE}$  is activated for selecting the chip and  $\overline{OE}$  is activated for enabling the output buffer during read operation.

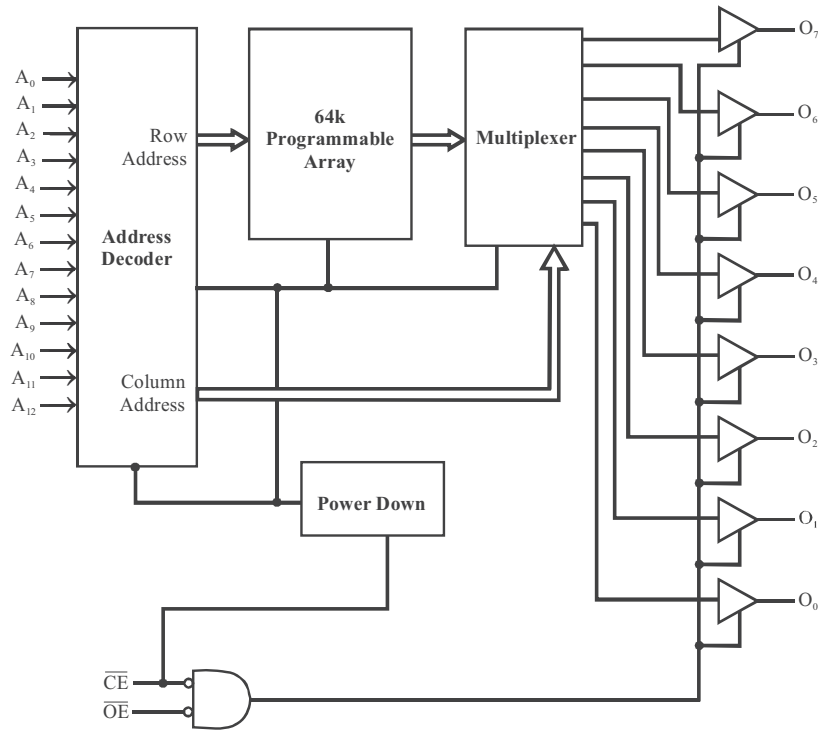


Fig. 3.3 : Logic block diagram of a CY27C64.

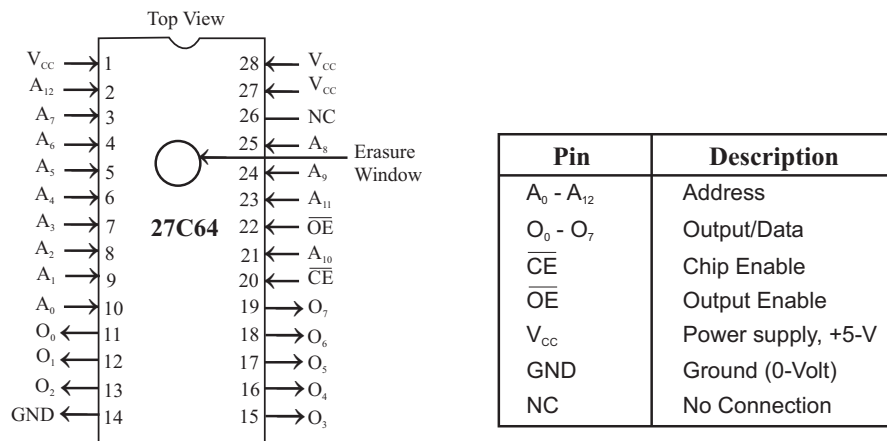


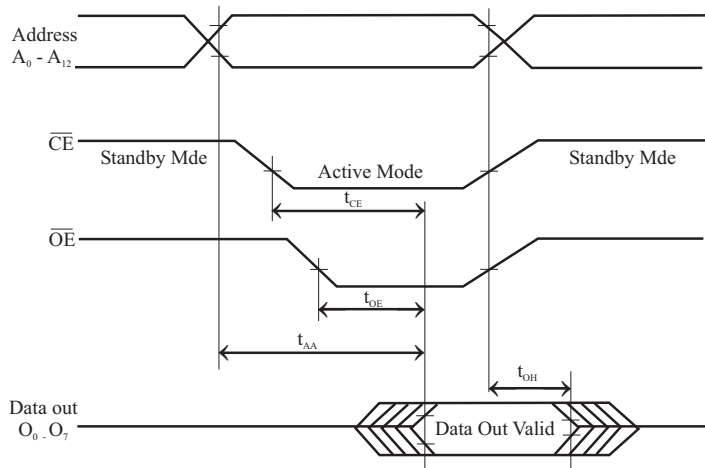
Fig. 3.4 : Pin configuration of a CY27C64 in read mode.

The CY27C64 EPROM is available with maximum access times of 70, 90, 120, 150 or 200 ns (nanosecond). The electrical characteristics or ratings of the EPROM are listed in Table-3.1.

**TABLE - 3.1 : ELECTRICAL CHARACTERISTICS OF CY27C64**

Description		Rating		Unit
		Min	Max	
Operating current	Commercial		80	mA
	Military		100	mA
Standby Current			15	mA
Output High Voltage		2.4		V
Output Low Voltage			0.4	V
Input High Voltage		2.0		V
Input Low Voltage			0.8	V
Output Capacitance			10	pF
Input Capacitance			10	pF

The timing diagram (or switching waveforms) of CY27C64 for read operation is shown in Fig. 3.5. Only four important timings are shown in this diagram. For detailed discussions on timing diagram refer to manufacturers data sheet. The switching timings of various signals of CY27C64 are listed in Table-3.2.



**Fig. 3.5 : Timing diagram of CY27C64 for read operation.**

The read operation is carried out in the following steps.

1. Place the address of the location to be read, on the address pins  $A_0 - A_{12}$ .
2. Enable the chip by asserting chip enable **low** ( $\overline{CE} = 0$ ).
3. Assert the output enable signal **low** ( $\overline{OE} = 0$ ).
4. The data can be read from the output lines ( $O_0$  to  $O_7$ ) after a delay time of  $t_{OE}$  (40 or 50 ns) after asserting  $\overline{OE}$  signal **low**.

**TABLE - 3.2 : SWITCHING CHARACTERISTICS OF CY27C64**

Parameter	Description	Time		Unit
		Min	Max	
$t_{AA}$	Address to output valid		70 to 200	ns
$t_{OE}$	Output enable active to output valid		40 or 50	ns
$t_{CE}$	Chip enable active to output valid		70 to 200	ns
$t_{OH}$	Data hold from address change	3		ns

When the address is placed on the address lines, the memory will take a time of  $t_{AA}$  to place the data on the output lines, provided the  $\overline{CE}$  and  $\overline{OE}$  are both asserted **low**, at the appropriate time.

The CY27C64 EPROM is equipped with an erasure window. When the window is exposed to UV light, the contents of EPROM are erased and then it can be reprogrammed. Wavelengths of light less than  $4000 \text{ \AA}$  (Angstrom unit) begin to erase the EPROM. Hence, an opaque label should be placed over the window if the EPROM is exposed to sunlight or fluorescent lighting for very long time.

The recommended dose of UV light for erasure is a wavelength of  $2537 \text{ \AA}$  for a minimum dose (UV intensity multiplied by exposure time) of  $25 \text{ W-sec/cm}^2$ . For an UV lamp with a  $12 \text{ mW/cm}^2$  power rating, the exposure time would be approximately 35 minutes. The EPROM has to be placed within a distance of 1 inch from the lamp during erasure. Permanent damage may result if EPROM is exposed to high-intensity UV light for very long time. (Maximum dosage is  $7258 \text{ W-sec/cm}^2$ .)

### 3.5 STATIC RAM

The static RAM (**R**andom **A**ccess **M**emory) is a read/write memory which consists of an array of flip-flops or similar storage devices. [Eventhough ROM memories are also technically random access memories, the read/write memories are called RAM.] Besides random access feature, the static RAMs are volatile in nature. In static RAM, the memory cell (storage location for each bit of information) consists of a flip-flop or a similar device. The stored information is retained in the memory cell as long as power is supplied to the circuit. Each memory cell typically consists of six to eight MOS transistors.

The static RAMs are manufactured by many semiconductor industries like Motorola, Hitachi, Toshiba, Cypress, etc. The manufacturers follow a common industry standard, so that a product from a different industry will be pin to pin compatible and differ slightly in electrical and switching characteristics. The various features of the 6264 (8 kb RAM) manufactured by CYPRESS Semiconductors Corporation are discussed in this section.

#### **CY6264 (Cypress Make CMOS 6264)**

The CY6264 is a high performance 8192 byte (8 kb) CMOS static RAM. The device has a power down mode. When CY6264 is not enabled (deselected), it will enter the power down mode and in this mode the power consumed is reduced to 30% of active mode power.

The logic block diagram and the pin configuration of CY6264 are shown in Fig. 3.6 and Fig. 3.7. The chip has 13 address inputs denoted as  $A_0$  -  $A_{12}$ . The address is used to access any one of the 8 kilo (8192) locations within the chip. It has eight IO pins for reading/writing the data and they are denoted as  $IO_0$  to  $IO_7$ .

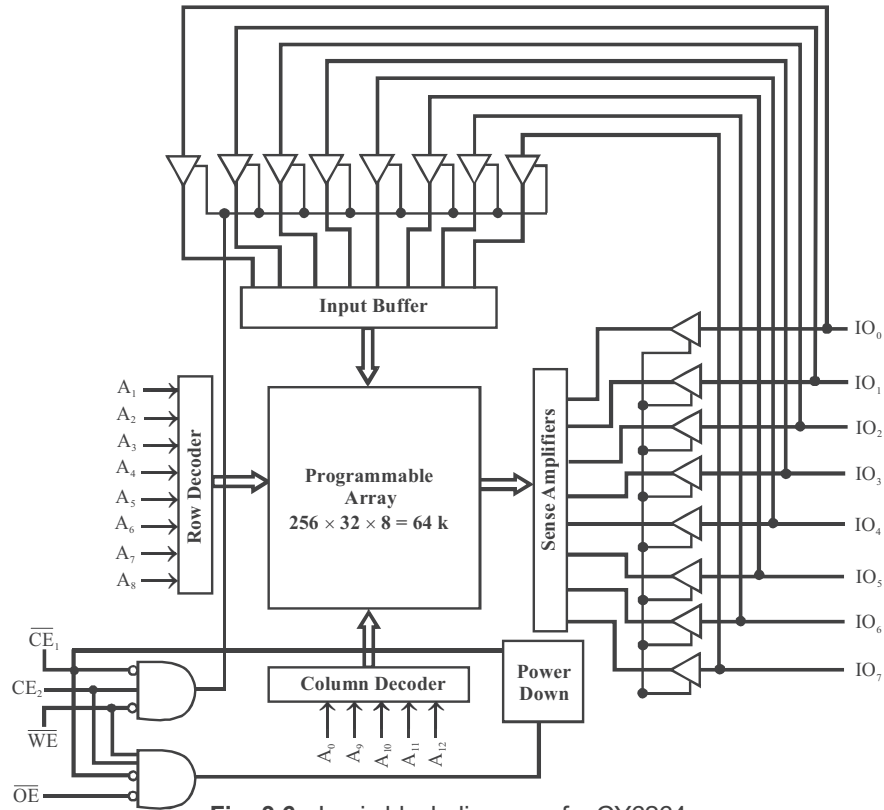


Fig. 3.6 : Logic block diagram of a CY6264.

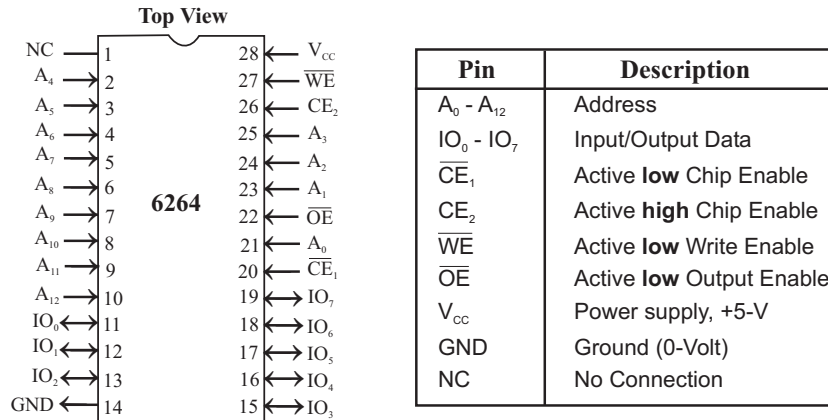


Fig. 3.7 : Pin configuration of a CY6264.

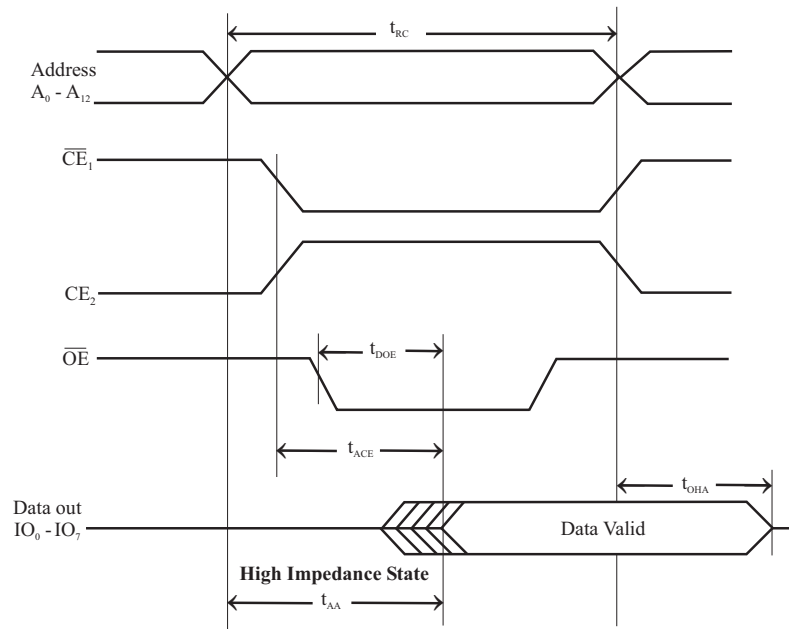
The chip has four control signals  $\overline{CE}_1$ ,  $CE_2$ ,  $\overline{WE}$  and  $\overline{OE}$ . When  $\overline{CE}_1$  and  $\overline{WE}$  inputs are both **low** and  $\overline{CE}_2$  is **high**, data on the eight data pins ( $IO_0$  through  $IO_7$ ) is written into the memory location addressed by the address pins ( $A_0$  through  $A_{12}$ ).

When  $\overline{CE}_1$  and  $\overline{OE}$  are both **low** and  $CE_2$  is **high**, the content of the memory location addressed by the address pins will be loaded on the eight data pins,  $IO_0$  to  $IO_7$ .

The CY6264 RAM is available with maximum access time of 55 or 70 ns (nanosecond). The electrical characteristics or ratings of the RAM are listed in Table-3.3.

**TABLE - 3.3 : ELECTRICAL CHARACTERISTICS OF CY6264**

Description	Value		Unit
	Min	Max	
Operating current		100	mA
Standby current		15 or 20	mA
Output high voltage	2.4		V
Output low voltage		0.4	V
Input high voltage	2.2	$V_{cc}$	V
Input low voltage	-0.5	0.8	V
Output capacitance		7	pF
Input capacitance		7	pF



**Fig. 3.8 : Read cycle timings of a CY6264.**

**TABLE - 3.4 : READ CYCLE TIMINGS OF CY6264**

Parameter	Description	Time		Unit
		Min	Max	
$t_{RC}$	Read cycle time	70		ns
$t_{AA}$	Address to data valid		70	ns
$t_{OHA}$	Data hold from address change	5		ns
$t_{ACE}$	CE <b>low/high</b> to data valid		70	ns
$t_{DOE}$	$\overline{OE}$ <b>low</b> to data valid		35	ns

The timing diagram (or switching waveforms) of CY6264 for read operation is shown in Fig. 3.8. Only five important timings are shown in this diagram. For further information of timing diagram, please refer to manufacturers data sheet. Timings of various signals of CY6264 are listed in Table-3.4.

The read operation is carried out in the following steps:

1. Place the address of the location to be read, on the address pins  $A_0$ - $A_{12}$ .
2. Enable the chip by asserting chip enable-1 ( $\overline{CE}_1$ ) as **low** and chip enable-2 ( $CE_2$ ) as **high**.
3. Assert the **Output Enable** ( $\overline{OE}$ ) signal **low**.
4. When  $\overline{OE}$  signal is asserted **low**, the data can be read from the Input/Output lines ( $IO_0$  to  $IO_7$ ) after a delay time of  $t_{DOE}$  (35 ns).

When the address is placed on the address line, the memory will take a time of  $t_{AA}$  to place the data on the output lines, provided the  $\overline{CE}_1$  and  $\overline{OE}$  are asserted **low** and  $CE_2$  is asserted **high** at the appropriate time.

The timing diagram (or switching waveform) of CY6264 for write operation is shown in Fig. 3.9. The diagram shows some important timings of write cycle. For detailed discussions on timing diagram, refer to manufacturers data sheet. The timings of various signals are listed in Table-3.5.

The write operation is carried in the following steps:

1. Place the address of the location to be written, on the address pins  $A_0$  -  $A_{12}$ .
2. Enable the chip by asserting  $\overline{CE}_1$  signal as **low** and after a small delay assert  $CE_2$  signal as **high**.
3. Assert the write enable  $\overline{WE}$  signal as **low**.
4. Place the data to be written on the  $IO_0$  to  $IO_7$  lines immediately after  $\overline{WE}$  is asserted **low**.

After the address is placed on the address lines and  $\overline{CE}_1$ ,  $CE_2$  and  $\overline{WE}$  are asserted appropriately, the data has to be placed on the data lines within the time  $t_{SD}$  (data set-up to write end).

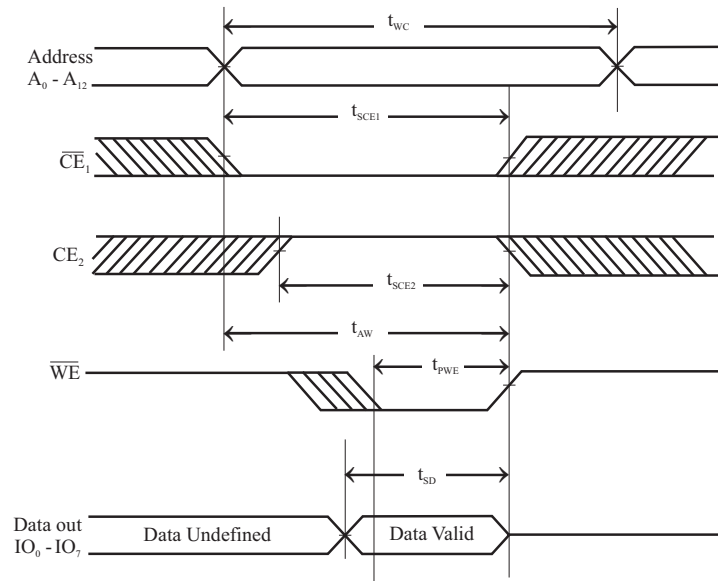


Fig. 3.9 : Write cycle timings of CY6264.

TABLE - 3.5 : WRITE CYCLE TIMINGS OF CY6264

Parameter	Description	Minimum time	Unit
$t_{WC}$	Write cycle time	50	ns
$t_{SCE1}$	$\overline{CE}_1$ low to write end	40	ns
$t_{SCE2}$	$CE_2$ high to write end	30	ns
$t_{AW}$	Address setup to write end	40	ns
$t_{PWE}$	$\overline{WE}$ pulse width	25	ns
$t_{SD}$	Data set-up to write end	25	ns

### 3.6 DRAM AND NVRAM

#### DRAM

DRAM (**D**ynamic **R**AM) is a read/write memory in which information is stored in the form of electric charge on the gate-to-substrate capacitance of a MOS transistor. This charge dissipates in a few milliseconds and the element must be refreshed periodically. DRAMs are volatile and have random access feature.

Dynamic RAMs are important because fewer elements are required to store a bit (typically each memory cell will have three to four transistors as opposed to six to eight in static RAM), so that more bits can be packed into an IC of a given physical area. They are also faster than the static RAM and consume less power in the quiescent state.

Refreshing of DRAMs need extra circuitry. So, the interfacing of DRAMs to microprocessor are more complex than the interfacing of static RAMs, the recent versions of DRAMs internal refreshing circuit. Manufacturing of DRAMs will be cheaper only for very large capacity memories. Therefore, small memories are generally static elements (upto 256 kb) and large memories (> 1Mb) are typically dynamic.

### **NVRAM**

The NVRAMs are non-volatile read/write memories. They are also called flash memory. These memory devices are electrically erasable in the system, but require more time to be erased than a static RAM. Therefore they are also called EEPROM (Electrically Erasable Programmable ROM). The drawback in EEPROMs is that it takes a long time for the system to erase and write. The maximum number of write operations that can be performed in most of the EEPROMs is about 10,000 operations.

The INTEL and XYCOR have released their versions of non-volatile RAM, which does not have the drawbacks of EEPROMs. (The drawbacks of EEPROM are high write time and limited number of write cycles.) This type of NVRAM consists of a high speed static RAM and a corresponding EEPROM on a single chip. For this reason it is also called shadow RAM. In these devices, for each cell of static RAM, there is one EEPROM cell. A typical example of shadow RAM is INTEL 2004 and XYCOR's X2004. The 2004 edition has a special pin called Non-volatile Enable (NE). Normally this pin is **high** and read or write operation is performed with static RAM. When  $\overline{NE}$  is asserted **low**, the data in the static RAM cells are written into the corresponding EEPROM cells.

### **3.7 INTERFACING STATIC RAM AND EPROM**

The primary function of memory interfacing is that the microprocessor should be able to read from and write into a set of semiconductor memory IC chips. Generally EPROM is interfaced for read operations and RAM is interfaced for read and write operations. The procedure for interfacing SRAM for read/write operation and EPROM for read operation are similar. So, they are dealt commonly in this section.

In order to perform the read/write operation the memory access time should be less than the read/write time of processor, chip select signals should be generated for selecting a particular memory IC, suitable control signals have to be generated for read/write operation and a specific address should be allotted to each memory location.

Hence memory interfacing deals with choosing memories with suitable access time, designing address decoding circuit to generate chip select signals, generating control signals for read/write operation and allocation of addresses to various memory ICs and their locations.

#### **Typical EPROM and Static RAM**

A typical semiconductor memory IC will have **n** address pins, **m** data pins (or output pins) and a minimum of two power supply pins (one for connecting required supply voltage ( $V_{CC}$ ) and the other for connecting ground). The control signals needed for static RAM are chip select (chip enable), read control (output enable) and write control (write enable). The control signals needed for read operation in EPROM are chip select (chip enable) and read control (output enable). A typical static RAM and EPROM are shown in Fig. 3.10 and Fig. 3.11 respectively.



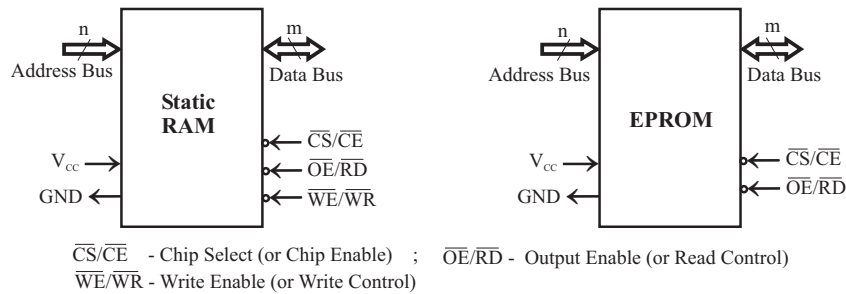


Fig. 3.10 : A typical static RAM IC.

Fig. 3.11 : A typical EPROM IC in read mode.

*Note : The pins of EPROM are redefined for write operation. An EPROM requires a different hardware setup and high supply voltage for write operation.*

### Memory Capacity

A semiconductor memory IC will have **n** address pins and **m** data pins. Such a memory has  $2^n$  locations and each location can store **m**-bit data. The size of data stored in each memory location is called memory word size. In INTEL 8085-based systems normally memories with word size of 1-byte are used. (But we can even interface memories with word size 1-bit, 2-bit and 4-bit.)

The memory capacity is specified in kilo bytes. If the memory IC has **m** data pins and **n** address pins, then the memory IC will have a capacity of  $2^n \times m$  bits. When  $m = 8$ , the memory capacity is  $2^n$  bytes. One kilo-byte is  $1024_{10}$  ( $= 400_H$ ) bytes. The relation between address pins and capacity of memory ICs are listed in Table-3.6.

**TABLE - 3.6 : RELATION BETWEEN NUMBER OF ADDRESS PINS AND MEMORY CAPACITY**

Number of address pins	Memory capacity			Range of address in hexa
	in decimal	in kilo	in hexa	
10	$2^{10} = 1024$	1k	400	000 to 3FF
11	$2^{11} = 2 \times 2^{10} = 2048$	2k	800	000 to 7FF
12	$2^{12} = 2^2 \times 2^{10} = 4 \times 2^{10} = 4096$	4k	1000	000 to FFF
13	$2^{13} = 2^3 \times 2^{10} = 8 \times 2^{10} = 8192$	8k	2000	0000 to 1FFF
14	$2^{14} = 2^4 \times 2^{10} = 16 \times 2^{10} = 16384$	16k	4000	0000 to 3FFF
15	$2^{15} = 2^5 \times 2^{10} = 32 \times 2^{10} = 32768$	32k	8000	0000 to 7FFF
16	$2^{16} = 2^6 \times 2^{10} = 64 \times 2^{10} = 65536$	64k	10000	0000 to FFFF

### Choice of Memory ICs and Address Allocation

The memory requirement of a system depends on the application for which it is designed. A system designer has a variety of choices for choosing memory ICs. The total memory requirement can be realized in a single IC or in multiple ICs.

The total memory requirement of the system will be split between EPROM and RAM memories. The EPROM memories are used for storing monitor programs, other permanent programs and data. The RAM memories are used for stack operations, temporary program and data storage.

Popular EPROM and static RAM ICs with 8085 systems and their capacity are listed here. Table-3.7 shows the number of address pins and data pins available on these ICs.

**EPROM**

2708 ( $1k \times 8 = 8$  kilo bits/1kb)  
 2716 ( $2k \times 8 = 16$  kilo bits/2 kb)  
 2732 ( $4k \times 8 = 32$  kilo bits/4 kb)  
 2764 ( $8k \times 8 = 64$  kilo bits/8 kb)  
 27256 ( $32k \times 8 = 256$  kilo bits/32 kb)  
 27512 ( $64k \times 8 = 512$  kilo bits/64 kb)

**Static RAM**

6208 ( $1k \times 8 = 8$  kilo bits/1kb)  
 6216 ( $2k \times 8 = 16$  kilo bits/2 kb)  
 6232 ( $4k \times 8 = 32$  kilo bits/4 kb)  
 6264 ( $8k \times 8 = 64$  kilo bits/8 kb)  
 62256 ( $32k \times 8 = 256$  kilo bits/32 kb)  
 62512 ( $64k \times 8 = 512$  kilo bits/64 kb)

*Note : In this book kb refers to kilobytes.*

**TABLE - 3.7 : NUMBER OF ADDRESS AND DATA PINS IN MEMORY ICs**

Memory IC EPROM/RAM	Capacity	Number of address pins	Number of data pins
2708/6208	1 kb	10	8
2716/6216	2 kb	11	8
2732/6232	4 kb	12	8
2764/6264	8 kb	13	8
27256/62256	32 kb	15	8
27512/62512	64 kb	16	8

*Note : 16kb memory is not available as a standard product.*

In 8085 system, the EPROM is mapped at the beginning of memory space. (i.e.,  $0000_H$  address is allotted to EPROM memory location). Whenever the power supply is switched ON, the microprocessor chip will be reset. This power-on reset will be implemented by the system designer. When the processor is reset all the internal registers, flag register and program counter will be cleared. Hence, after a reset, the program counter will have an address  $0000_H$  and so the processor starts fetching and executing the instruction stored at  $0000_H$ .

The system designer will store the monitor program starting from the address  $0000_H$ . The monitor program should be executed to initialize system peripherals whenever the system is switched ON. To enable automatic execution of monitor program, whenever the system is switched ON, the EPROM should be mapped from  $0000_H$  location in 8085-based system. Monitor program is a permanent program written by the system designer to take care of system initializations. System initializations includes the following :

- Programming 8279 for keyboard scanning and display refreshing.
- Programming peripheral ICs 8259, 8257, 8255, 8251, 8254, etc.
- Initializing stack.
- Display a message on display (output) device.
- Initializing interrupt vector table.

<i>Note : 8279 - Programmable keyboard/display controller.</i>	<i>8257 - DMA controller.</i>
<i>8259 - Programmable interrupt controller.</i>	<i>8251 - USART.</i>
<i>8255 - Programmable peripheral interface.</i>	<i>8254 - Programmable timer.</i>

### Generation of Chip Select Signals

Decoders are used to generate chip select signals. The 2-to-4 decoder will give four chip select signals. The 3-to-8 decoder will give eight chip select signals. The 4-to-16 decoder will give sixteen chip select signals.

Decoder is a logic circuit that identifies each combination of the signals present at its input. Decoders have  $n$  input lines and  $2^n$  output lines. In logic **low** decoder, at any one time one of the  $2^n$  outputs will remain **low** and all other outputs will remain **high**.

The output which remains **low** depends on the input signal. Hence if the decoder outputs are connected to chip select pins of ICs in the microprocessor system at any one time, only one chip will be selected. The input to the decoders are unused address lines or high order address lines.

While interfacing memories, low order address lines are connected to memory ICs. The remaining unused address lines (or high order address lines) are connected to the input of the decoder. The outputs of the decoder are connected to  $\overline{CS}$  or  $\overline{CE}$  pins of memory ICs.

In a microprocessor-based system, all the memory ICs and peripheral ICs are connected to a common system bus. Therefore, the data, address and control lines are connected to all the slaves (memory/peripheral ICs). But all the slaves remain in **high impedance** state. So, they cannot communicate with the master (processor) through bus (i.e., they are physically connected but electrically isolated).

When the address is given out by the processor for read/write operation, only one of the memory ICs is selected and the selected memory IC will come to normal logic. The selection logic depends on address decoding logic. All other memory ICs will remain in **high impedance** state. So, they are electrically isolated from the system. The read/write operation is performed by the processor with the selected memory IC.

### Decoder

Popular decoders used in the microprocessor-based system are 74LS138 and 74LS139. The 74LS138 is a 3-to-8 decoder and 74LS139 is dual 2-to-4 decoder.

The 74LS138 decoder consists of 3-input lines, 8-output lines (logic **low**) and three enables or ground. In the three enables, two are logic **low** and one is a logic **high** enable. The pin configuration of 3-to-8 decoder (74LS138) is shown in Fig. 3.12. The truth table of the decoder is given in Table-3.8.

The 74LS139 decoder consists of two numbers of 2-to-4 decoder packed in a single IC package. Each decoder has two input pins, four output lines and a logic **low** enable. The pin configuration of 74LS139 is shown in Fig. 3.13. The truth table of 2-to-4 decoder is given in Table-3.9. In the 74LS139 each decoder can work independently.

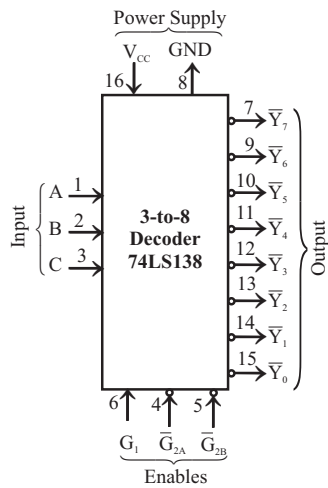


Fig. 3.12 : Signals of 74LS138.

TABLE - 3.8 : TRUTH TABLE OF 3-TO-8 DECODER

Enables			Input			Output							
$G_1$	$\overline{G}_{2A}$	$\overline{G}_{2B}$	C	B	A	$\overline{Y}_7$	$\overline{Y}_6$	$\overline{Y}_5$	$\overline{Y}_4$	$\overline{Y}_3$	$\overline{Y}_2$	$\overline{Y}_1$	$\overline{Y}_0$
1	0	0	0	0	0	1	1	1	1	1	1	1	0
1	0	0	0	0	1	1	1	1	1	1	0	1	1
1	0	0	0	1	0	1	1	1	1	1	0	1	1
1	0	0	0	1	1	1	1	1	1	0	1	1	1
1	0	0	1	0	0	1	1	1	0	1	1	1	1
1	0	0	1	0	1	1	1	0	1	1	1	1	1
1	0	0	1	1	0	1	1	0	1	1	1	1	1
1	0	0	1	1	1	0	1	1	1	1	1	1	1
0	1	1	X	X	X	H	H	H	H	H	H	H	H

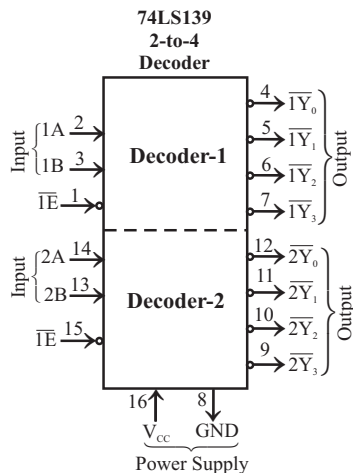


Fig. 3.13 : Signals of 74LS139.

TABLE - 3.9 : TRUTH TABLE OF THE  
2-TO-4 DECODER

Enable	Input		Output			
$\overline{E}$	B	A	$\overline{Y}_3$	$\overline{Y}_2$	$\overline{Y}_1$	$\overline{Y}_0$
0	0	0	1	1	1	0
0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	1
1	X	X	H	H	H	H

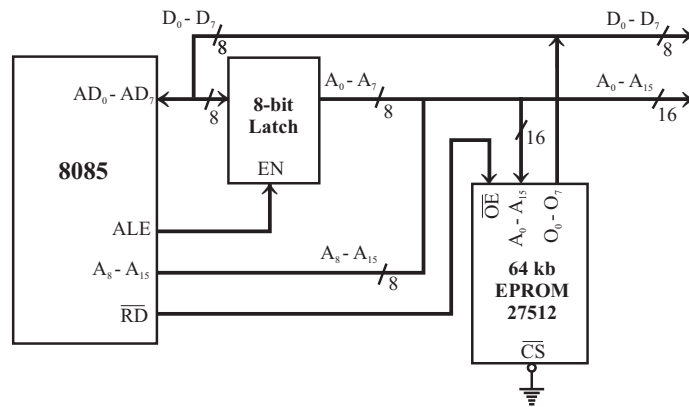
### 3.8 MEMORY ORGANIZATION IN 8085-BASED SYSTEM

A microprocessor-based system requires both EPROM and RAM. Hence the available memory space has to be divided between EPROM and RAM. The 8085 has 64kb of addressable memory space and allotting this address space for EPROM and RAM depends on the system designer as well as the application for which the system is designed.

Some systems may require large memory space. So, the full memory space is utilized. But in some systems the memory requirement may be less and in this case the full memory space will not be utilized. When the full memory space is not utilized, the unused memory addresses can be used for addressing IO devices. Such IO devices are called memory-mapped IO devices and they can be accessed similar to that of a memory device.

The required EPROM memory capacity of the system can be implemented in one IC or in multiple ICs. Similarly the RAM capacity of the system can be implemented in one IC or in multiple ICs. This choice depends on the availability of memory IC and the system designer. Some examples of memory organizations for 8085 microprocessor-based system are discussed in this section.

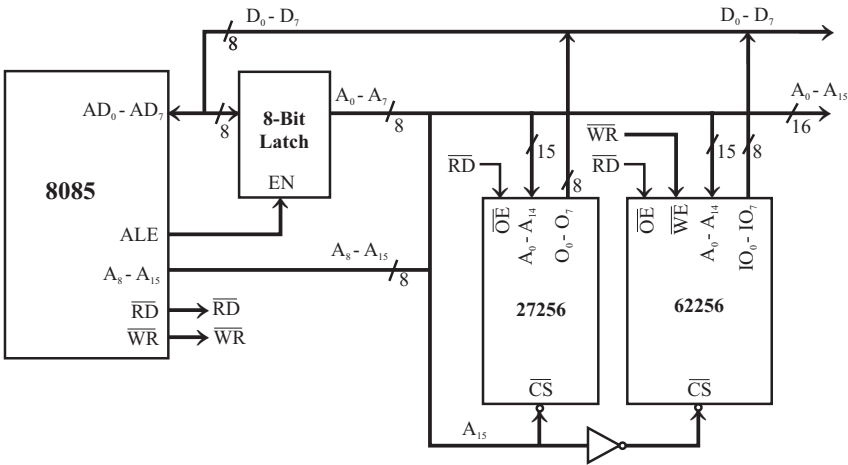
Consider a system in which the full memory space 64kb is utilized for EPROM memory. In this system the entire 16 address lines of the processor are connected to address input pins of memory IC in order to address the internal locations of memory and Chip Select ( $\overline{CS}$ ) pin of EPROM is permanently tied to logic **low** (i.e., tied to ground) as shown in Fig. 3.14. Now the range of address for EPROM is  $0000_H$  to  $FFFF_H$ .



**Fig. 3.14 :** Example of implementing 64 kb EPROM in the 8085 system.

Consider a system in which the available 64 kb memory space is equally divided between EPROM and RAM. Let us implement 32 kb memory capacity of EPROM using single IC 27256. Similarly, 32 kb RAM capacity is implemented using single IC 62256. The 32 kb memory requires 15 address lines and so the address lines  $A_0 - A_{14}$  of the processor are connected to 15 address pins of both EPROM and RAM as shown in Fig. 3.15. The unused address line  $A_{15}$  is used as a chip select signal for selecting either EPROM or RAM. The  $A_{15}$  line is directly connected to the  $\overline{CS}$  pin of EPROM and it is inverted and connected to  $\overline{CS}$  pin of RAM. Therefore, the EPROM is selected when  $A_{15} = 0$  and RAM is selected when  $A_{15} = 1$ . The address range of EPROM will be  $0000_H$  to  $7FFF_H$  and that of RAM will be  $8000_H$  to  $FFFF_H$ .

Consider a system in which 32kb memory space is implemented using four 8 kb memory. Let two 8 kb memory be EPROM and the remaining two be RAM. Each 8kb memory requires 13 address lines. So, the address lines  $A_0 - A_{12}$  of the processor are connected to 13 address pins of all the memory ICs. The address lines  $A_{13}$  and  $A_{14}$  can be decoded using a 2-to-4 decoder to generate four chip select signals. These four chip select signals can be used to select one of the four memory IC at any one time. The address line  $A_{15}$  is used as an enable for the decoder. The simplified schematic of this memory organization is shown in Fig. 3.16 and address allotted to each memory IC is shown in Table-3.10.



**Fig. 3.15 :** Example of implementing 32 kb EPROM and 32 kb RAM in an 8085 system.

**TABLE - 3.10 : ADDRESS ALLOCATION FOR MEMORY ICs SHOWN IN FIG. 3.16**

[illegible]

Consider a system in which the 64kb memory space is implemented using eight numbers of 8kb memory. Each 8kb memory requires 13 address lines and so the address line  $A_0$ - $A_{12}$  of the processor are connected to 13 address pins of all the memory ICs. The address lines  $A_{13}$ ,  $A_{14}$  and  $A_{15}$  are decoded using a 3-to-8 decoder to generate eight chip select signals. These eight chip select signals can be used to select one of the eight memory IC at any one time. Design example-2 given at the end of this chapter is an example of implementing 64kb address space using 8 numbers of 8kb memory.

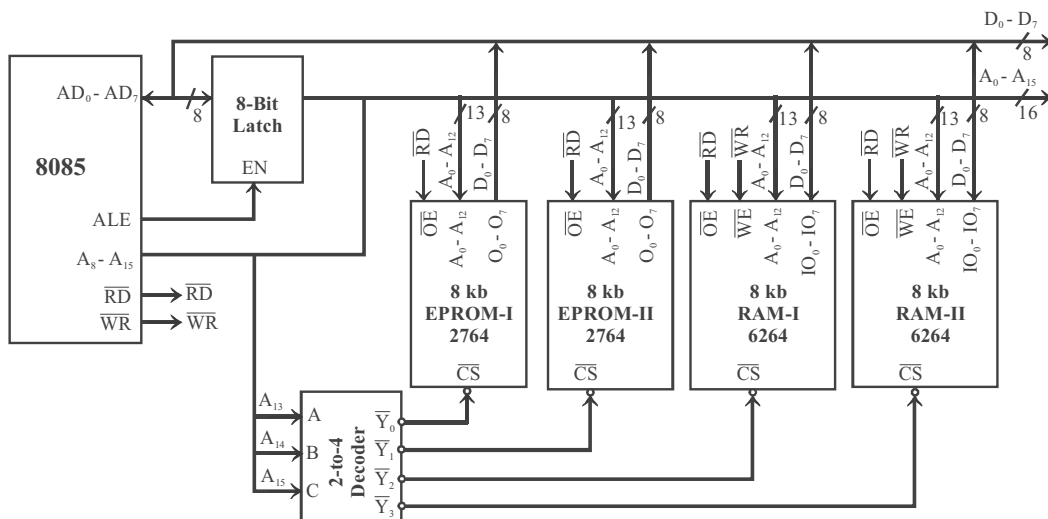


Fig. 3.16 : Example of implementing 16 kb EPROM and 16 kb RAM in an 8085 system.

### 3.9 IO STRUCTURE OF A TYPICAL MICROCOMPUTER

The IO devices connected to a microcomputer system provides an efficient means of communication between the microcomputer system and the outside world. These IO devices are commonly called peripherals and include keyboards, CRT displays, printers and disks (floppy disk, hard disk and Compact Disc (CD)).

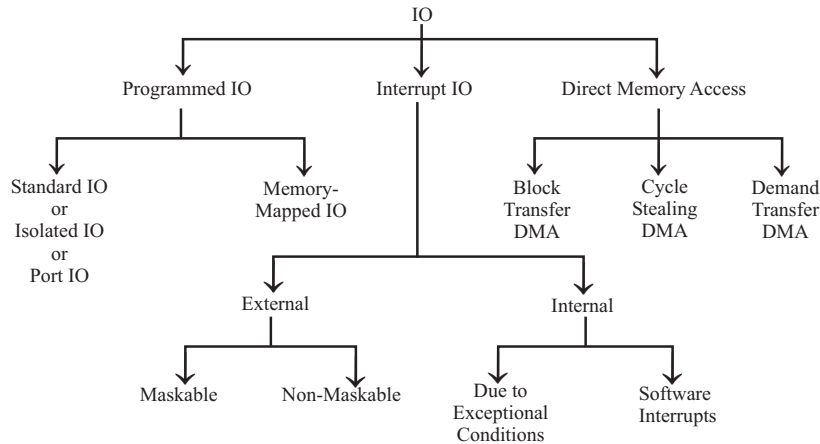
The characteristics of the IO devices are normally different from the characteristics of the microprocessor. Since the characteristics of the IO devices are not compatible with that of the microprocessor, interface hardware circuitry between the microprocessor and IO device are necessary.

There are three major types of data transfer between the microcomputer and an IO device. They are as follows:

- Programmed IO
- Interrupt driven IO
- Direct memory access (DMA)

In programmed IO the data transfer is accomplished through an IO port and controlled by software. In interrupt driven IO, the IO device will interrupt the processor and initiate data transfer. In DMA, the data transfer between memory and IO can be performed by bypassing the

microprocessor. Each type of data transfer scheme mentioned above, includes different methods of data transfer schemes. Figure 3.17 shows all the types of data transfer schemes in a microcomputer and it can also be called IO structure of a microcomputer.



**Fig. 3.17 :** IO structure of a typical microcomputer.

### 3.10 INTERFACING IO AND PERIPHERAL DEVICES

The IO devices are generally slow devices. So, they are connected to the system bus through ports. The ports are buffer IC which is used to temporarily hold the data transmitted from the microprocessor to IO device or to hold the data transmitted from IO device to the microprocessor.

To data transfer from the input device to the processor the following operations are performed:

- The input device will load the data to the port.
- When the port receives the data, it sends message to the processor to read the data.
- The processor will read the data from the port.
- After the data has been read by the processor the input device will load the next data into the port.

To data transfer from the processor to the output device the following operations are performed:

- The processor will load the data to the port.
- The port will send a message to the output device to read the data.
- The output device will read the data from the port.
- After the data has been read by the output device the processor can load the next data to the port.

#### INTEL IO Port Devices

The various INTEL IO port devices are 8212, 8155/8156, 8255, 8355 and 8755.

#### INTEL 8212

The 8212 is a 24-pin IC. It consists of eight number of D-type latches, each followed by a tristate buffer. It has 8-input lines  $DI_1$  to  $DI_8$  and 8-output lines  $DO_1$  to  $DO_8$ . The 8212 can be used as an input or output device and the function is determined by the mode pin. However, it cannot be used simultaneously for input and output in the same circuit, since its mode pin is hardwired. It has 2-device select signals  $\overline{DS}_1$  and  $DS_2$ . The port is selected by the processor by sending appropriate address to device select pins.



**Output Port :** When  $MD = 1$ ,  $\overline{DS}_1 = 0$  and  $DS_2 = 1$

**Input Port :** When  $MD = 0$ ,  $\overline{DS}_1 = 0$  and  $DS_2 = 1$

### INTEL 8155

INTEL 8155 has  $256 \times 8$  static RAM, two numbers of 8-bit parallel IO port (ports A and B), one number of 6-bit parallel IO port (port-C) and 14-bit timer. The ports A and B can be programmed to work as simple or handshake input or output port. If port-A and port-B are simple ports then port-C can be used as input or output port. The timer can be programmed to operate in four different modes. INTEL 8155 requires six internal addresses and has one logic **low** Chip Select pin ( $\overline{CS}$ ). The addresses of internal devices of 8155 are listed in Table-3.11.

**TABLE - 3.11 : INTERNAL ADDRESS OF 8155/8156**

Internal device	$A_2$	$A_1$	$A_0$
Control Register/ Status Register	0	0	0
Port-A	0	0	1
Port-B	0	1	0
Port-C	0	1	1
LSB of Timer	1	0	0
MSB of Timer	1	0	1

### INTEL 8156

INTEL 8156 is same as 8155, but it has logic **high** Chip Select ( $CS$ ), i.e., the chip is selected when  $CS = 1$ .

### INTEL 8255

It has 3 numbers of 8-bit parallel IO ports (ports A, B and C).

Port-A can be programmed in mode-0, mode-1 or mode-2 as input or output port. Port-B can be programmed in mode-1 and mode-2 as IO port. When ports A and B are in mode-0, port-C can be used as IO port. The individual pins of port-C can be set or reset. INTEL 8255 requires four internal addresses and has one logic **low** Chip Select ( $\overline{CS}$ ) pin. The address of internal devices of 8255 are listed in Table-3.12.

**TABLE - 3.12 : INTERNAL ADDRESS OF 8255**

Internal device	$A_1$	$A_0$
Port-A	0	0
Port-B	0	1
Port-C	1	0
Control Register	1	1

### INTEL 8355

It has  $2k \times 8$  ROM and two numbers of 8-bit port (Ports A and B). The individual pins of ports A and B can be programmed as input or output lines by sending a control word to DDR (**D**ata **D**irection **R**egister). The address of internal devices of 8355 are listed in Table-3.13. The 8355 requires four internal addresses and has one logic **low** Chip Select ( $\overline{CS}$ ) pin.

### INTEL 8755

Same as 8355 but has  $2k \times 8$  EPROM.

### INTEL peripheral devices

Apart from port ICs, dedicated programmable controller/peripheral ICs are used in the system for various activities. Some of the controller/peripheral devices used in the 8085 system and their functions and internal addresses are listed in Table-3.14.

**TABLE - 3.13 : INTERNAL ADDRESS OF 8355/8755**

Internal device	$A_1$	$A_0$
Port-A	0	0
Port-B	0	1
DDRA	1	0
DDR B	1	1

**TABLE - 3.14 : FUNCTIONS AND INTERNAL ADDRESSES OF PERIPHERAL DEVICES**

Device	Function	Internal addresses
INTEL 8279	Keyboard/display controller. Used for keyboard scanning and display refreshing.	<b>Two-internal addresses</b> $A_0 = 0 \rightarrow$ Data register $A_0 = 1 \rightarrow$ Control register
INTEL 8257 or INTEL 8237	DMA controller. Used for supporting DMA access to the IO device. It acts as a master during the DMA mode. It is a slave device during programming mode.	<b>Sixteen-internal addresses</b> $A_3 \ A_2 \ A_1 \ A_0$ 0   0   0   0 0   0   0   1 .   .   .   . 1   1   1   1
INTEL 8259	Interrupt controller. Used to expand the hardware interrupt INTR to eight interrupts in an 8085-based system and 256 interrupts in an 8086-based system.	<b>Two-internal addresses</b> $A_0 = 0$ $A_0 = 1$
INTEL 8253/ 8254	Programmable timer. Used in the system to produce various timing signals. It has three independent counters and can be programmed in six operating modes.	<b>Four-internal addresses</b> $A_1 \ A_0$ Counter-0     0   0 Counter-1     0   1 Counter-2     1   0 Control Register 1   1
INTEL 8251 (USART)	<b>Universal Synchronous/Asynchronous Receiver Transmitter.</b> Used for serial data communication.	<b>Two-internal addresses</b> $C/\bar{D} = 0 \rightarrow$ Data register $C/\bar{D} = 1 \rightarrow$ Control register

**IO Mapping**

The port and peripheral devices will have one logic **low/high** chip select pin. The processor can access the port/peripheral device by supplying internal address and chip select signals. Therefore, the port and peripheral device interfacing (IO interfacing) deals with allocation of various internal addresses and generation of chip select signals.

There are two ways of interfacing IO devices in 8085-based system.

- Memory-mapped IO device.
- Standard IO-mapped IO device or Isolated IO mapping.

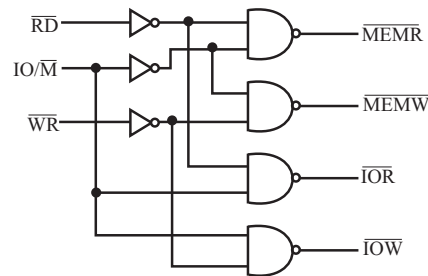
*Note : The interfacing of IO ports and controller/peripheral ICs are commonly referred as IO device mapping.*

In memory mapping of IO devices the ports are allotted a 16-bit address like that of the memory location. Some of the chip select signals generated to select memory ICs are used for selecting the IO port devices. Hence, the processor treats the IO ports as memory locations for reading and writing (i.e., the devices which are mapped by memory mapping are accessed by executing memory read cycle or memory write cycle).

In standard IO mapping or isolated IO mapping, a separate 8-bit address is allotted for the IO ports and the peripheral ICs. The processor differentiates the IO-mapped devices, from the memory-mapped devices in the following ways:

1. For accessing the IO-mapped devices the processor executes IO read or write cycle.
2. During IO read or write cycle, the 8-bit address is placed on both low order address lines and the high order address lines.
3.  $\text{IO}/\overline{\text{M}}$  is asserted **high** to indicate the IO operation (for read as well as write).

A 8085 processor does not provide separate read ( $\overline{\text{RD}}$ ) and write ( $\overline{\text{WR}}$ ) signals for memory and IO devices. But it differentiates the memory and IO device accessed by  $\text{IO}/\overline{\text{M}}$  signal. The three signals  $\overline{\text{RD}}$ ,  $\overline{\text{WR}}$  and  $\text{IO}/\overline{\text{M}}$  can be decoded as shown in Fig. 3.18 to provide separate read and write control signals for IO devices and memory devices.



**Fig. 3.18 :** Circuit to generate separate read and write signals for memory and IO devices in an 8085-based system.

When the devices are IO-mapped, then only IN and OUT instructions have to be used for data transfer between the device and the processor. For the IO-mapped devices a separate decoder should be used to generate the required chip select signals.

**TABLE - 3.15 : COMPARISON OF MEMORY MAPPING AND IO MAPPING OF IO DEVICE**

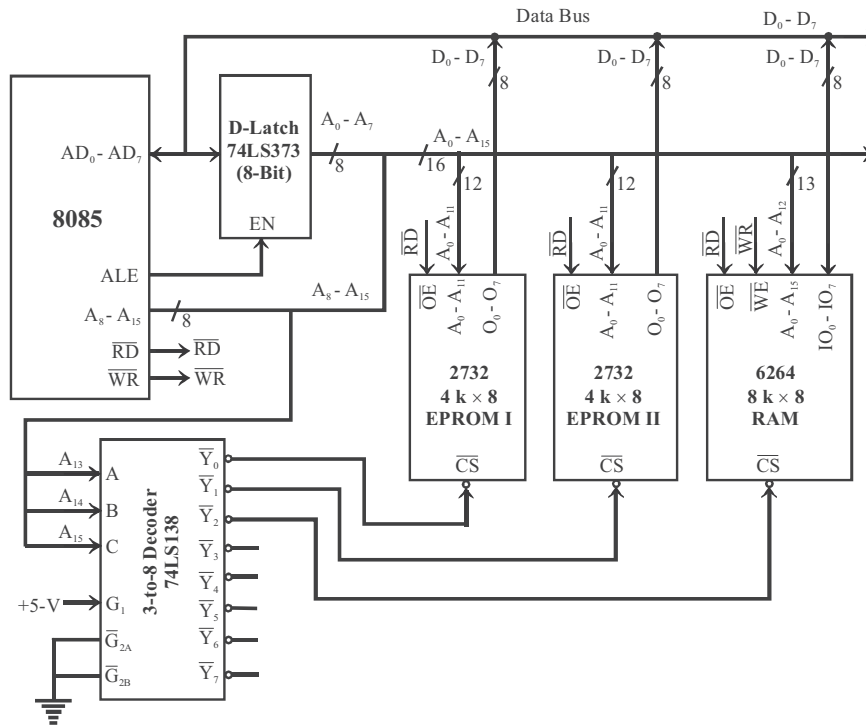
Memory mapping of IO device	IO mapping of IO device
1. 16-bit addresses are provided for IO devices.	1. 8-bit addresses are provided for IO devices.
2. The devices are accessed by memory read or memory write cycles.	2. The devices are accessed by IO read or IO write cycle. During these cycles, the 8-bit address is available on both low order address lines and high order address lines.
3. The IO ports or peripherals can be treated like memory locations and so all instructions related to memory can be used for data transfer between the IO device and the processor.	3. Only IN and OUT instructions can be used for data transfer between the IO device and the processor.
4. In memory-mapped ports, the data can be moved from any register to the ports and vice versa.	4. In IO-mapped ports, the data transfer can take place only between the accumulator and the ports.
5. When memory mapping is used for IO devices, the full memory address space cannot be used for addressing memory. Hence memory mapping is useful only for small systems, where the memory requirement is less.	5. When IO mapping is used for IO devices, then the full memory address space can be used for addressing the memory. Hence it is suitable for systems which requires a large memory capacity.
6. In memory-mapped IO devices, a large number of IO ports can be interfaced.	6. In IO mapping, only 256 ports ( $2^8 = 256$ ) can be interfaced.
7. For accessing memory-mapped devices, the processor executes the memory read or write cycle. During this cycle, $\text{IO}/\overline{\text{M}}$ is asserted <b>low</b> ( $\text{IO}/\overline{\text{M}}=0$ ).	7. For accessing the IO-mapped devices, the processor executes the IO read or write cycle. During this cycle, $\text{IO}/\overline{\text{M}}$ is asserted <b>high</b> ( $\text{IO}/\overline{\text{M}}=1$ ).

**DESIGN EXAMPLE - 1**

*Interface two numbers of 4 kb EPROM and one number of 8 kb RAM with 8085 processor. Explain the interface diagram and allocate binary addresses to memory ICs.*

**Solution**

The IC 2732 is selected for EPROM memory and the IC 6264 is selected for RAM memory. Both the memory IC's have time compatibility with 8085 processor.



**Fig. DE1 : Memory interface diagram for Design Example - 1.**

The 4 kb EPROM IC requires 12 address lines ( $2^{12} = 4 \text{ k}$ ). The 8 kb RAM IC requires 13 address lines ( $2^{13} = 8 \text{ k}$ ). The address lines  $A_0 - A_{11}$  are connected to both EPROM and RAM address input pins. The address lines  $A_{13}$ ,  $A_{14}$  and  $A_{15}$  are not used for memory address. Hence by decoding these address lines we can generate chip select signals.

The 3-to-8 decoder, 74LS138 is employed to produce the chip select signals for the system. The decoder has 8-output lines which can be used as 8-chip select signals. In this, three chip select signals are used for selecting memory ICs and the remaining five can be used for selecting other peripheral ICs in the system or for future expansion of the memory capacity. The interface diagram is shown in Fig. DE1. Address allotted to memory ICs are shown in Table-DE1.

The EPROM's are mapped in the beginning of memory space. The remaining addresses can be allotted to RAM's. The EPROM memory is mapped from  $0000_H$  to  $0FFF_H$  and  $2000_H$  to  $2FFF_H$ . The RAM memory is mapped from  $4000_H$  to  $5FFF_H$ .

**TABLE - DE1 : ADDRESS ALLOCATION TABLE FOR DESIGN EXAMPLE - 1**

Memory IC	Binary address															Hexa address	
	Decoder input			Input to memory address pins													
A <sub>15</sub>	A <sub>14</sub>	A <sub>13</sub>	A <sub>12</sub>	A <sub>11</sub>	A <sub>10</sub>	A <sub>9</sub>	A <sub>8</sub>	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>		
EPROM I 2732	0	0	0	X	0	0	0	0	0	0	0	0	0	0	0	0	0000
	0	0	0	X	0	0	0	0	0	0	0	0	0	0	0	1	0001
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
	0	0	0	X	1	1	1	1	1	1	1	1	1	1	1	1	0FFF
EPROM II 2732	0	0	1	X	0	0	0	0	0	0	0	0	0	0	0	0	2000
	0	0	1	X	0	0	0	0	0	0	0	0	0	0	0	1	2001
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
	0	0	1	X	1	1	1	1	1	1	1	1	1	1	1	1	2FFF
RAM 6264	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4000
	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	4001
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	5FFF

*Note : X indicates the unused address line for the particular memory IC and they are considered as zero.*

**DESIGN EXAMPLE - 2**

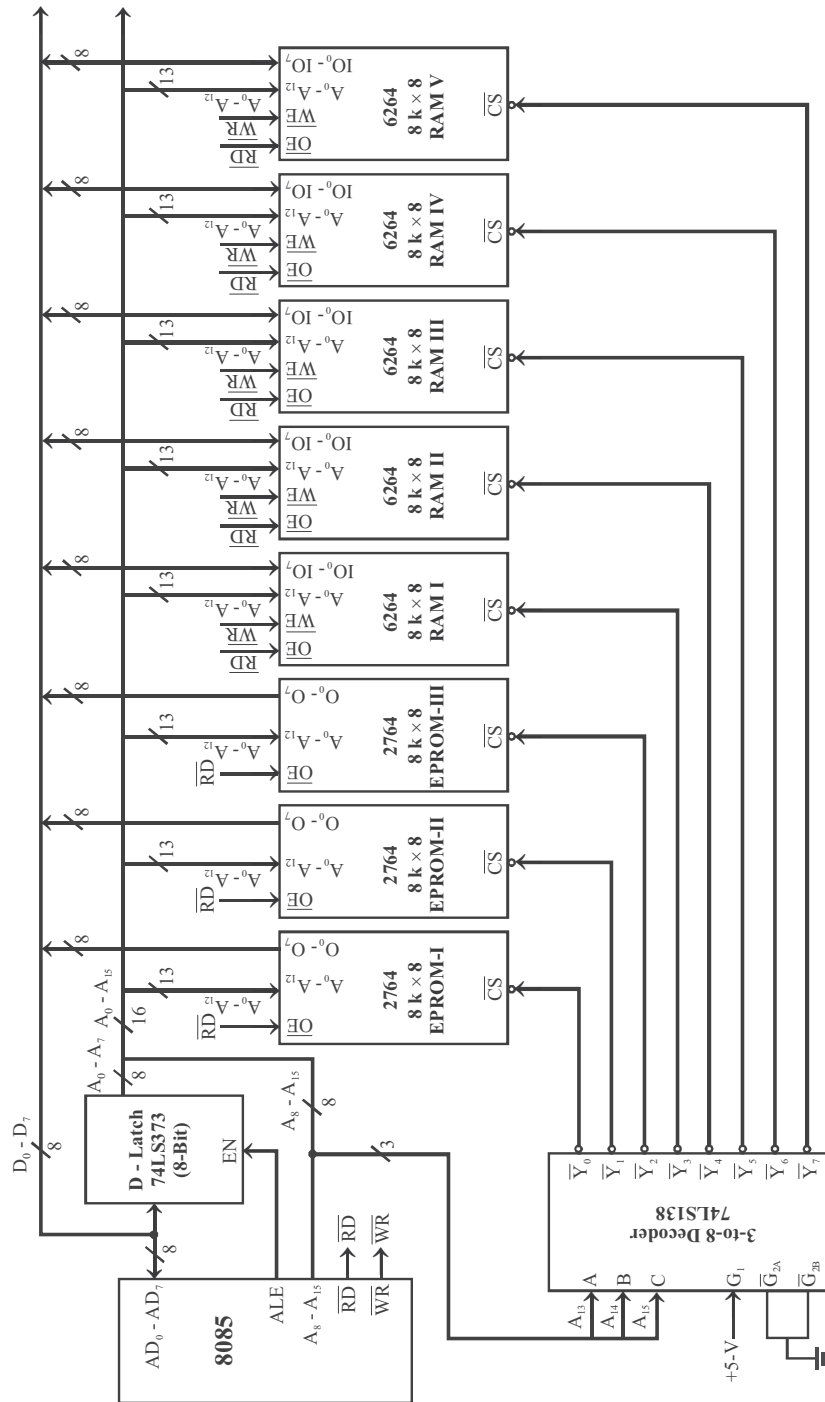
*Interface three numbers of 8 kb EPROM and 5 numbers of 8 kb static RAM to microprocessor 8085 to have a total memory capacity of 64 kb.*

**Solution**

The IC 2764 is selected for EPROM memory and the IC 6264 is selected for RAM memory. Both the memory ICs have time compatibility with the 8085 processor.

The 8 kb EPROM IC requires 13 address lines ( $2^{13} = 8 \text{ k}$ ). The 8 kb RAM IC also requires 13 address lines ( $2^{13} = 8 \text{ k}$ ). The address lines A<sub>0</sub>-A<sub>12</sub> are connected to all the EPROM's and RAMs. Hence A<sub>0</sub>-A<sub>12</sub> will select the required memory location. The address lines A<sub>13</sub>, A<sub>14</sub> and A<sub>15</sub> are not used for memory address. Hence by decoding these address lines we can generate chip select signals.

The 3-to-8 decoder, 74LS138 is employed to produce the chip select signals for the system. The decoder has 8-output lines which can be used as 8-chip select signals. All the 8-chip select signals are used to select memory ICs. EPROM's are mapped at the beginning of memory space. The decoder will select a memory IC by decoding the address lines A<sub>13</sub>, A<sub>14</sub> and A<sub>15</sub>. The address lines A<sub>0</sub>-A<sub>12</sub> will select a particular memory location in the selected IC. The interface diagram is shown in Fig. DE2 and address allocation table is shown in Table-DE2.



**TABLE - DE2 : ADDRESS ALLOCATION TABLE FOR DESIGN EXAMPLE- 2**

Memory IC chip	Binary address															Hexa address	
	Decoder input			Input to memory address pins													
	A <sub>15</sub>	A <sub>14</sub>	A <sub>13</sub>	A <sub>12</sub>	A <sub>11</sub>	A <sub>10</sub>	A <sub>9</sub>	A <sub>8</sub>	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>		A <sub>0</sub>
EPROM I	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0000
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0001
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0002
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1FFF
EPROM II	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	2000
	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	2001
	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	2002
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	3FFF
EPROM III	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4000
	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	4001
	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	4002
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	5FFF
RAM I	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	6000
	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	6001
	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	6002
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	7FFF
RAM II	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	8000
	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	8001
	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	8002
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	9FFF
RAM III	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	A000
	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	A001
	1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	A002
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	BFFF
RAM IV	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	C000
	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	C001
	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	C002
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	DFFF
RAM V	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	E000
	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	E001
	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	E002
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	FFFF

In this system the full memory capacity of 64 kb is utilized for memory. Hence the peripheral ICs and the IO ports should be IO-mapped in the system. The EPROM is mapped from  $0000_H$  to  $5FFF_H$ . The RAM is mapped from  $6000_H$  to  $FFFF_H$ . The EPROM capacity is 24 kb. The RAM capacity is 40 kb.

### DESIGN EXAMPLE - 3

*In a microprocessor system using 8085, the memory requirement is 8 kb EPROM and 8 kb RAM. For interfacing IO devices, three numbers of 8255 are required. Select suitable memories and explain how they are interfaced to the system. Interface the 8255 by memory mapping.*

#### Solution

The IC 2764 is selected for EPROM memory and the IC 6264 is selected for RAM memory. Both the memory IC's have time compatibility with 8085 processor.

The 8 kb EPROM, 2764 requires 13 address lines ( $2^{13} = 8\text{ k}$ ). The 8 kb RAM, 6264 also requires 13 address lines ( $2^{13} = 8\text{ k}$ ). The address lines  $A_0$  to  $A_{12}$  are connected to both EPROM and RAM memory ICs. The 8255 requires four internal addresses. Let us connect  $A_1$  of 8085 to  $A_0$  of 8255 and  $A_2$  of 8085 to  $A_1$  of 8255. The 8255 is memory-mapped in the system.

**Note :** The internal devices of 8255 can be selected by connecting any two address lines of the processor to  $A_0$  and  $A_1$  of 8255.

For the memories and 8255's we require 5 chip select signals. Hence we can use a 3-to-8 decoder 74LS138 for generating eight chip select signals by decoding the unused address lines  $A_{13}$ ,  $A_{14}$  and  $A_{15}$ . The decoder enabled pins are permanently tied to appropriate levels. In the eight chip select signals, five are used for selecting memory ICs and 8255 and the remaining three can be used for future expansion. The memory/8255 interface diagram is shown in Fig. DE3.

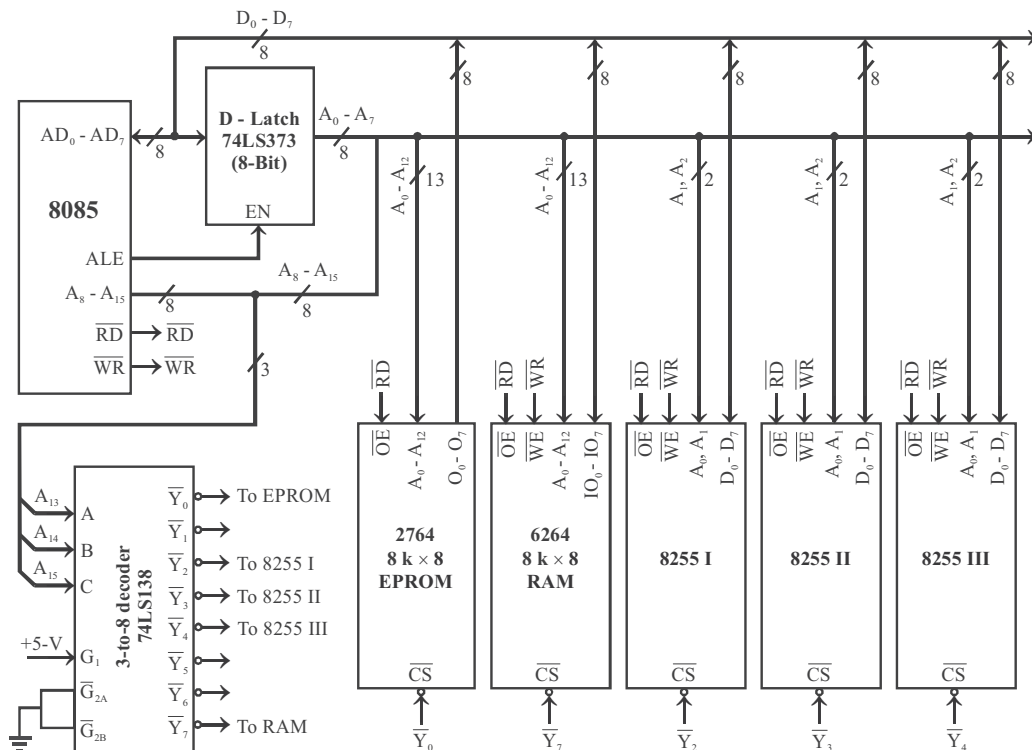


Fig. DE3 : Memory interface diagram for Design Example - 3.



**TABLE - DE3 : ADDRESS ALLOCATION TABLE FOR DESIGN EXAMPLE - 3**

Device	Binary address												Hexa address				
	Decoder input			Input to address pins of memory/8255													
A <sub>15</sub>	A <sub>14</sub>	A <sub>13</sub>	A <sub>12</sub>	A <sub>11</sub>	A <sub>10</sub>	A <sub>9</sub>	A <sub>8</sub>	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>		
2764 EPROM	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0000
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0001
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0002
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1FFF
6264 RAM	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	E000
	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	E001
	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	E002
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	FFFF
8255 I Port-A Port-B Port-C Control register	0	1	0	X	X	X	X	X	X	X	X	X	X	0	0	X	4000
	0	1	0	X	X	X	X	X	X	X	X	X	X	0	1	X	4002
	0	1	0	X	X	X	X	X	X	X	X	X	X	1	0	X	4004
	0	1	0	X	X	X	X	X	X	X	X	X	X	1	1	X	4006
8255 II Port-A Port-B Port-C Control register	0	1	1	X	X	X	X	X	X	X	X	X	X	0	0	X	6000
	0	1	1	X	X	X	X	X	X	X	X	X	X	0	1	X	6002
	0	1	1	X	X	X	X	X	X	X	X	X	X	1	0	X	6004
	0	1	1	X	X	X	X	X	X	X	X	X	X	1	1	X	6006
8255 III Port-A Port-B Port-C Control register	1	0	0	X	X	X	X	X	X	X	X	X	X	0	0	X	8000
	1	0	0	X	X	X	X	X	X	X	X	X	X	0	1	X	8002
	1	0	0	X	X	X	X	X	X	X	X	X	X	1	0	X	8004
	1	0	0	X	X	X	X	X	X	X	X	X	X	1	1	X	8006

*Note : The X indicates that the address line is not used for the particular device and they are considered as zero.*

The EPROM is mapped at the starting of memory space. The RAM is mapped at the end of memory space. The EPROM is mapped from 0000<sub>H</sub> to 1FFF<sub>H</sub>. The RAM is mapped from E000<sub>H</sub> to FFFF<sub>H</sub>. The four internal devices of 8255 are control register, port-A, port-B and port-C. A 16-bit address is allotted to each internal device of 8255 as shown in Table-DE3.

#### DESIGN EXAMPLE - 4

*Interface 2 kb RAM and 256 × 8 ROM with 8085 processor to satisfy the total memory requirement of 8 kb RAM and 1 kb ROM.*

#### Solution

The memory requirement of 8 kb RAM can be achieved with 4 numbers of 2 kb RAM. The memory requirement of 1kb ROM can be achieved with 4 numbers of 256 × 8 ROM. ( $4 \times 256 = 1024 = 1k$ ). The 2 kb RAM requires 11 address lines ( $2^{11} = 2k$ ). The 256 × 8 ROM requires 8 address lines ( $2^8 = 256$ ). The address

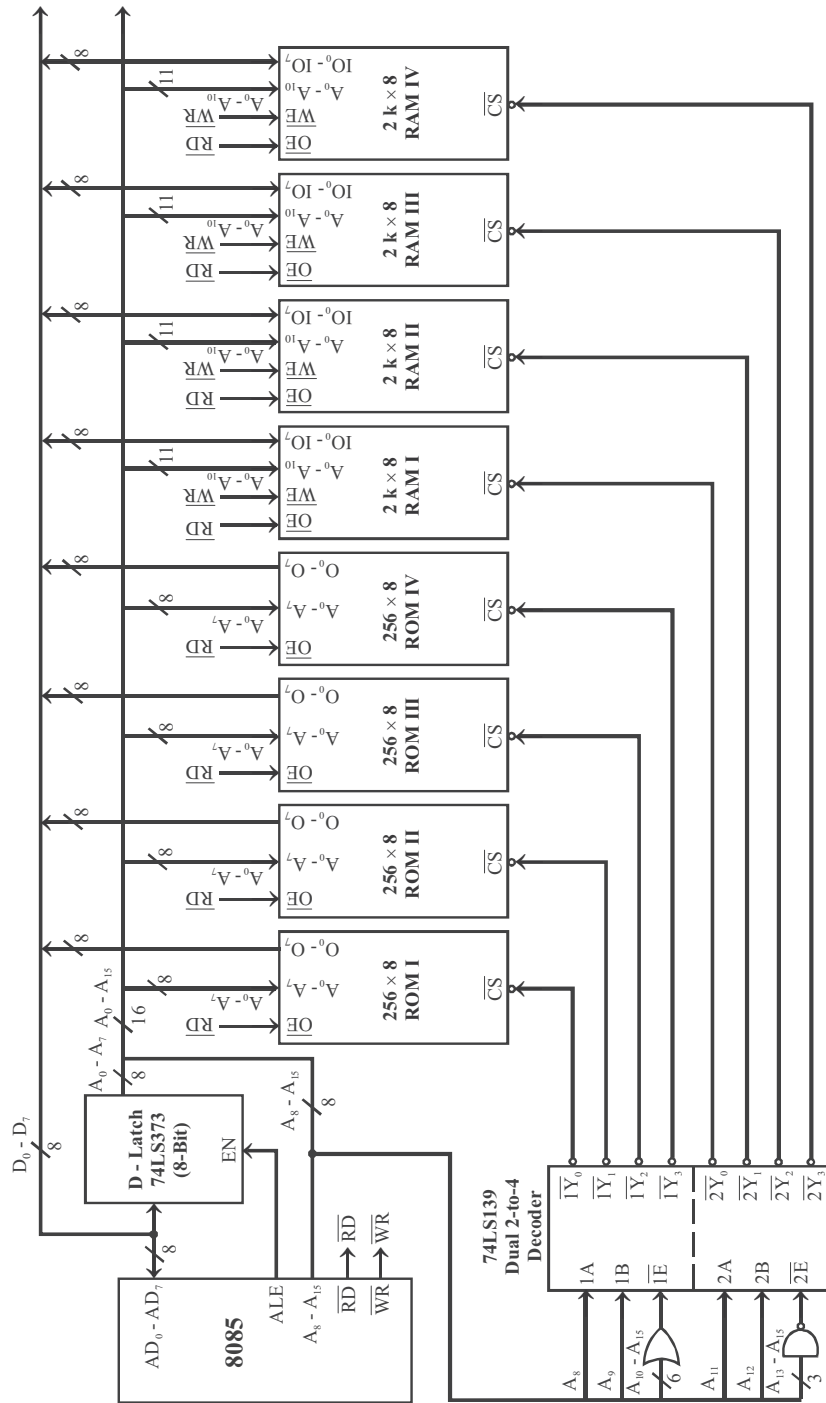


Fig. DE4 : Memory interface diagram for Design Example - 4.

**TABLE - DE4 : ADDRESS ALLOCATION TABLE FOR DESIGN EXAMPLE-4**

Device	Binary address												Hexa address				
	ROM decoder enable				Decoder input		Input to ROM memory address pins										
$256 \times 8$ ROM I	$A_{15}$	$A_{14}$	$A_{13}$	$A_{12}$	$A_{11}$	$A_{10}$	$A_9$	$A_8$	$A_7$	$A_6$	$A_5$	$A_4$	$A_3$	$A_2$	$A_1$	$A_0$	0000
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0001
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0002
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	00FF
$256 \times 8$ ROM II	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0100
	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0101
	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0102
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	01FF
$256 \times 8$ ROM III	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0200
	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0201
	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0202
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
	0	0	0	0	0	0	1	0	1	1	1	1	1	1	1	1	02FF
$256 \times 8$ ROM IV	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0300
	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1	0301
	0	0	0	0	0	0	1	1	0	0	0	0	0	0	1	0	0302
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	03FF
Device	Binary address												Hexa address				
	RAM decoder enable			Decoder input		Input to RAM memory address pins											
$2\text{ k} \times 8$ RAM I	$A_{15}$	$A_{14}$	$A_{13}$	$A_{12}$	$A_{11}$	$A_{10}$	$A_9$	$A_8$	$A_7$	$A_6$	$A_5$	$A_4$	$A_3$	$A_2$	$A_1$	$A_0$	E000
	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	E001
	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	E002
	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	$\vdots$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$2\text{ k} \times 8$ RAM II	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	E7FF
	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	E800
	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	1	E801
	1	1	1	0	1	0	0	0	0	0	0	0	0	0	1	0	E802
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$2\text{ k} \times 8$ RAM III	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	FFFF
	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	F000
	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	F001
	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	0	F002
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$2\text{ k} \times 8$ RAM IV	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	F7FF
	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	F800
	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	F801
	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	0	F802
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$2\text{ k} \times 8$ RAM IV	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	FFFF

lines  $A_0 - A_{10}$  are connected to RAM ICs. Hence, they will select the required memory location in that ICs. The address lines  $A_0 - A_7$  are connected to ROM ICs. Hence, they will select the required memory location in those ICs.

Totally there are 8-memory ICs hence we require 8-chip select signals. The 8-chip select signals can be generated by using a dual 2-to-4 decoder 74LS139. One of the 2-to-4 decoder is used to generate chip select signals for ROM memory ICs and the other decoder is used to generate chip select signals for RAM memory ICs. The address lines  $A_8$  and  $A_9$  are used to generate chip select signals for ROM memory. The address lines  $A_{10}$  to  $A_{15}$  are logically ORed and used as enable for ROM decoder. The address lines  $A_{11}$  and  $A_{12}$  are used to generate chip select signals for RAM memory. The address lines  $A_{13}$  to  $A_{15}$  are logically NANDed and used as enable for RAM decoder.

ROM memories are mapped in the beginning of memory space. The RAM memories are mapped at the end of memory space. The ROM memories are mapped from  $0000_H$  to  $03FF_H$ . The RAM memories are mapped from  $E000_H$  to  $FFFF_H$ .

### DESIGN EXAMPLE - 5

*A system requires 16 kb EPROM and 16 kb RAM. Also the system has 2 numbers of 8255, one number of 8279, one number of 8251 and one number of 8254.*

*(8255 - Programmable peripheral interface, 8279-Keybaord/display controller, 8251 - USART and 8254 - Timer)*

*Draw the Interface diagram. Allocate addresses to all the devices. The peripheral IC's should be IO-mapped.*

### Solution

The IO devices in the system should be mapped by standard IO mapping. Hence separate decoders can be used to generate chip select signals for memory IC's and peripheral IC's.

For 16 kb EPROM, we can provide 2 numbers of 2764 ( $8\text{ k} \times 8$ ) EPROM. For 16 kb RAM we can provide 2 numbers of 6264 ( $8\text{ k} \times 8$ ) RAM.

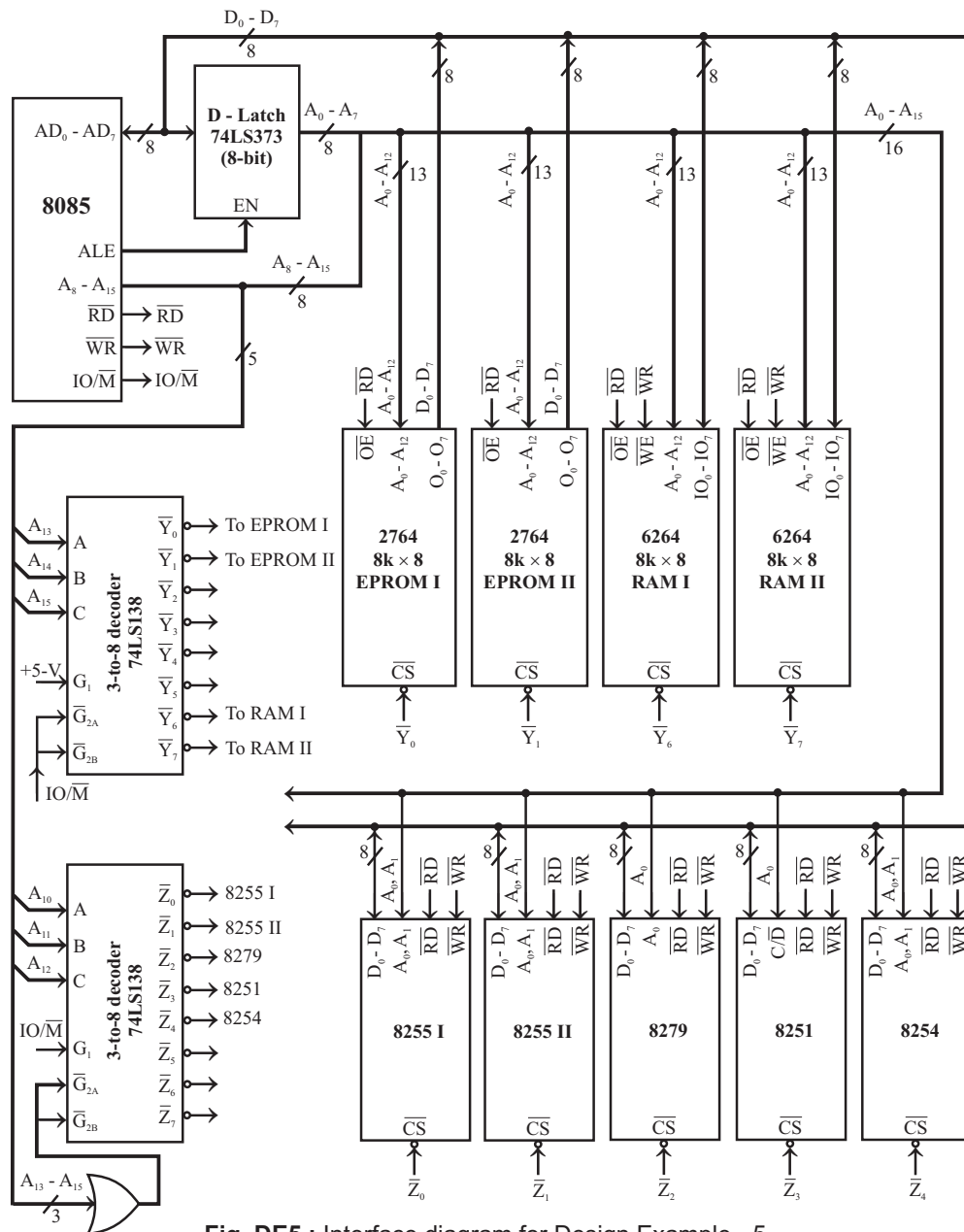
The 8 kb memory requires 13 address lines ( $2^{13} = 8\text{ k}$ ). Hence the address lines  $A_0 - A_{12}$  are used for selecting the memory locations. The unused address lines  $A_{13}$ ,  $A_{14}$  and  $A_{15}$  are used as input to decoder 74LS138 (3-to-8-decoder) of memory IC. The logic **low** enables of this decoder are tied to  $\overline{IO/\overline{M}}$  of 8085, so that this decoder is enabled for memory read/write operation. The other enable pins of decoder are tied to appropriate logic levels permanently. The 4 outputs of the decoder are used to select memory IC's and the remaining 4 are kept for future expansion.

The EPROM is mapped in the beginning of memory space from  $0000_H$  to  $3FFF_H$ . The RAM is mapped at the end of memory space from  $C000_H$  to  $FFFF_H$ .

There are five peripheral IC's to be interfaced to the system. The chip select signals for these IC's are given through another 3-to-8 decoder 74LS138 (IO decoder). The input to this decoder is  $A_{10}$ ,  $A_{11}$  and  $A_{12}$ . The address lines  $A_{13}$ ,  $A_{14}$  and  $A_{15}$  are logically ORed and applied to **low** enable of IO decoder. The logic **high** enable of IO decoder is tied to  $\overline{IO/\overline{M}}$  signal of 8085, so that this decoder is enabled for IO read/write operation.

Here, the high order address lines can be used for decoding because the processor outputs the 8-bit port address both on  $AD_0$  to  $AD_7$  and  $A_8$  to  $A_{15}$ . The address lines  $A_0$  and  $A_1$  are used to select the internal devices of the peripheral ICs. The output of the decoder are used to select the ICs. Three outputs of the decoder will be spare for future expansion.

*Note : Since the IO devices are IO-mapped in the system, 8-bit addresses have been allotted to them.*



**Fig. DE5 :** Interface diagram for Design Example - 5.

**TABLE - DE5 : ADDRESS ALLOCATION TABLE FOR DESIGN EXAMPLE- 5**

Device	Binary address															Hexa address	
	Input to memory decoder			Input to memory address pins													
A <sub>15</sub>	A <sub>14</sub>	A <sub>13</sub>	A <sub>12</sub>	A <sub>11</sub>	A <sub>10</sub>	A <sub>9</sub>	A <sub>8</sub>	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>		
2764 I 8 k × 8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0000
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1FFF
2764 II 8 k × 8	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	2000
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	3FFF
6264 I 8 k × 8	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	C000
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	DFFF
6264 II 8 k × 8	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	E000
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	FFFF
Device		Binary address															Hexa address
		IO decoder enable			IO decoder input			Input to IO device address pins									
A <sub>15</sub> A <sub>7</sub>	A <sub>14</sub> A <sub>6</sub>	A <sub>13</sub> A <sub>5</sub>	A <sub>12</sub> A <sub>4</sub>	A <sub>11</sub> A <sub>3</sub>	A <sub>10</sub> A <sub>2</sub>	A <sub>9</sub> A <sub>1</sub>	A <sub>8</sub> A <sub>0</sub>										
8255 I																	
Port-A		0	0	0	0	0	0	0	0			0	0			00	
Port-B		0	0	0	0	0	0	0	0			0	1			01	
Port-C		0	0	0	0	0	0	0	0			1	0			02	
Control register		0	0	0	0	0	0	0	0			1	1			03	
8255 II																	
Port-A		0	0	0	0	0	0	1	0			0	0			04	
Port-B		0	0	0	0	0	0	1	0			0	1			05	
Port-C		0	0	0	0	0	0	1	0			1	0			06	
Control register		0	0	0	0	0	0	1	0			1	1			07	
8279																	
Data register		0	0	0	0	0	1	0				X	0			08	
Control register		0	0	0	0	0	1	0				X	1			09	
8251																	
Data register		0	0	0	0	0	1	1				X	0			0C	
Control register		0	0	0	0	0	1	1				X	1			0D	
8254																	
Counter-0		0	0	0	1	0	0	0				0	0			10	
Counter-1		0	0	0	1	0	0	0				0	1			11	
Counter-2		0	0	0	1	0	0	0				1	0			12	
Control register		0	0	0	1	0	0	0				1	1			13	

*Note : Don't care (X) is considered as zero.*

## DESIGN EXAMPLE - 6

*In a microprocessor-based system 8085, 8 kb EPROM and 8 kb RAM are needed. For interfacing IO devices two numbers of 8155 are required. Select suitable memories and explain how they are interfaced in the system. Interface the 8155 ports by IO mapping.*

### Solution

The IC 2764 (8 k  $\times$  8) is selected for EPROM memory and IC 6264 (8 k  $\times$  8) is selected for RAM memory. Both the memory IC's have time compatibility with 8085 processor.

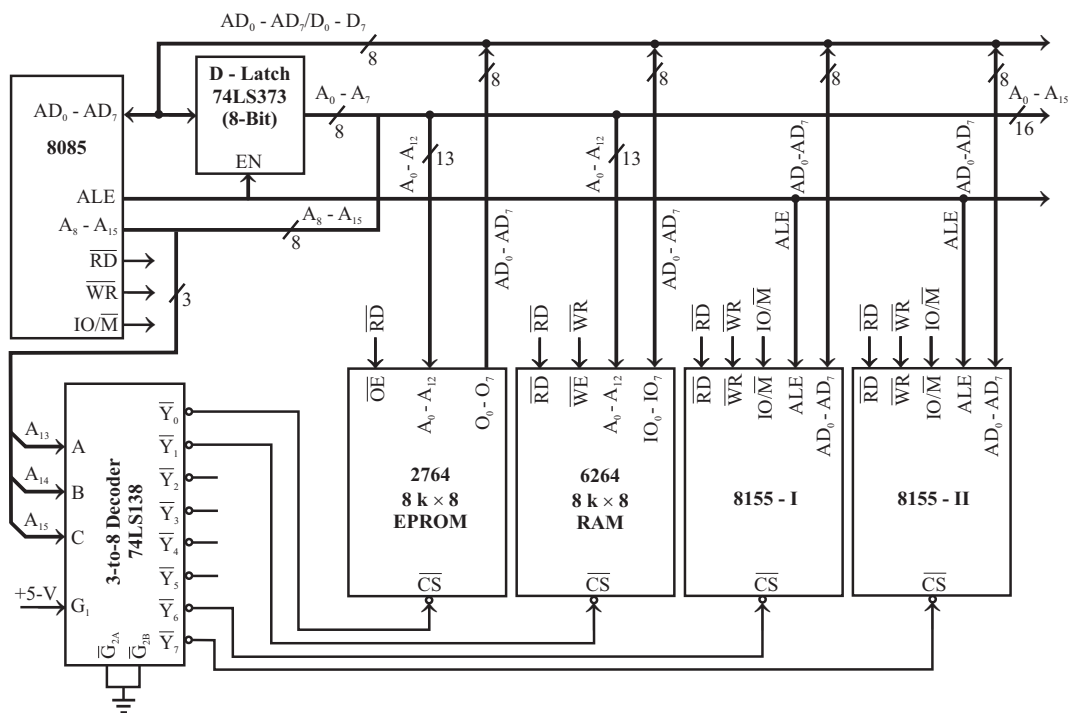
The 8kb memories require 13 address lines ( $2^{13} = 8 \text{ k}$ ). Hence, the address lines  $A_0 - A_{12}$  are used to select memory locations.

In addition to 6264, each one of the 8155 chip provides a static RAM capacity of 256 bytes. The RAM locations of 8155 are selected by address lines  $A_0$ - $A_6$ .

A 3-to-8 decoder, 74LS138 is used for generating chip select signals by decoding the address lines  $A_{13}$ ,  $A_{14}$  and  $A_{15}$ .

The 8155 has internal address latch and decoder to differentiate memory operation and IO operation. To utilize this facility, the control signals ALE and  $\overline{\text{IO}/\overline{\text{M}}}$  are connected to 8155.

The 8155 ports and memory locations can be selected from the decoder used for memory devices. It differentiates the memory and IO operation from  $\text{IO}/\overline{\text{M}}$  signal. Eight bit addresses are allotted to ports of 8155 and sixteen bit addresses are allotted to RAM memory locations of 8155.



**Fig. DE6 :** Interface diagram for Design Example - 6.

**TABLE - DE6 : ADDRESS ALLOCATION TABLE FOR DESIGN EXAMPLE - 6**

Device	Binary address															Hexa address	
	Decoder input			Input to address pins of memory/8155													
A <sub>15</sub>	A <sub>14</sub>	A <sub>13</sub>	A <sub>12</sub>	A <sub>11</sub>	A <sub>10</sub>	A <sub>9</sub>	A <sub>8</sub>	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>		
2764 8k × 8 EPROM	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0000
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0001
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1FFF
6264 8k × 8 RAM	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	2000
	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	2001
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	3FFF
8155 I  RAM 256 × 8  Control register Port-A Port-B Port-C LSB timer MSB timer	1	1	0	X	X	X	X	X	0	0	0	0	0	0	0	0	C000
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
	1	1	0	X	X	X	X	X	1	1	1	1	1	1	1	1	C0FF
	1	1	0	X	X	0	0	0									C0
	1	1	0	X	X	0	0	1									C1
	1	1	0	X	X	0	1	0									C2
	1	1	0	X	X	0	1	1									C3
	1	1	0	X	X	1	0	0									C4
	1	1	0	X	X	1	0	1									C5
	1	1	1	X	X	X	X	X	0	0	0	0	0	0	0	0	E000
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
	1	1	1	X	X	X	X	X	1	1	1	1	1	1	1	1	E0FF
Control register Port-A Port-B Port-C LSB timer MSB timer	1	1	1	X	X	0	0	0									E0
	1	1	1	X	X	0	0	1									E1
	1	1	1	X	X	0	1	0									E2
	1	1	1	X	X	0	1	1									E3
	1	1	1	X	X	1	0	0									E4
	1	1	1	X	X	1	0	1									E5

*Note : Don't care (X) is considered as zero.*



### 3.11 SUMMARY

---

- The function of the memory is to store programs and data.
- The memories of a microcomputer system can be classified into processor memory, primary or main memory and secondary memory.
- The processor memory are a set of microprocessor registers.
- Primary or main memory is the storage area that is directly accessed by the microprocessor.
- The primary memories are semiconductor devices.
- The semiconductor memories are ROM, EPROM, EEPROM, static RAM, DRAM and NVRAM.
- The typical semiconductor memory IC will have  $n$  address pins and  $m$  data pin. The capacity of such a memory will be  $2^n \times m$  bits.
- The common features of semiconductor memories are random access and non-destructive readout.
- The ROM is read only memory and it is non-volatile.
- The memory cell of ROM memory will have a MOS transistor either with open gate (representing 0) or closed gate (representing 1).
- The EPROM is a semiconductor read only memory capable of reprogramming.
- The EPROM is non-volatile and has random access feature.
- The EPROM can be erased by passing UV light.
- The memory cell of EPROM consist of a MOS transistor with isolated gate.
- Read/write semiconductor memories are called RAM.
- The RAM memories are volatile and have random access feature.
- The memory cell in static RAM is a flip-flop made of 6 to 8 MOS transistors.
- The DRAM's are read/write memories in which the information is stored in the form of electric charge on the gate to substrate capacitance of a MOS transistor.
- The contents of DRAM's has to be refreshed periodically by the refreshing circuit.
- Large capacity DRAM's are cheaper than static RAM.
- The various types of non-volatile RAM are flash memory, EEPROM and shadow RAM.
- The primary function of memory interfacing is that the microprocessor should be able to read and write into a set of semiconductor memory IC chip.
- Memory interfacing deals with choosing memories with suitable access time, generating chip select signal and control signals for read/write operation.
- A memory IC with  $n$  address pins will have  $2^n$  memory location.
- The EPROM is used for storing monitor program, other permanent programs and data.
- The RAM memory is used for stack, temporary program and data storage.
- In 8085 system, the EPROM is mapped at the beginning of memory space in order to achieve automatic execution of monitor program after reset.
- Monitor program is a permanent program written by system designer for initializing system parameters.
- The decoders are used for generating chip select signals.
- The decoders will have  $n$  input lines and  $2^n$  output lines. In logic low decoder, at anyone time one of the  $2^n$  output will remain low and all other output will remain high.
- The three major types of data transfer between the microprocessor and IO device are programmed IO, interrupt driven IO and direct memory access.
- The IO devices are slow devices and so they are connected to the system bus through ports.
- The various INTEL IO port devices are 8212, 8155/8156, 8255, 8355 and 8755.
- The 8212 can be used as input or output device and the function is determined by the mode pin.

- INTEL 8155 has  $256 \times 8$  static RAM, 2 numbers of 8-bit parallel IO port, 1 number of 6-bit parallel IO port and 14-bit timer.
- INTEL 8255 consists of 3 numbers of 8-bit parallel IO port.
- INTEL 8255 requires 4 internal addresses and has logic **low** chip select pin.
- The two ways of interfacing IO devices to 8085 processor are memory mapping of IO device or standard IO mapping.
- In memory mapping of IO devices a 16-bit address is allotted to an IO device.
- In standard IO mapping an 8-bit address is allotted to each IO device.
- When a device is memory-mapped in the system, the processor will access the device like a memory location.
- The 8085 processor differentiates the memory address and IO address using the signal  $IO/\bar{M}$ .

### 3.12 SHORT QUESTIONS AND ANSWERS

---

3.1 *What is memory ?*

A memory is a storage device in a microprocessor-based system and its primary function is to store programs and data.

3.2 *Why are semiconductor memories used as main memory in microprocessor system ?*

Semiconductor memories have processor compatible access time for read and write operations, they are used as main memories.

3.3 *What are the different types of semiconductor memory ?*

The different types of semiconductor memory are RAM, PROM, EPROM, static RAM, DRAM and NVRAM.

3.4 *List the features of semiconductor memories.*

1. Semiconductor memories are random access memories.
2. In semiconductor memories, a read operation by the processor will not destroy the stored information.
3. The read and write time of the semiconductor memory is compatible for the microprocessor.

3.5 *What is meant by volatile and non-volatile memories ?*

If the information stored in the memory is lost when the power supply is switched OFF, then the memory is called volatile.

If the content of memory is preserved even after the power supply is switched OFF, then the memory is called non-volatile.

3.6 *List the volatile and non-volatile semiconductor memories.*

The volatile semiconductor memories are static RAM and DRAM. The non-volatile semiconductor memories are ROM, PROM, EPROM and NVRAM.

3.7 *What are the characteristics of **ROM** memory ?*

The characteristics of ROM are as follows:

1. It is non-volatile memory.
2. The contents of ROM can be read by the processor but it cannot write into it.
3. The ROM has the feature of random access.
4. The memory cell has a MOS transistor either with an open gate or a closed gate.

### 3.8 How are ROMs classified?

ROMs can be classified into the following three categories based on the method of programming.

1. Custom programmed or Mask programmed ROM
2. Programmable or Field programmable ROM
3. Reprogrammable or Erasable - programmable ROM.

### 3.9 List the characteristics of EPROM.

1. The EPROM is non-volatile.
2. It has random-access feature.
3. The contents of EPROM can be erased by passing UV light and then the device can be reprogrammed.
4. The EPROM is read only memory and for writing into EPROM, a separate hardware set up is required.

### 3.10 Write a short note on the memory cell of EPROM.

The memory cell of EPROM contains a MOS transistor with isolated gate. The isolated gate is located between the normal control gate and the source/drain region of transistor. The information is stored as a charge or no charge in the floating gate.

### 3.11 What is NVRAM?

The non-volatile read/write memories are called NVRAM. The various types of NVRAMs are flash memory, EEPROM and Shadow RAM.

### 3.12 List the features of static RAM.

1. Static RAMs are read/write memories.
2. They are volatile and have random access feature.
3. The memory cell is a flip-flop constructed using 6 to 8 MOS transistors.

### 3.13 What is DRAM ?

DRAMs are read/write semiconductor memories in which the information is stored in the form of electric charge on the gate to substrate capacitance of a MOS transistor.

### 3.14 List the characteristics of DRAM.

1. The DRAMs are volatile and have random access feature.
2. They are read/write memories.
3. The contents of DRAM have to be refreshed periodically using refreshing circuits.
4. The memory cell of DRAM will have 3 to 4 MOS transistor.

### 3.15 Compare the Static RAM with DRAM.

Static RAM	DRAM
<ol style="list-style-type: none"> <li>1. Information is stored as voltage level in a flip-flop.</li> <li>2. Six to eight transistors are required to form one memory cell.</li> <li>3. Packing density is low.</li> <li>4. The contents of memory need not be refreshed.</li> </ol>	<ol style="list-style-type: none"> <li>1. Information is stored as a charge in the gate to substrate capacitance.</li> <li>2. Three to four transistors are required to form one memory cell.</li> <li>3. Packing density is high.</li> <li>4. The contents of memory has to be refreshed periodically.</li> </ol>

*3.16 What is physical memory space?*

The memory locations that are directly addressed by the microprocessor is called physical memory space.

*3.17 What is memory word size?*

The size of data that can be stored in memory location is called memory word size.

*3.18 What is meant by memory mapping?*

Memory mapping is the process of interfacing memories to microprocessor and allocating addresses to each memory location.

*3.19 What is memory access time?*

Memory access time is the time taken by the processor to read or write a memory location. Read operation is the time between a valid address on the bus and the end of read control signal. Write operation is the time between a valid address on the bus and the end of write control signal.

*3.20 What are the factors to be considered while selecting a semiconductor memory for a microprocessor system ?*

The following are the factors to be considered while selecting a semiconductor memory IC:

- Capacity and organization (Memory word size).
- Timings of various signals.
- Power consumption and bus loading (Current levels).
- Physical dimensions and packaging.
- Cost, reliability and availability.

*3.21 What is bus contention?*

If two devices drive the data bus simultaneously then it is called bus contention. It may lead to following undesirable events:

1. Damaging one or both the IC chip.
2. The high current may cause a voltage spike in the supply system leading to data loss.

*3.22 Why is EPROM mapped at the beginning of memory space in 8085 ?*

When EPROM is mapped at the beginning of memory space, then 0000<sub>H</sub> address will be allotted to EPROM. The monitor program can be stored from 0000<sub>H</sub> address. Whenever the processor is reset, the program counter will be cleared (i.e it will have 0000<sub>H</sub> address) and the monitor program will be executed automatically.

*3.23 What is chip select and how it is generated?*

Chip select is the control signal that has to be asserted TRUE to bring an IC from **high impedance** state to normal state. Generally the chip select signals are generated in a system by decoding the unused address lines with the help of decoders.

*3.24 What are the typical control signals involved in EPROM interfacing ?*

The control signals needed for EPROM are chip select and output enable.

*3.25 What are the typical control signals involved in RAM interfacing ?*

The control signals needed for RAM interfacing are chip enable, output enable and write enable.

3. 26 *What is programmed IO ?*

If the data transfer between an IO device and the processor is accomplished through an IO port and controlled by a program then the IO device is called programmed IO.

3.27 What is interrupt IO?

If the IO device initiates the data transfer through interrupt, then the IO is called interrupt driven IO.

3.28 What is DMA?

The direct data transfer between the IO device and the memory is called DMA.

3.29 *What is the need for Port?*

IO devices are generally slow devices and their timing characteristics do not match with processor timings. Hence, the IO devices are connected to a system bus through the ports.

3.30 What is a port?

A port is a buffered I/C which is used to hold the data transmitted from the microprocessor to IO device or vice versa.

3.31 Give some examples of port devices used in a 8085 microprocessor-based system.

The various INTEL IO port devices used in 8085 microprocessor-based system are 8212, 8155, 8156, 8255, 8355 and 8755.

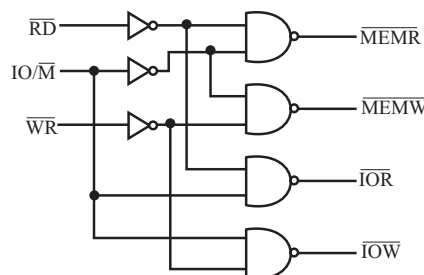
3.32 Write a short note on INTEL 8255?

The INTEL 8255 is a IO port device consisting of 3 numbers of 8-bit parallel IO ports. The ports can be programmed to function either as an input port or as an output port in different operating modes. It requires 4 internal addresses and has one logic **low** chip select pin.

3.33 What are the different methods of interfacing IO devices to 8085-based system.

There are two methods of interfacing IO devices to 8085 system. They are memory mapping of IO device and standard IO mapping.

3.34 Draw a simple circuit to decode the three control signals  $\overline{RD}$ ,  $\overline{WR}$  and  $IO/\overline{M}$  and to produce separate read/write control signals for memory and IO devices.



**Fig. Q3.34 :** Circuit to generate separate read and write signals for memory and IO devices.

*3.35 Compare the memory-mapped IO with the standard IO-mapped IO.*

Memory-mapped IO	Standard IO-mapped IO
<ol style="list-style-type: none"><li>1. Sixteen bit address is allotted to an IO device.</li><li>2. The devices are accessed by memory read or memory write cycle.</li><li>3. All instructions related to memory can be used for data.</li><li>4. A large number of IO ports can be interfaced.</li></ol>	<ol style="list-style-type: none"><li>1. Eight bit address is allotted to an IO device.</li><li>2. The devices are accessed by IO read or IO write cycle.</li><li>3. Only IN and OUT instructions can be used for data transfer.</li><li>4. Only 256 ports can be interfaced.</li></ol>

*3.36 What is the drawback in memory-mapped IO?*

When IO devices are memory-mapped, some of the addresses are allotted to IO devices. So the full address space cannot be used for addressing memory (i.e., physical memory address space will be reduced). Hence, memory mapping is useful only for small systems, where the memory requirement is less.

---

## INTERRUPT STRUCTURE

### 4.1 INTERRUPT AND ITS NEED

Microprocessors allow normal program execution to be interrupted in order to carry out a specific task/work. A processor can be interrupted in the following ways :

- (i) by an external signal generated by a peripheral,
- (ii) by an internal signal generated by a special instruction in the program,
- (iii) by an internal signal generated due to an exceptional condition which occurs while executing an instruction. (For example, in 8086 processor, 'divide by zero' is an exceptional condition which initiates type-0 interrupt and such an interrupt is also called exception.)

In general, the process of interrupting the normal program execution to carry out a specific task/work is referred to as interrupt.

The interrupt is initiated by a signal generated by an external device or by a signal generated internal to the processor. When a microprocessor receives an interrupt signal, it stops executing the current normal program, saves the status (or content) of various registers (PC in case of 8085) in stack and then executes a subroutine/procedure in order to perform the specific task/work requested by the interrupt. The subroutine/procedure that is executed in response to an interrupt is also called **Interrupt Service Routine (ISR)**. At the end of ISR, the stored status of registers in stack are restored to respective registers and the processor resumes the normal program execution from the point (instruction) where it was interrupted.

The external interrupts are used to implement interrupt driven data transfer scheme. The interrupts generated by special instructions are called software interrupts and they are used to implement system services/calls (or monitor services/calls). The system/monitor services are procedures developed by the system designer for various operations and stored in memory. The user can call these services through software interrupts. The interrupts generated by exceptional conditions are used to implement error conditions in the system.

#### Interrupt Driven Data Transfer Scheme

Interrupts are useful for efficient data transfer between the processor and the peripheral. When a peripheral is ready for data transfer, it interrupts the processor by sending an appropriate signal. Upon receiving an interrupt signal, the processor suspends the current program execution, saves the status in a stack and executes an ISR to perform the data transfer between the peripheral and the processor. At the end of ISR the processor status is restored from stack and the processor resumes its normal program execution. This type of data transfer scheme is called interrupt driven data transfer scheme.

The data transfer between the processor and peripheral devices can be implemented either by polling technique or by interrupt method. In polling technique, the processor has to periodically poll or check the status/readiness of the device and can perform data transfer only when the device is ready. In polling technique the processor time is wasted, because the processor has to suspend its work and check the status of the device in predefined intervals.

Alternatively, if the device interrupts the processor to initiate a data transfer whenever it is ready then the processor time is effectively utilized because the processor need not suspend its work and check the status of the device in predefined intervals.

For example, consider the data transfer from a keyboard to the processor. Normally a keyboard has to be checked by the processor once in every 10 millisecond for a key press. Therefore, once in every 10 milliseconds the processor has to suspend its work and then check the keyboard for a valid key code. Alternatively, the keyboard can interrupt the processor, whenever a key is pressed and a valid key code is generated. In this way the processor need not waste its time to check the keyboard once in every 10 milliseconds.

#### 4.2 CLASSIFICATION OF INTERRUPTS

---

In general interrupts can be classified in the following three ways:

- Hardware and software interrupts.
- Vectored and non-vectored interrupts.
- Maskable and non-maskable interrupts.

Interrupts initiated by external hardware by sending an appropriate signal to the interrupt pin of the processor is called hardware interrupt. The 8085 processor has five interrupt pins TRAP, RST 7.5, RST 6.5, RST 5.5 and INTR and the interrupts initiated by applying appropriate signal to these pins are called hardware interrupts of 8085.

Software interrupts are program instructions. These instructions are inserted at desired locations in a program. While running a program, if a software interrupt instruction is encountered then the processor initiates an interrupt. The 8085 processor has 8 types of software interrupts. The software interrupt instruction is INT n, where n is the type number in the range 0 to 7.

When an interrupt signal is accepted by the processor, and the program control automatically branches to a specific address (called vector address) then the interrupt is called vectored interrupt. The automatic branching to a vector address is predefined by the manufacturer of the processor. (In these vector addresses the interrupt service subroutines(ISR) are stored.) In non-vectored interrupts the interrupting device should supply the address of the ISR to be executed in response to the interrupt. All the 8085 interrupts excepts INTR are vectored interrupts.

The processors have the facility for accepting or rejecting hardware interrupts. Programming the processor to reject an interrupt is referred to as masking or disabling and programming the processor to accept an interrupt is referred to as unmasking or enabling. In 8085 the hardware



interrupts RST 7.5, RST 6.5, and RST 5.5 can be masked/unmasked using SIM instruction. All the hardware interrupts except TRAP are disabled by executing DI instruction and they are enabled by executing EI instruction.

The interrupts whose request can be either accepted or rejected by the processor are called maskable interrupts. The interrupts whose request has to be definitely accepted (i.e., it cannot be rejected) by the processor are called non-maskable interrupts. Whenever a request is made by a non-maskable interrupt, the processor has to definitely accept that request and service that interrupt by suspending its current program and executing an ISR. In 8085 processor all the hardware interrupts except TRAP are maskable. The interrupt initiated through TRAP pin and all software interrupts are non-maskable.

### 4.3 INTERRUPTS OF 8085

The interrupt in 8085 can come from one of the following two sources:

1. One source is from an external signal applied to TRAP, RST7.5, RST6.5, RST5.5 or INTR pin of the processor. The interrupts initiated by applying appropriate signals to these pins are called hardware interrupts.
2. The second source of an interrupt is the execution of the interrupt instruction "RST n" where n can take values from 0 to 7. The interrupts initiated by "RST n" instructions are called software interrupts.

#### 4.3.1 Software Interrupts Of 8085

Software interrupts are program instructions. When a software interrupt instruction is executed, the processor executes an **Interrupt Service Routine (ISR)** stored in the vector address of that software interrupt instruction. The software interrupts of 8085 are RST0, RST1, RST2, RST3, RST4, RST5, RST6 and RST7. The software interrupts of 8085 are vectored interrupts. Software interrupts cannot be masked or be disabled. The Vector addresses of software interrupts are given in Table-4.1.

Software interrupt instructions are included at the appropriate (or required) place in the main program. When the processor encounters the software instruction, it pushes the content of PC (**P**rogram **C**ounter) to stack. Then, it loads the vector address in to the PC and starts executing an ISR stored in this address. The last instruction of the ISR will be RET instruction. When the RET instruction is executed, the processor POPs the content of top of stack to PC. Hence, the processor control returns to main program after servicing the interrupt. *[Execution of ISR is referred to as servicing of interrupt.]*

**TABLE- 4.1**

Interrupt	Vector address
RST 0 RST 1	0000 <sub>H</sub> 0008 <sub>H</sub>
RST 2 RST 3	0010 <sub>H</sub> 0018 <sub>H</sub>
RST 4 RST 5	0020 <sub>H</sub> 0028 <sub>H</sub>
RST 6 RST 7	0030 <sub>H</sub> 0038 <sub>H</sub>

### 4.3.2 Hardware Interrupts of 8085

The hardware interrupts of 8085 are initiated by an external device by placing an appropriate signal at the interrupt pin of the processor. The processor keeps on checking the interrupt pins at the second T-state of the last machine cycle of every instruction. If the processor finds a valid interrupt signal and if the interrupt is unmasked and enabled, then the processor accepts the interrupt. The acceptance of the hardware interrupt is acknowledged by sending an  $\overline{\text{INTA}}$  signal to the interrupting device.

When the interrupt is accepted, the processor saves the content of the PC (Program Counter) in stack and then loads the vector address of the interrupt to the PC. (If the interrupt is non-vectorized, then the interrupting device has to supply the address of ISR when it receives  $\overline{\text{INTA}}$  signal.) Then the processor starts executing ISR in this address. The last instruction of ISR will be an RET instruction. When the processor executes the RET instruction, it POP the content of top of stack to PC. Thus the processor control returns to the main program after servicing the interrupt.

The hardware interrupts of 8085 are TRAP, RST 7.5, RST 6.5, RST 5.5 and INTR. TRAP, RST 7.5, RST 6.5 and RST 5.5 are vectored interrupts. In vectored interrupts the address to which the program control is transferred (when the interrupt is accepted) is fixed by the manufacturer. The vector addresses of hardware interrupts are given in Table-4.2. The INTR is a non-vectorized interrupt. Hence when a device interrupts through INTR, it has to supply the address of ISR after receiving interrupt acknowledge signal.

**TABLE- 4.2**

Interrupt	Vector address
RST 7.5	003C <sub>H</sub>
RST 6.5	0034 <sub>H</sub>
RST 5.5	002C <sub>H</sub>
TRAP	0024 <sub>H</sub>

The type of signal that has to be placed on the interrupt pin of hardware interrupts of 8085 are defined by INTEL. The TRAP interrupt is edge and level sensitive. Hence, to initiate TRAP, the interrupt signal has to make a **low** to **high** transition and then it has to remain **high** until the interrupt is recognized. The RST 7.5 interrupt is edge sensitive (positive edge). In order to initiate the RST 7.5, the interrupt signal has to make a **low** to **high** transition and it need not remain **high** until it is recognized. The RST 6.5, RST 5.5 and INTR are level sensitive interrupts. Hence, for these interrupts the interrupt signal should remain **high**, until it is recognized.

TRAP is a non-maskable interrupt and RST 7.5, RST 6.5 and RST 5.5 are maskable interrupts, which use the SIM (Set Interrupt Mask) instruction. Interrupts can be masked by moving an appropriate data (or code) to the accumulator and then executing the SIM instruction. The status of maskable interrupts can be read into the accumulator by executing the RIM instruction (RIM - Read Interrupt Mask).

All the hardware interrupts, except TRAP are disabled when the processor is reset and they can also be disabled by executing the DI instruction. (DI - Disable Interrupt). When an interrupt is disabled, it will not be accepted by the processor (i.e., INTR, RST 5.5, RST 6.5 and RST 7.5 are disabled by the DI instruction and upon hardware reset). In order to enable (or to allow) the disabled interrupts, the processor has to execute the EI instruction (EI - Enable Interrupt).

### **4.3.3 Priorities of Interrupts of 8085**

When all the interrupts are enabled, the priority sequence of hardware interrupts from highest to lowest is TRAP, RST 7.5, RST 6.5, RST 5.5 and INTR. When the 8085 processor accepts an interrupt it will disable all the hardware interrupts except TRAP. Hence in order to allow the higher priority interrupt while executing Interrupt Service Subroutine (ISR) for lower priority interrupt, enable the interrupt system in the beginning of ISR of lower priority interrupt, by executing EI instruction.

For example, if the processor accepts RST 5.5 interrupt, then it will disable RST 7.5, RST 6.5 and INTR interrupts. In order to allow the higher priority interrupt RST 7.5 and RST 6.5 while executing ISR of RST 5.5, the EI instruction should be executed in the beginning of ISR of RST 5.5.

The execution of software interrupt will not disable any hardware interrupt. Therefore while executing ISR of software interrupts, the processor will recognize or allow the hardware interrupts.

## **4.4 ENABLING, DISABLING AND MASKING OF 8085 INTERRUPTS**

### **TRAP**

The interrupt TRAP is non-maskable and it cannot be disabled by DI instruction. Also the TRAP is not disabled by system (processor) reset or after recognition of another interrupt. The only signal which can override TRAP is HOLD signal. (i.e., If the processor receives HOLD and TRAP at the same time then HOLD is recognized first and only then is TRAP recognized.)

### **INTR**

The interrupt INTR is disabled by any one of the following operations:

- Executing DI instruction.
- System or processor reset.
- After recognition (acceptance) of an interrupt.

The interrupt INTR can be enabled by executing EI instruction.

### **RST 7.5, RST 6.5 and RST 5.5**

The interrupt RST 7.5, RST 6.5 and RST 5.5 are disabled by any one of the following operations.

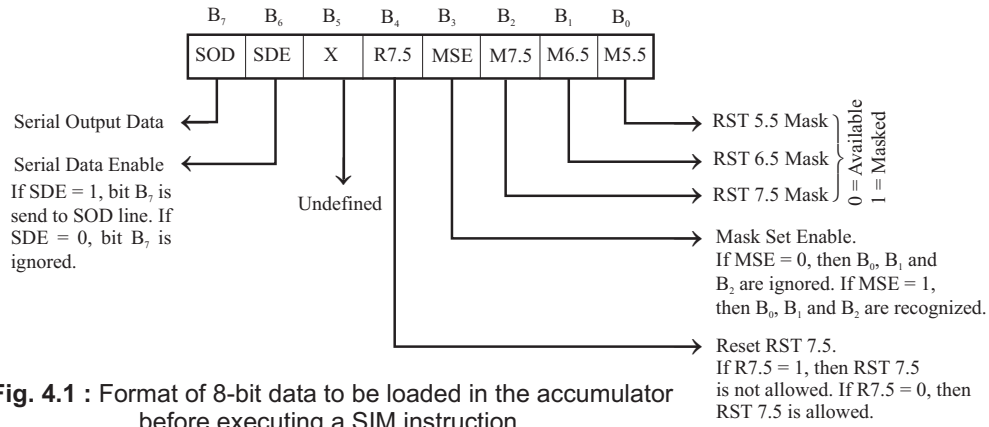
- Executing DI instruction.
- System or processor reset.
- After recognition (acceptance) of an interrupt.

These hardware interrupts can be enabled by executing EI instruction.

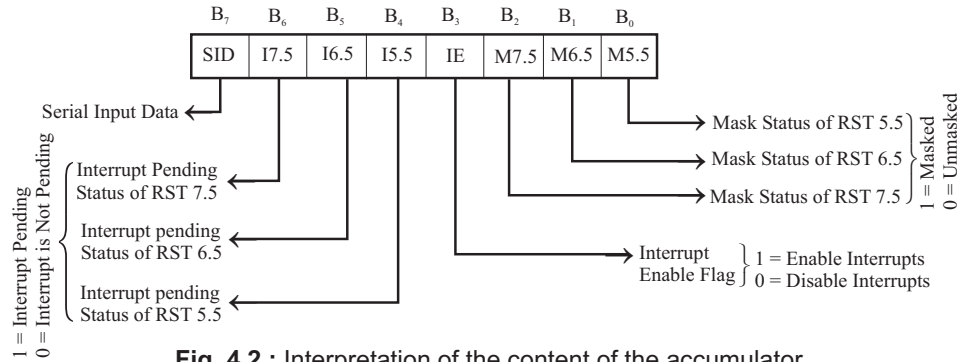
The 8085 provides additional masking facility for RST 7.5, RST 6.5 and RST 5.5 using SIM instruction. The status of these interrupts can be read by executing RIM instruction.

The masking or unmasking of RST 7.5, RST 6.5 and RST 5.5 interrupts can be performed by moving an 8-bit data to accumulator and then executing SIM instruction. The format of the 8-bit data is shown in Fig. 4.1.

The status of pending interrupts can be read from accumulator after executing RIM instruction. When RIM instruction is executed, an 8-bit data is loaded to the accumulator, which can be interpreted as shown in Fig. 4.2.



**Fig. 4.1** : Format of 8-bit data to be loaded in the accumulator before executing a SIM instruction.



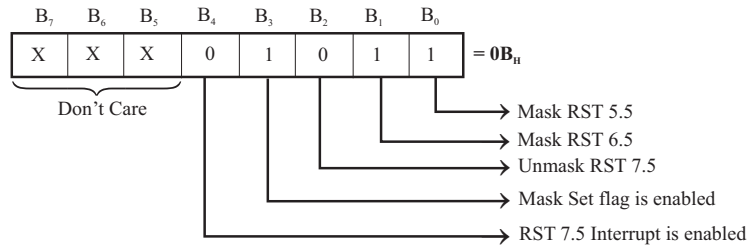
**Fig. 4.2** : Interpretation of the content of the accumulator after executing a RIM instruction.

### EXAMPLE 1

Write a program segment to mask RST 6.5 and RST 5.5 interrupts and enable RST 7.5 interrupt.

### Solution

The 8-bit data format to be loaded in the accumulator for enabling RST 7.5 and masking RST 6.5 and RST 5.5 is shown below. The data to be loaded in accumulator is 0B<sub>H</sub>.



**Program segment**

```

EI          ; Enable all interrupts of 8085
MVI A,0BH   ; Move 0B16 to A-register
SIM         ; Mask 6.5 and 5.5, Enable 7.5

```

**EXAMPLE 2**

*Assume that the 8085 microprocessor returns to the main program after servicing RST 6.5. (Remember that while servicing an interrupt all other interrupts are disabled.)*

*Write a program segment to check whether RST 5.5 interrupt is pending. If it is pending then the program has to enable RST 5.5 without affecting any other interrupts. Otherwise the program has to enable all interrupts and return to main program.*

**Solution**

The status of pending interrupts can be read by executing RIM instruction. This will load an 8-bit data in accumulator. If RST 5.5 is pending then the bit  $D_4$  in accumulator will be **1** and if it is not pending then bit  $D_4$  will be **0**. The following program segment have been written to check whether bit  $D_4$  is **1** or **0**. If it is **1** then the program control jumps to another part of program to enable RST 5.5 and mask other interrupts.

**Program segment**

```

RIM          ;Read the status of interrupts.
MOV C,A      ;Save the status in C-register.
ANI 10H      ;Check whether RST 5.5 is pending.
JNZ NEXT     ;If RST 5.5 is pending, go to NEXT.
EI           ;If RST 5.5 is not pending, enable
RET          ;all interrupts and return to main program.
NEXT: MOV A,C ;Get the interrupt status in A-register.
ANI FEH      ;Set  $D_0 = 0$ , for enabling RST 5.5
ORI 08H      ;Set  $D_3 = 1$ , for enabling interrupt enable flag.
SIM          ;Enable RST 5.5
JMP ISR55    ;Jump to Interrupt Service Routine of RST 5.5

```

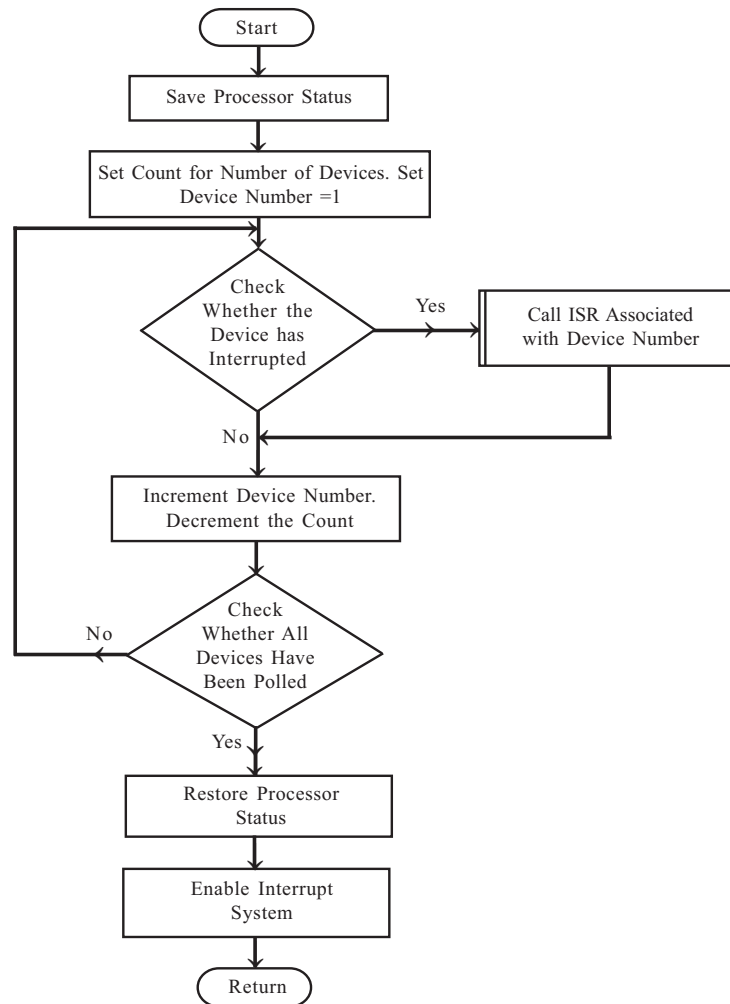
**4.5 POLLING OF INTERRUPTS****Polling**

When multiple devices interrupt the processor through one interrupt pin then a scheme or method is necessary to identify the interrupting device. The method of identifying the interrupting device is called polling. In the polling technique, each interrupting device will have an interrupt or status flag which is checked by the processor one by one. When a device want to interrupt the processor, it asserts the status flag **high**.

In the polling scheme, the processor has to execute the following steps in order to process the interrupts. The flowchart for interrupt polling routine is shown in Fig. 4.3.

1. Branch to interrupt polling routine.
2. Identify the device that caused the interrupt.
3. Branch to ISR associated with this device and execute ISR.
4. Enable the interrupt system.
5. Return to main program.

Polling can be performed either by software or by hardware. In software polling, the entire polling process is governed by processor instructions (program). In hardware polling, the hardware takes care of checking the status of interrupting devices and allowing then one by one to the processor.

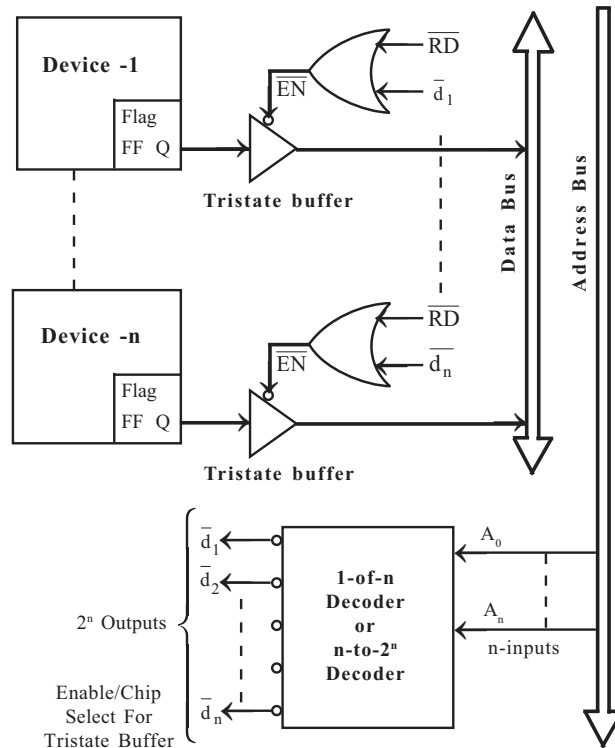


**Fig. 4.3 :** Flowchart of interrupt polling routine.

### **Software polling**

The hardware setup necessary for software polling is shown in Fig. 4.4. In this system, each interrupting device has an interrupt status flag associated with it. This flag is the output of a Flip-Flop (FF). The status flip-flop is connected to the data bus of the processor through a tristate buffer. Whenever the device needs to interrupt the processor, it sets the flag as high.

A decoder has been used to generate n-numbers of logic **low** device select signals ( $d_1$  to  $d_n$ ), by using n-address lines of the processor. The read control signal ( $\overline{RD}$ ) of the processor is logically ORed with device select signals and then used to enable the tristate buffer.



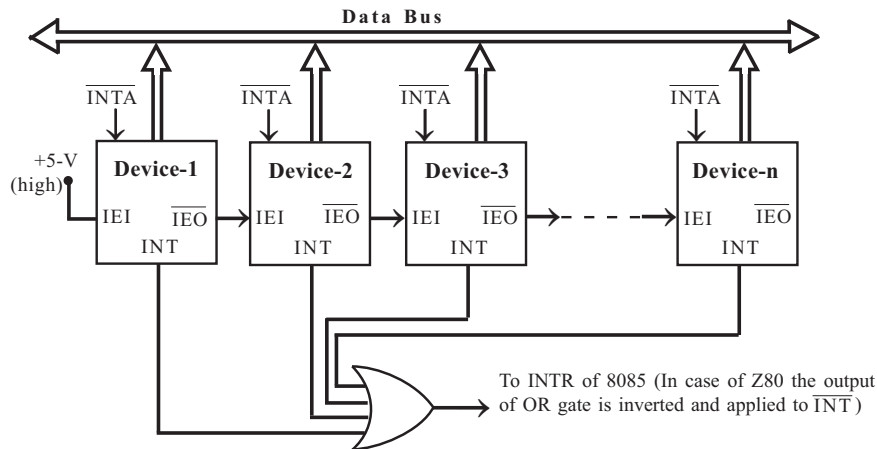
**Fig. 4.4 :** Circuit for software polling.

The processor checks the status of the interrupting device by executing a read cycle. In the read cycle, the processor sends an appropriate address on the address bus and then  $\overline{RD}$  is asserted **low**. Using this address, the decoder generates the device select signal, which is logically ORed with  $\overline{RD}$  to produce enable signal for tristate buffer. When the tristate buffer is enabled, the status bit on the flip-flop is placed on the data bus. Now, the processor reads the status bit and checks whether it is **1** or **0**. If it is **1** then the processor executes the ISR (**I**nterrupt **S**ervice **R**outine) corresponding to the device (whose status is checked). If it is **0** then the processor continues polling. The polling stops only after all the devices have been scanned for pending interrupt.

### Hardware polling

In hardware polling, the processor need not check the status of interrupting device. The hardware setup will be such that it allows the interrupts one by one to the processor. The commonly used mechanism for hardware polling is "daisy chaining" as shown in Fig. 4.5.

Each device in the hardware setup shown in Fig. 4.5 requires a minimum of two input control signals and two output control signals. The input control signals are  $\overline{INTA}$  and  $IEI$ . The output control signals are  $\overline{IEO}$  and  $INT$ .



**Fig. 4.5 :** A daisy chain for hardware polling.

- $\overline{\text{INTA}}$  : **Interrupt Acknowledge Signal:** This signal is similar to interrupt acknowledge signal of the processor. This signal informs the interrupting device that the processor is running interrupt acknowledge machine cycle. (In response, the interrupting device can place 8-bit RST opcode on the data bus in case of 8085 or 8-bit vector address in case of Z80.)
- IEI : **Interrupt Enable Input:** If this input signal is **high** then the device will place an 8-bit binary code on the data bus. If this input signal is **low**, then the device will wait.
- $\overline{\text{IEO}}$  : **Interrupt Enable Output:** This is an output signal generated by the device. Normally this signal is **high**. If the device has to interrupt the processor then it asserts  $\overline{\text{IEO}}$  **low**. The signal  $\overline{\text{IEO}}$  is also asserted **low** if the input signal IEI of the concerned device is **low**. After servicing the interrupt, this signal is asserted **high**.
- INT : **Interrupt Signal:** This is an output signal of the device and asserted **high** to interrupt the processor.

In the hardware setup of Fig. 4.5, device-1 has the highest priority and device-n has the lowest priority. Also, a device can load an 8-bit code on the data bus only if IEI is **high** and  $\overline{\text{IEO}}$  is **low**.

Let us assume that all the devices are interrupting the processor simultaneously. Now, each device will assert its INT **high** and  $\overline{\text{IEO}}$  **low**. Since  $\overline{\text{IEO}}$  of one device is connected to IEI of the next device, all the devices will wait, except device-1 because IEI of device-1 is permanently tied to **high**.

All the INT signals of the devices are ORed and applied as interrupt signal to the processor. When the processor accepts the interrupts, it asserts  $\overline{\text{INTA}}$  **low**. On receiving  $\overline{\text{INTA}}$  **low**, the device-1 will load an 8-bit code data bus and then assert  $\overline{\text{IEO}}$  **high** (because, device-1 alone has its IEI signal as **high** and  $\overline{\text{IEO}}$  signal as **low**). Now the processor disables all other interrupts and starts servicing the device-1. At the end of service, the interrupts are enabled.



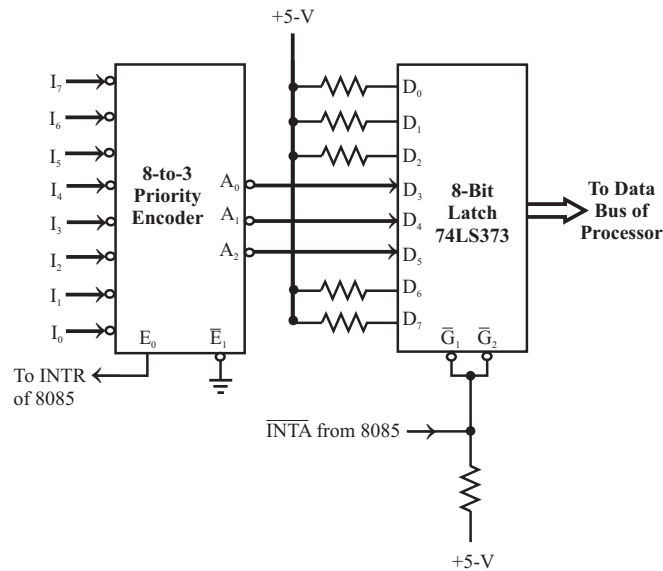
Now, the device-2 is allowed to interrupt the processor. After recognizing (accepting) the interrupt of device-2, its  $\overline{\text{IEO}}$  will be **high** which allows the interrupt of device-3 and so on. Thus in hardware polling the devices are allowed one by one to interrupt the processor.

#### 4.6 INTR AND ITS EXPANSION

The INTR is general interrupt request. An external device can interrupt the processor by placing a **high** signal on INTR pin of 8085. If the processor accepts the interrupt, then it will send an acknowledge signal  $\overline{\text{INTA}}$  to the interrupting device. On receiving the acknowledge signal, the interrupting device has to place either an **RST n** opcode (or CALL opcode followed by 16-bit address) on the data bus.

On receiving the **RST n** opcode, the 8085 processor generates the vector address of **RST n** instruction. It saves the content of Program Counter (PC) in stack. Then it loads the vector address in PC and executes an **Interrupt Service Routine (ISR)** stored at this address. (when it receives CALL opcode it executes an interrupt service routine stored at CALL address.)

The INTR interrupt can be expanded to accept 8-interrupt inputs using 8-to-3 priority encoder as shown in Fig. 4.6. (Other expansion schemes are shown in Fig. 4.8 and Fig. 4.10.)



**Fig. 4.6 :** Expanding an INTR of the 8085 using an 8-to-3 priority encoder.

The priority encoder has 8 inputs  $I_0$  to  $I_7$  and three outputs  $A_0$  to  $A_2$ . It also has an output control signal,  $E_0$ . If the priority encoder receives a logic **low** at one of the inputs, for example  $I_n$ , then it asserts  $E_0$  **high** and outputs the binary value of  $n$  on the output lines  $A_0$ ,  $A_1$  and  $A_2$  lines (i.e., if input  $I_0$  is **low** then output is 000; if input  $I_1$  is **low** then output is 001 and so on). In this scheme  $I_7$  has the highest priority and  $I_0$  has the lowest priority.

Eight external devices can interrupt the processor through  $I_0$  to  $I_7$  lines, by placing a logic **low** on these pins. On receiving a valid interrupt signal the priority encoder allows the highest priority interrupt by asserting  $E_0$  **high** and sending the corresponding binary value on  $A_0$ ,  $A_1$  and  $A_2$  lines. The  $E_0$  is connected to INTR of 8085 and  $A_0$ ,  $A_1$  and  $A_2$  are connected to the inputs  $D_3$ ,  $D_4$  and  $D_5$  of an 8-bit latch. All other inputs of the latch are tied to +5-V (logic 1) permanently.

The opcodes and vector addresses of RST n instructions are shown in Table-4.3. If we carefully look at the opcode of RST instruction, the binary bits  $D_3$ ,  $D_4$ ,  $D_5$  constitutes the binary value of n in RST n instruction and all other bits are 1's. The priority encoder helps in placing the RST opcodes at the input of latch (74LS373). [The priority encoder places the RST n opcode for the interrupt  $I_n$ .]

**TABLE - 4.3 : OPCODES OF RST INSTRUCTIONS**

RST instruction	Opcode in binary								Opcode in hexa	Vector address
	$D_7$	$D_6$	$D_5$	$D_4$	$D_3$	$D_2$	$D_1$	$D_0$		
RST 0	1	1	0	0	0	1	1	1	C7	0000 <sub>H</sub>
RST 1	1	1	0	0	1	1	1	1	CF	0008 <sub>H</sub>
RST 2	1	1	0	1	0	1	1	1	D7	0010 <sub>H</sub>
RST 3	1	1	0	1	1	1	1	1	DF	0018 <sub>H</sub>
RST 4	1	1	1	0	0	1	1	1	E7	0020 <sub>H</sub>
RST 5	1	1	1	0	1	1	1	1	EF	0028 <sub>H</sub>
RST 6	1	1	1	1	0	1	1	1	F7	0030 <sub>H</sub>
RST 7	1	1	1	1	1	1	1	1	FF	0038 <sub>H</sub>

When the processor accepts the interrupt, it sends  $\overline{INTA}$  signal to the interrupting device. This signal is used to enable the latch. When the latch is enabled, the RST opcode available at the input is latched into output lines. The output of latch is connected to data bus of the processor. Hence, the opcode will be placed on the data bus.

This opcode is read by the processor and then it generates the vector address of the RST instruction internally. The processor saves the current value of Program Counter (PC) in stack and loads the vector address in PC. Now the processor starts servicing the interrupt.

#### 4.7 PROGRAMMABLE INTERRUPT CONTROLLER - INTEL 8259

The 8259 is a programmable interrupt controller. It is used to expand the interrupts of 8085 or 8086 processor. One 8259 can accept eight interrupt requests and allows one by one to the processor INTR pin. The interrupt controller can be used in cascaded mode to expand the interrupts upto 64.

##### Features of 8259

1. It is programmed to work with either 8085 or 8086 processor.
2. It manages 8 interrupts according to the instructions written into its control registers.
3. The 8086 processor-based system, supplies the type number of the interrupt and the type number is programmable. In 8085 processor-based system, it vectors an interrupt request anywhere in the memory map and the interrupt vector address is programmable.

4. The priorities of the interrupts are programmable. The different operating modes which decides the priorities are automatic rotation mode, specific rotation mode and fully nested mode.
5. Interrupts can be masked or unmasked individually.
6. The 8259 is programmed to accept either level triggered interrupt signal or edge triggered interrupt signal.
7. The 8259 provides the status of pending interrupts, masked interrupts and interrupt being serviced.
8. The 8259s can be cascaded to accept a maximum of 64 interrupts.

#### 4.7.1 Interfacing 8259 With 8085 Microprocessor

The 8259 is a 28-pin IC packed in DIP. The various pins of 8259 are shown in Fig. 4.7. It requires two internal addresses and they are  $A_0 = 0$  or  $A_0 = 1$ . It can be either memory-mapped or IO-mapped in the system. The interfacing of 8259 to 8085 is shown in Fig. 4.8. In Fig. 4.8, the 8259 is IO-mapped in the system. The low order data bus lines  $D_0$ - $D_7$  are connected to  $D_0$ - $D_7$  of 8259. The address line  $A_0$  of the 8085 processor is connected to  $A_0$  of 8259 to provide the internal address. The 8259 requires one chip select signal. The chip select signal for 8259 is generated by using 3-to-8 decoder. The address lines  $A_4$ ,  $A_5$  and  $A_6$  are used as input to decoder. The control signal  $\text{IO}/\overline{\text{M}}$  is used as logic **high** enables for decoder and the address line  $A_7$  is used as logic **low** enable for decoder. The IO addresses of 8259 are shown in Table-4.4. The signals  $\text{CAS}_0$ - $\text{CAS}_2$  are used only in cascade operation of 8259s.

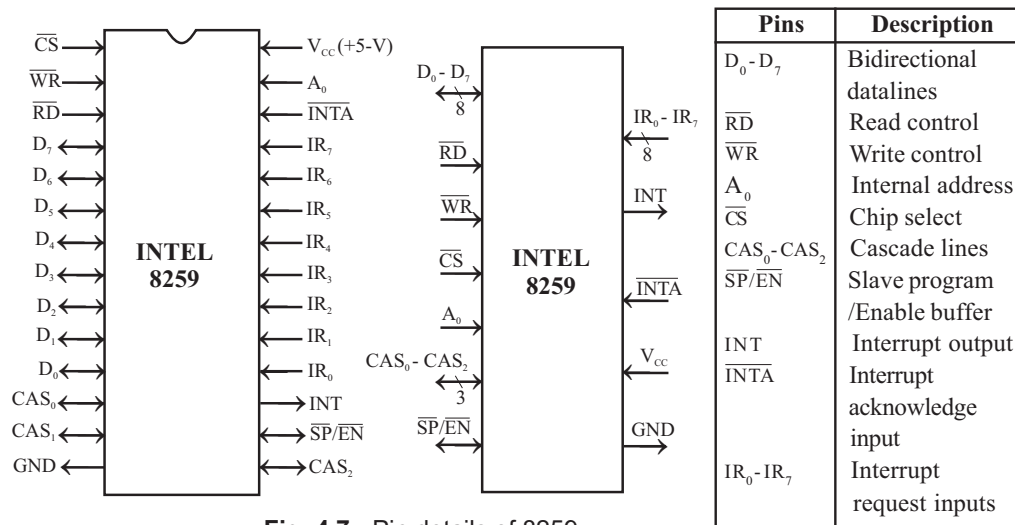


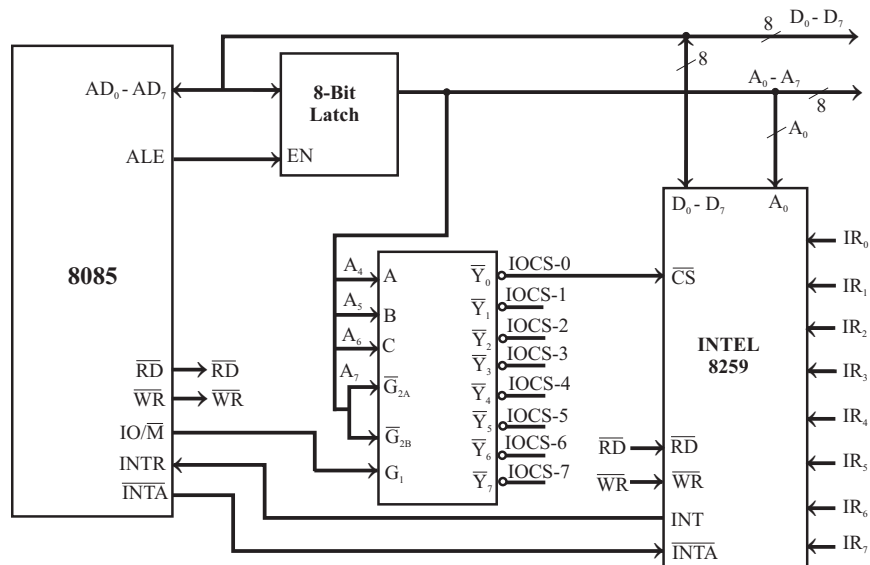
Fig. 4.7 : Pin details of 8259.

The  $\overline{\text{SP/EN}}$  pin can be used as input or output signal. In non-buffered mode it is used as input signal and tied to logic-1 in master 8259 and logic-0 in slave 8259. In buffered mode it is used as output signal to disable the data buffers while data is transferred from 8259A to the CPU.

**TABLE - 4.4 : IO ADDRESS OF 8259**

	Binary address								Hexa address
	Decoder input/ enable			Input to address pin of 8259					
	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	
For A <sub>0</sub> of 8259 to be zero	0	0	0	0	x	x	x	0	00
For A <sub>0</sub> of 8259 to be one	0	0	0	0	x	x	x	1	01

*Note : Don't care "x" is considered as zero.*

**Fig. 4.8 : Interfacing 8259 to 8085 microprocessor.**

### Working of 8259 With 8085 Processor

First the 8259 should be programmed by sending Initialization Command Word (ICW) and Operational Command Word (OCW). These command words will inform 8259 about the following:

- Type of interrupt signal (Level triggered/Edge triggered).
- Type of processor (8085/8086).
- Call address and its interval (4 or 8) in the case of 8085 and interrupt type number in case of 8086.
- Masking of interrupts.
- Priority of interrupts.
- Type of end of interrupt.

Once the 8259 is programmed it is ready for accepting the interrupt signal. When it receives an interrupt through any one of the interrupt lines IR<sub>0</sub>-IR<sub>7</sub>, it checks for its priority and also checks whether it is masked or not. If the previous interrupt is completed and if the current request has the highest priority and unmasked, then it is serviced.

For servicing this interrupt the 8259 will send INT signal to INTR pin of 8085. In response it expects an acknowledgement  $\overline{\text{INTA}}$  from the processor. When the processor accepts the interrupt, it sends three  $\overline{\text{INTA}}$  one by one. In response to first, second and third  $\overline{\text{INTA}}$  signals, the 8259 will supply CALL opcode, low byte of call address and high byte of call address respectively. Once the processor receives the call opcode and its address, it saves the content of Program Counter (PC) in stack and load the CALL address in PC and starts executing the interrupt service routine stored in this call address.

#### 4.7.2 Functional Block Diagram Of 8259

The functional block diagram of 8259 is shown in Fig. 4.9. It shows eight functional blocks. They are: Control logic, Read/Write logic, Data bus buffer, Interrupt Request Register (IRR), In-Service Register (ISR), Interrupt Mask Register (IMR), Priority Resolver (PR) and Cascade buffer.

The data bus and its buffer are used for the following activities:

1. The processor sends control word to data bus buffer through  $D_0-D_7$ .
2. The processor reads status word from data bus buffer through  $D_0-D_7$ .
3. From the data bus buffer the 8259 sends type number (in case of 8086) or the call opcode and address (in case of 8085) through  $D_0-D_7$  to the processor.

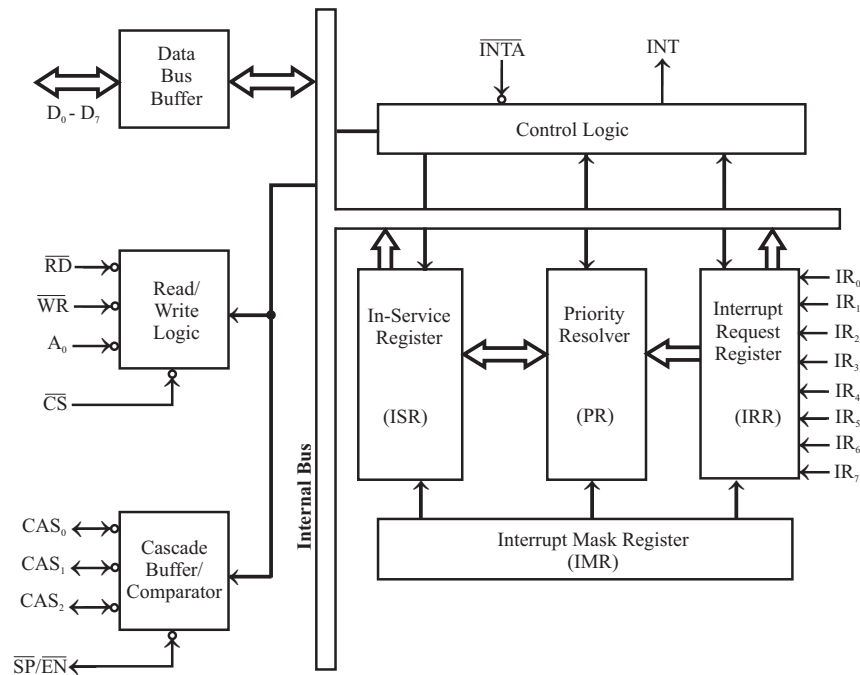


Fig. 4.9 : Functional block diagram of an 8259.



**Fig. 4.10** : Example of cascade connection of programmable interrupt controllers-8259.

The processor uses the  $\overline{RD}$ ,  $\overline{WR}$  and  $A_0$  to read or write 8259. The 8259 is selected by  $\overline{CS}$ . The IRR has eight input lines ( $IR_0$ - $IR_7$ ) for interrupts. When these lines go **high**, the requests are stored in IRR. It registers a request only if the interrupt is unmasked. Normally  $IR_0$  has highest priority and  $IR_7$  has the lowest priority. The priorities of the interrupt request input are also programmable.

The interrupt mask register stores the masking bits of the interrupt lines to be masked. The relevant information is sent by the processor through OCW1. The in-service register keeps track of the interrupt input currently being serviced. For each input that is currently being serviced, the corresponding bit will be set in the in-service register. The priority resolver examines the interrupt request, mask and the in-service registers and determines whether the INT signal should be sent to the processor or not.

The cascade buffer/comparator is used to expand the interrupts of 8259. Figure 4.10 is an example of 8259s in cascade connection. In this configuration, one 8259 will be directly interrupting 8085 and it is called master 8259. To each interrupt request input of master 8259 ( $IR_0$ - $IR_7$ ) one slave 8259 can be connected. The 8259s interrupting the master 8259 are called slave 8259s.

Each 8259 has its own address so that each 8259 can be programmed independently by sending command words and status the independently bytes can be read from it.

The cascade pins ( $CAS_0$ ,  $CAS_1$  and  $CAS_2$ ) from the master are connected to the corresponding pins of the slave. For the master, these pins function as output and for the slave device they function as input. For the slave 8259, the  $\overline{SP}/\overline{EN}$  pin is tied **low** to let the device know that it is a slave.

### 4.7.3 Processing of Interrupts By 8259

To implement interrupts, the processor interrupt should be enabled and 8259 initialized. The 8259 is initialized by sending ICWs and OCWs. The ICWs are used to set up the proper conditions and specify CALL vector addresses. The OCWs are used to perform functions such as masking interrupts, setting up status, read operations, etc. After the 8259 is initialized, the following sequence of events occur when one or more interrupt request lines go **high**.

1. The IRR stores the request.
2. The priority resolver checks three register (IRR, IMR, ISR). The IRR is checked for interrupt request. The IMR is checked for masking bits and the ISR for the interrupt request being served. It resolves the priority and sets the INT **high** when appropriate.
3. The processor acknowledges the interrupt by sending  $\overline{INTA}$  signal.
4. When the  $\overline{INTA}$  is received, the appropriate priority bit in the ISR is set to indicate which interrupt level is being served and the corresponding bit in the IRR is reset to indicate that the request is accepted. Then, the opcode for the CALL instruction is placed on the data bus.
5. When the processor decodes the CALL instruction, it places two more  $\overline{INTA}$  signals on the data bus.
6. When the 8259 receives the second  $\overline{INTA}$ , it outputs the low-order byte of CALL address on the data bus. When the third  $\overline{INTA}$  signal is received the 8259 outputs high-order byte of CALL address on the data bus. The CALL address is the vector memory location for the interrupt (this address is programmed by sending ICW1 and ICW2 to the control register during the initialization).

7. Once the processor reads the CALL opcode and address from 8259 the bit corresponds to the current interrupt being serviced in the in-service register should be reset to allow next interrupt. This is done automatically if 8259 is programmed for **Automatic End Of Interrupt (AEOI)**. Alternatively the processor can send command word at the end of interrupt service routine to inform 8259 about the end of interrupt.
8. After receiving the CALL opcode and address, the processor save the content of **Program Counter (PC)** in stack and load the call address in PC. Thus the program control is transferred to the memory location specified by the CALL instruction.

#### 4.7.4 Programming 8259 (or Initializing 8259)

The 8259 has four numbers of **Initialization Command Word (ICW)** and three numbers of **Operational Command Word (OCW)**. The command words are sent to 8259 by selecting it by  $\overline{CS}=0$  and  $A_0=0$  or 1. Certain command words are sent to the internal address,  $A_0=0$  and others with  $A_0=1$ .

The OCW1 should be sent to 8259 after sending ICWs. The OCW2 can be sent any time (either before servicing interrupt or at the end of interrupt service routine). The order of sending ICWs and OCWs are shown as a flowchart in Fig. 4.11. The format of ICWs and OCWs are shown in Fig. 4.12 and Fig. 4.13.

The ICWs are used to program the following features of 8259.

- Call address interval.
- Level or Edge triggered.
- Cascade mode or single.
- Vector addresses or Type number.
- 8085 or 8086 mode.
- Auto or Normal end of interrupt.
- Special fully nested mode.

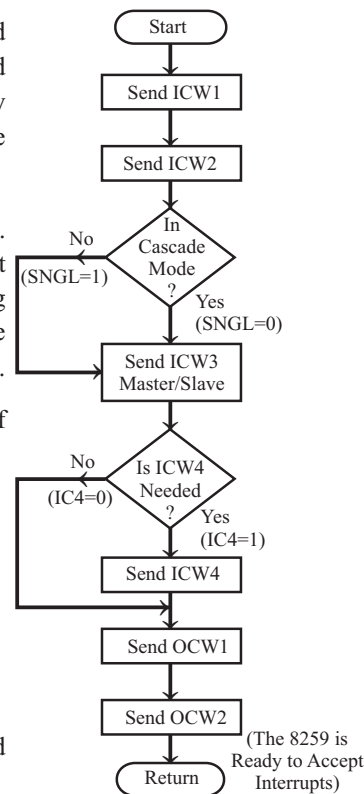
The OCWs are used to read the status of interrupts and also to program the following features of 8259.

- Masking or Unmasking of individual interrupts.
- Specific or Non-specific end of interrupt.
- Priority modes.

A brief discussion about ICWs and OCWs are presented in the following sections.

##### Initialization Command Words (ICW)

The 8259A has four ICWs and they are named as ICW1, ICW2, ICW3 and ICW4. When only one 8259 is used in the system, we have to program 8259 by sending ICW1, ICW2 and ICW4. When a number of 8259s are used in the system then we have to program each 8259 by sending all the four ICWs. The format of ICW3 for master and slave 8259 are different.



**Fig. 4.11** : Sending order of ICWs and OCWs.



**ICW1 :** The IC4 bit is set to one if we decide to send ICW4. The single or cascade mode of operation is selected by programming the "SNGL" bit. The LTIM bit determines whether the interrupt request input are positive edge-triggered or level triggered. The bit ADI is used to program a call address interval of 4 or 8 in case of 8085 system. The upper three bits ( $D_5$ ,  $D_6$  and  $D_7$ ) of ICW1 are used to program the upper three bits of low byte of call address. The lower five bits of low byte of call address are automatically inserted by 8259 as shown in Table-4.5.

**TABLE - 4.5 : LOW BYTE CALL ADDRESS**

Interrupt input	Low byte call address															
	Interval = 4								Interval = 8							
	$B_7$	$B_6$	$B_5$	$B_4$	$B_3$	$B_2$	$B_1$	$B_0$	$B_7$	$B_6$	$B_5$	$B_4$	$B_3$	$B_2$	$B_1$	$B_0$
IR0	$A_7$	$A_6$	$A_5$	0	0	0	0	0	$A_7$	$A_6$	0	0	0	0	0	0
IR1	$A_7$	$A_6$	$A_5$	0	0	1	0	0	$A_7$	$A_6$	0	0	1	0	0	0
IR2	$A_7$	$A_6$	$A_5$	0	1	0	0	0	$A_7$	$A_6$	0	1	0	0	0	0
IR3	$A_7$	$A_6$	$A_5$	0	1	1	0	0	$A_7$	$A_6$	0	1	1	0	0	0
IR4	$A_7$	$A_6$	$A_5$	1	0	0	0	0	$A_7$	$A_6$	1	0	0	0	0	0
IR5	$A_7$	$A_6$	$A_5$	1	0	1	0	0	$A_7$	$A_6$	1	0	1	0	0	0
IR6	$A_7$	$A_6$	$A_5$	1	1	0	0	0	$A_7$	$A_6$	1	1	0	0	0	0
IR7	$A_7$	$A_6$	$A_5$	1	1	1	0	0	$A_7$	$A_6$	1	1	1	0	0	0

**ICW2 :** This command word is used to program the high byte of call address.

**ICW3 :** The ICW3 should be sent to 8259s in cascade operation. Separate formats are provided for master and slave 8259s. In cascade mode, slave 8259s are connected to one or more IR inputs of master 8259 and each slave is provided with a slave ID number. The connection of slave 8259s to the IR inputs of master are informed to master through ICW3. For slave 8259s, the ID numbers are informed through ICW3.

**ICW4 :** The ICW4 is used to inform 8259 whether it is connected to 8085 or 8086-based system. For 8085-based system the right most bit is programmed as zero. The AEOL bit is used to program the method of terminating the interrupt. If AEOL is set to one, then the 8259 will automatically reset the interrupt request bit in in-service register after supplying the call instruction to the processor. If AEOL bit is programmed as zero then the processor has to send OCW2 to terminate the interrupt.

The BUF and M/S bits are used to select the buffered or non-buffered operation of master/slave 8259. The SFNM bit is used to nest or include the priorities of the slave IR input with the master IR input. For example, if  $IR_4$  of a master 8259 has a slave 8259 connected to it and they are programmed for SFNM operation, the priorities of  $IR_0$  to  $IR_7$  of slave 8259 will be higher than  $IR_5$  to  $IR_7$  of master 8259.

### Operation Command Words (OCWs)

The 8259 has three **Operation Command Words (OCWs)** and they are named as OCW1, OCW2 and OCW3.

**OCW1 :** The OCW1 is sent to 8259 to mask or unmask the IR inputs of 8259. At any time the mask status of interrupts can be read by the processor by using the same address of OCW1.

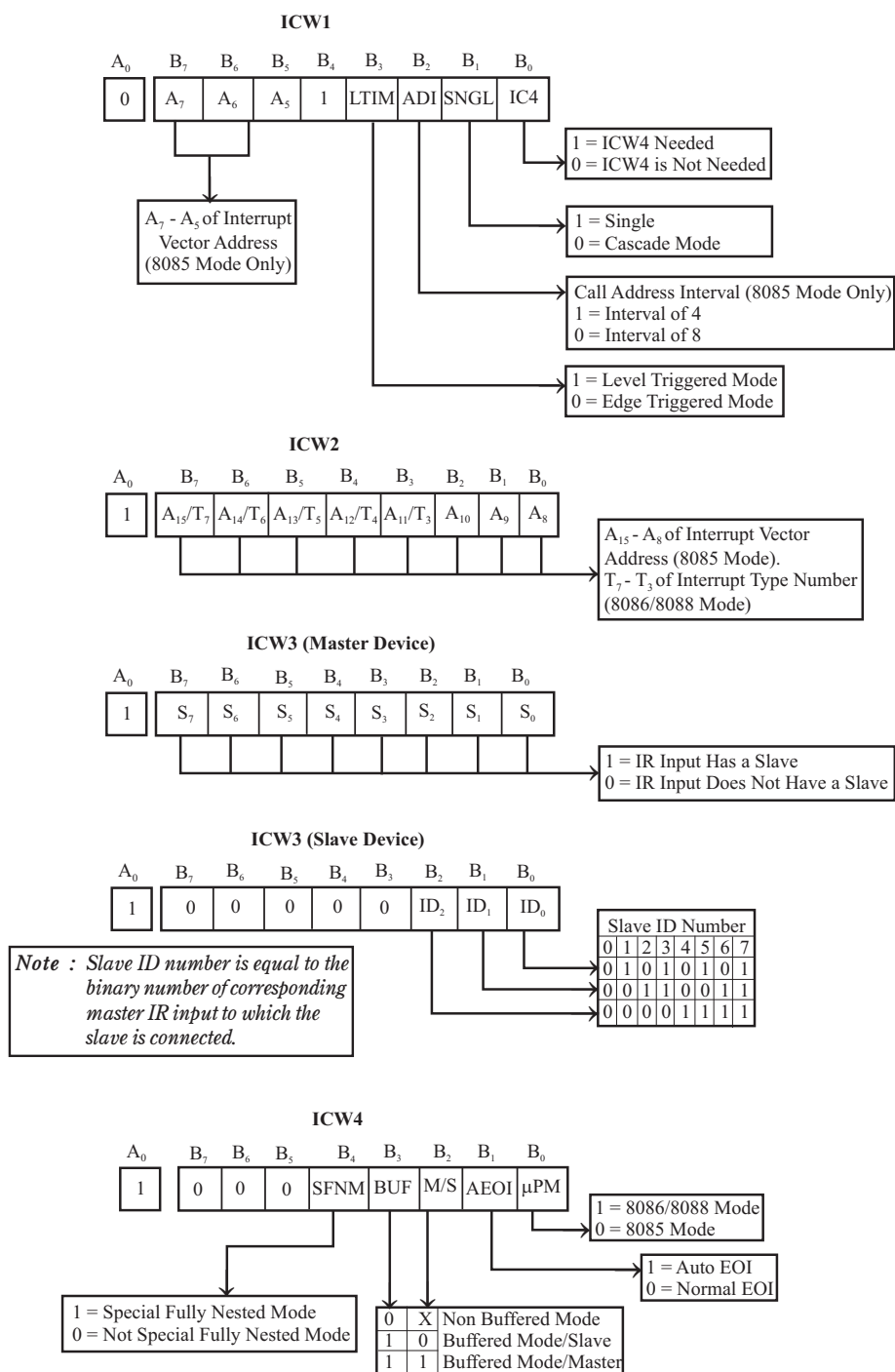


Fig. 4.12 : Format of ICWs.

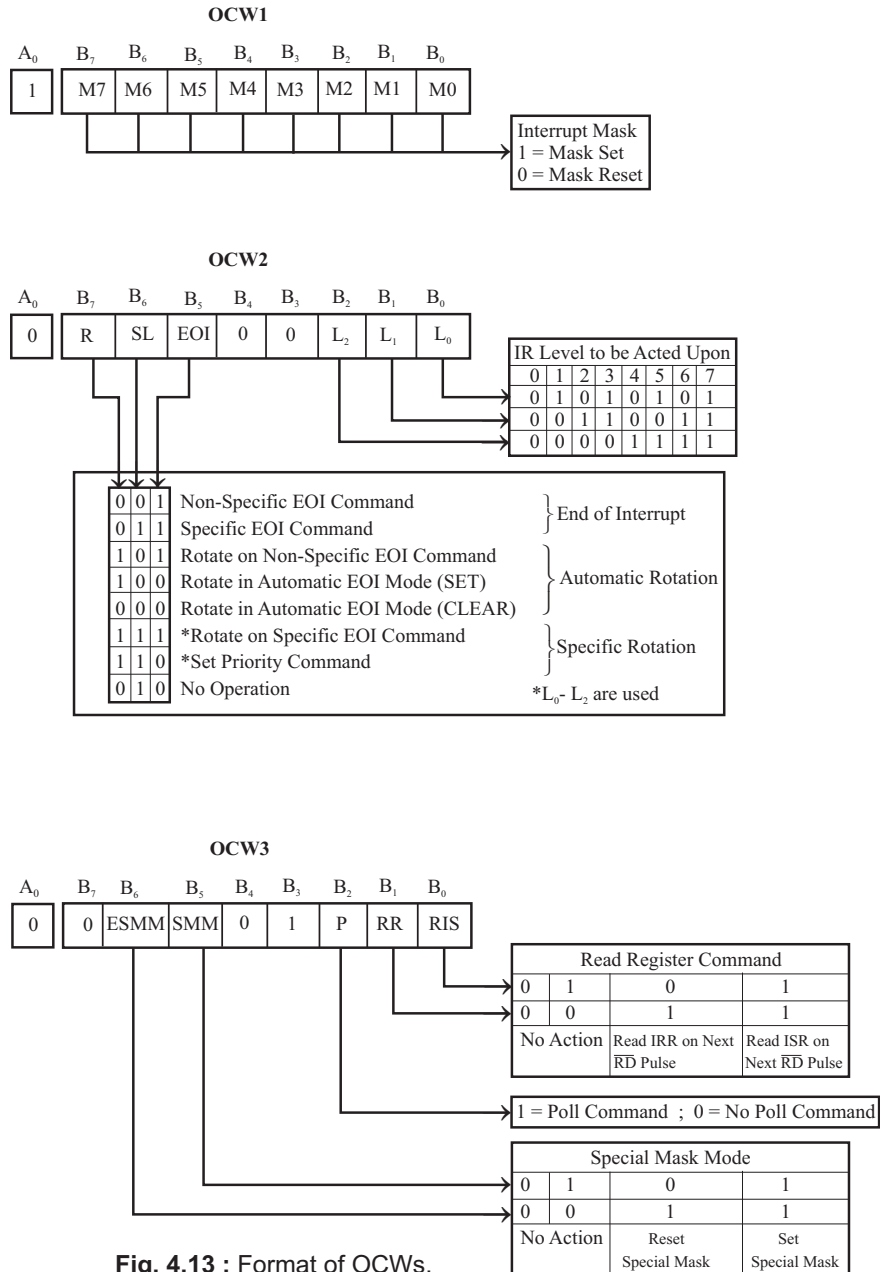


Fig. 4.13 : Format of OCWs.

- OCW2 :** The OCW2 is sent to 8259A only when the AEI mode (in ICW4) is not selected. The OCW2 is sent by the processor to decide on the type of **End-of-Interrupt (EOI)** and to program the priorities of the interrupt (i.e. IR inputs of 8259A). The different methods of EOI are discussed here.
- (i) **Non-specific End-of-Interrupt :** This command is sent by processor to 8259 to terminate the current interrupt being serviced by 8259. This resets the corresponding bit in in-service register of 8259 and allows the next higher priority interrupt.
  - (ii) **Specific End-of-Interrupt :** This command is sent by the processor to reset or terminate a specific interrupt request, decided by the lower three bits of OCW2.
  - (iii) **Rotate on Non-specific EOI :** This command will take action same as that of non-specific EOI except that it rotates the priorities after resetting the bit in in-service register. In this case the interrupts will have rotating priority, in which the priority of the currently serviced interrupt becomes the least.
  - (iv) **Rotate on Automatic EOI :** This command is sent to 8259 to select automatic EOI with rotating priority.
  - (v) **Rotate on Specific EOI :** This command will take action similar to that of specific EOI except that it rotates the priorities of the interrupts after they are serviced.
  - (vi) **Set priority :** The command is sent to set the priority of the interrupt level specified by lower three bits of OCW2 as the least.
- OCW3 :** The OCW3 is used to set special mask mode, poll the active interrupt request and read the in-service and interrupt request registers. In special mask mode, the mask status are negated to allow the interrupts masked by interrupt mask register.

#### 4.8 SUMMARY

- Interrupt is a signal sent by an external device to the processor, to request the processor to perform a particular task or work.
- Interrupts are used for data transfer between the peripheral and the microprocessor.
- The execution of interrupt service routine is called servicing of interrupt.
- There are two types of interrupts namely, hardware and software interrupts.
- Software interrupts are program instructions.
- The software interrupts of 8085 are RST 0, RST 1, RST 2, RST 3, RST 4, RST 5, RST 6 and RST 7.
- The hardware interrupts are initiated by an external device by placing an appropriate signal at the interrupt pin of the processor.
- The hardware interrupts of 8085 are TRAP, RST 7.5, RST 6.5, RST 5.5 and INTR.
- The TRAP, RST 7.5, RST 6.5 and RST 5.5 are vectored interrupts.
- The TRAP interrupt is edge and level sensitive.
- The RST 7.5 interrupt is edge sensitive (positive edge).
- The RST 6.5, RST 5.5 and INTR are level sensitive.
- In vectored interrupts the address to which the program control is transferred is fixed by the manufacturer of the microprocessor.
- Masking is preventing the interrupt from disturbing the main program.
- The interrupts RST 7.5, RST 6.5, RST 5.5 and INTR are maskable interrupts.
- All the hardware interrupts except TRAP are disabled when the processor is resetted.
- INTR is a general interrupt request which can support eight interrupting devices with the help of external hardware circuits.
- The TRAP interrupt cannot be disabled by DI instruction or by system reset.

- The INTR, RST7.5, RST6.5 and RST5.5 interrupts are disabled after execution of DI instruction or after system reset or after recognition of one of the interrupts.
- The programmable interrupt controller is used to expand the INTR interrupt of 8085 or 8086 processor .
- The 8259 consists of control logic, read/write logic, data bus buffer, priority resolver, interrupt request register, interrupt mask register, in-service register and cascade buffer.
- The automatic rotation mode, specific rotation mode and fully nested mode are the different operating modes that decides priority of the interrupts input to 8259.
- The 8259 provides the status of pending interrupts, masked interrupts and interrupt being serviced.
- The 8259 is initialized by sending ICWs and OCWs.
- The ICWs are used to set up the proper conditions and specify CALL vector addresses.
- The OCWs are used to perform function such as masking interrupts, setting up status, read operations, etc.
- The interrupts input to 8259 can be masked or unmasked individually.
- The 8259 that is directly interrupting the processor is called master 8259.

#### 4.9 SHORT QUESTIONS AND ANSWERS

---

##### 4.1 What is an Interrupt ?

Interrupt is a signal sent by an external device to the processor so as to request the processor to perform a particular task or work.

##### 4.2 How are interrupts classified ?

There are three methods of classifying interrupts.

- Method I : The interrupts are classified into Hardware and Software interrupts.
- Method II : The interrupts are classified into Vectored and Non-vectored interrupt.
- Method III : The interrupts are classified into Maskable and Non-maskable interrupts.

##### 4.3 How does a microprocessor service an interrupt request ?

When the processor recognizes an interrupt, it saves the processor status in stack. Then it calls and executes an **Interrupt Service Routine (ISR)**. At the end of ISR, it restores the processor status and the program control is transferred to the main program.

##### 4.4 What is the function of interrupt service routine?

For each interrupt the processor has to perform a specific job. An interrupt service routine has been developed in order to perform the operations required for a device that is interrupting the processor.

##### 4.5 How are interrupts affected by system reset?

Whenever the processor or system is reset, all the interrupts except TRAP are disabled. In order to enable the interrupts, EI instruction has to be executed after a reset.

##### 4.6 What are Software interrupts?

Software interrupts are program instructions. These instructions are inserted at desired locations in a program. While running a program, if a software interrupt instruction is encountered then the processor executes an interrupt service routine.

##### 4.7 What is Hardware interrupt?

If an interrupt is initiated in a processor by applying an appropriate signal to an interrupt pin, then the interrupt is called Hardware interrupt.

4.8 *What is the difference between Software and hardware interrupts?*

Software interrupt is initiated by the main program, but a hardware interrupt is initiated by an external device.

In 8085, the software interrupt cannot be disabled or masked but the hardware interrupt except TRAP can be disabled or masked.

4.9 *What are vectored and non-vectored interrupt?*

When an interrupt is accepted, if the processor control branches to a specific address defined by the manufacturer, then the interrupt is called vectored interrupt.

In non-vectored interrupt, there is no specific address for storing the interrupt service routine. Hence, the interrupting device should give the address of the interrupt service routine.

4.10 *List the software and hardware interrupts of 8085.*

Software interrupts : RST 0, RST 1, RST 2, RST 3, RST 4, RST 5, RST 6 and RST 7.

Hardware interrupts : TRAP, RST 7.5, RST 6.5, RST 5.5 and INTR.

4.11 *What is TRAP?*

TRAP is a non-maskable interrupt of 8085. It is not disabled by processor reset or after recognition of interrupt.

4.12 *Does HOLD has higher priority than TRAP or not?*

The interrupts including TRAP are recognized only if the HOLD is not valid, hence TRAP has lower priority than HOLD.

4.13 *What is masking and why it is required?*

Masking is preventing the interrupt from disturbing the current program execution. When the processor is performing an important job (process) and if the process should not be interrupted then all the interrupts should be masked or disabled.

In processor with multiple interrupts, the lower priority interrupt can be masked so as to prevent it from interrupting, the execution of interrupt service routine of higher priority interrupt.

4.14 *When does the 8085 processor accept a hardware interrupt?*

The processor keeps on checking the interrupt pins at the second T-state of the last machine cycle of every instruction. If the processor finds a valid interrupt signal and if the interrupt is unmasked and enabled then the processor accepts the interrupt. The acceptance of the interrupt is acknowledged by sending an  $\overline{\text{INTA}}$  signal to the interrupting device.

4.15 *List the type of signals that have to be applied to initiate a hardware interrupt in 8085.*

The TRAP is level and edge-sensitive and so the interrupt signal has to take a **low** to **high** transition and then remain **high** until it is recognized. The RST 7.5 is edge-sensitive and so the interrupt signal has to take a **low** to **high** transition and need not remain **high** until it is recognized. The RST 6.5, RST 5.5 and INTR are level-sensitive and so the interrupt signal should be **high** until the interrupt is recognized.

4.16 *What are maskable and non-maskable interrupts of 8085?*

The TRAP is non-maskable interrupt. The RST 7.5, RST 6.5 and RST 5.5 are maskable interrupts. The INTR of 8085 can also be disabled by DI instruction.

4.17 When will the 8085 processor disable the interrupt system ?

The interrupts of 8085 except TRAP are disabled after any one of the following operations.

- Executing EI instruction.
- System or processor reset.
- After recognition (acceptance) of an interrupt.

4.18 What is the function performed by DI instruction?

The function of DI instruction is to disable the entire interrupt system.

4.19 What is the function performed by EI instruction?

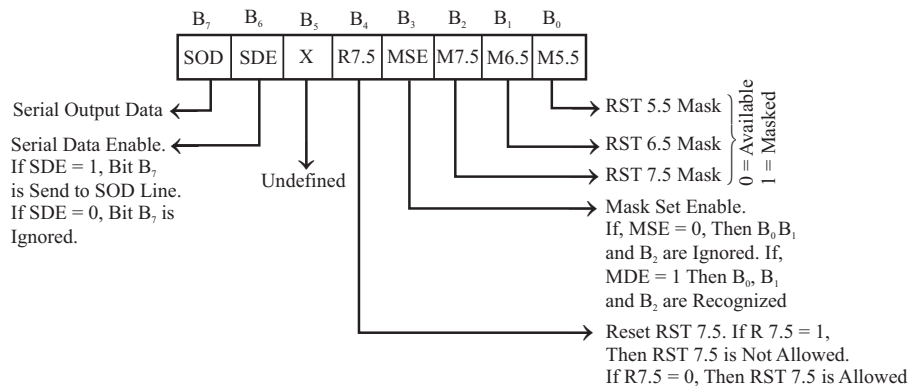
The EI instruction can be used to enable all the interrupts after disabling.

4.20 How can the interrupt INTR of 8085 be expanded?

The interrupt INTR of 8085 can be expanded upto eight interrupts using 8-to-3 priority encoder. It can also be expanded to eight interrupts using one number of 8259 (Programmable interrupt controller) or upto 64 interrupts using 8259's in cascaded mode.

4.21 How can the hardware interrupt of 8085 be masked or unmasked?

The masking or unmasking of RST 7.5, RST 6.5 and RST 5.5 interrupts can be performed by moving an 8-bit data to accumulator and then executing SIM instruction. The format of the 8-bit data is shown in Fig. Q4.21.



**Fig. Q4.21** : Format of an 8-bit data to be loaded in the accumulator before executing a SIM instruction.

4.22 How can we check whether an 8085 interrupt is masked or not?

The masking status of an 8085 interrupt can be obtained by executing RIM instruction. When RIM instruction is executed, a 8-bit data is loaded in the accumulator. The bits B<sub>0</sub>, B<sub>1</sub> and B<sub>2</sub> will give the masking status of RST 5.5, RST 6.5 and RST 7.5 respectively. If this bit is 1, then the corresponding interrupt is masked, otherwise it is unmasked.

4.23 How can we check the interrupt request pending status of 8085 interrupt?

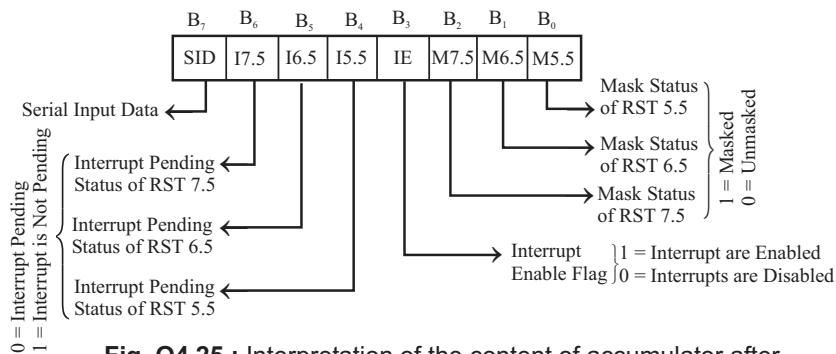
The pending status of an 8085 interrupt can be obtained by executing RIM instruction. When the RIM instruction is executed an 8-bit data is loaded in accumulator. The bits B<sub>4</sub>, B<sub>5</sub> and B<sub>6</sub> will give the pending status of RST 5.5, RST 6.5 and RST 7.5 respectively. If this bit is 1, then the interrupt is pending, otherwise it is not pending.

## 4.24 What is vectoring?

Vectoring is the process of generating the address of interrupt service routine to be loaded in program counter.

## 4.25 How can the status of maskable interrupts be read in 8085 processor?

The status of hardware interrupts like interrupt request pending or not, interrupts enabled or not, and masked or unmasked can read from accumulator after executing RIM instruction. When RIM instruction is executed an 8-bit data is loaded in accumulator which can be interpreted as shown in Fig. Q4.25.



**Fig. Q4.25 :** Interpretation of the content of accumulator after executing RIM instruction.

## 4.26 How are vector addresses generated for hardware interrupts of 8085?

For the hardware interrupts TRAP, RST 7.5, RST 6.5 and RST 5.5 the vector addresses are generated by the processor itself. These addresses are fixed by the manufacturer.

## 4.27 How is a vector address generated for the INTR interrupt of 8085?

For the INTR interrupt, the interrupting device has to place either RST opcode or CALL opcode followed by a 16-bit address. If RST opcode is placed, then the corresponding vector address is generated by the processor. In case of CALL opcode the given 16-bit address will be the vector address.

## 4.28 How are vector addresses generated for software interrupts of 8085?

For the software interrupts RST 0 to RST 7, the vector addresses are generated internal to the processor. These vector addresses are fixed by the manufacturer.

## 4.29 What is polling ?

Polling is a scheme or an algorithm to identify the devices interrupting the processor. Polling is employed when multiple devices interrupt the processor through one interrupt pin of the processor.

## 4.30 What are the different types of polling?

Polling can be classified into software and hardware polling. In software polling the entire polling process is governed by a program. In hardware polling, the hardware takes care of checking the status of interrupting devices and allowing the requests one by one to the processor.

## 4.31 What is the need for interrupt controller?

An interrupt controller is employed to expand the interrupt input. It can handle the interrupt request from various devices and allow the requests one by one to the processor.



4.32 List some of the features of INTEL 8259 (Programmable Interrupt Controller).

- It manages eight interrupt requests.
- The priorities of interrupts are programmable.
- Interrupt vector addresses are programmable.
- Interrupts can be masked or unmasked individually.

4.33 Write the various functional blocks of INTEL 8259 ?

The various functional blocks of 8259 are Control logic, Read/ Write logic, Data bus buffer, Interrupt Request Register (IRR), Interrupt Mask Register (IMR) and In-Service Register (ISR), Priority Resolver (PR) and Cascade buffer.

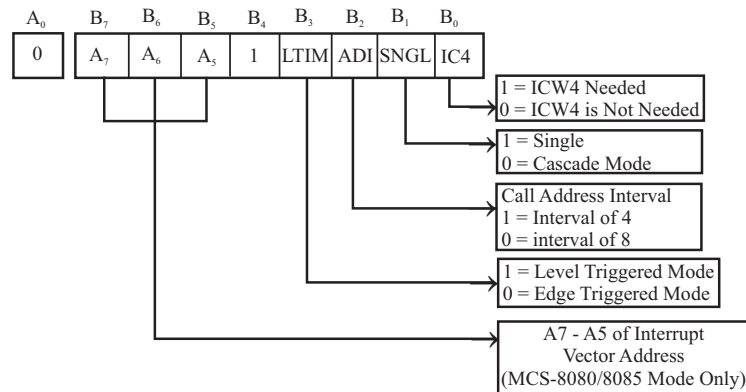
4.34 What is master and slave 8259 ?

When 8259's are connected in cascade, one 8259 will be directly interrupting 8085 and it is called master 8259. To each interrupt request input of master 8259, one slave 8259 can be connected. The 8259's interrupting the master 8259 are called slave 8259.

4.35 How is 8259 programmed?

The 8259 is programmed by sending Initialization Command Words (ICWs) and Operational Command Words (OCWs).

4.36 Write the format of ICW1?



4.37 What are the features of 8259 that are programmed using ICW's?

The ICW's are used to program the following features of 8259:

- Call address interval (in case of 8085).
- 8085 or 8086 mode.
- Cascade mode or single.
- Auto or Normal end of interrupt.
- Level or Edge triggered.
- Special fully nested mode.
- Vector address / Type number.

4.38 What are the features of 8259 that can be programmed using OCW's?

The OCW's are used to program the following features of 8259:

- Masking of individual interrupts.
- Specific or Non-specific end of the interrupt.
- Priority modes.

4.39 What is the difference between programming master 8259 and slave 8259 ?

The ICW 3 will be different for master 8259 and slave 8259. For master, the ICW3 will inform the IR input that are having slaves. For slave, the ICW3 will inform its slave ID number.

4.40 When is ICW4 sent to 8259 ?

The ICW4 is sent to 8259 to perform any one of the following features :

- 8085 or 8086 mode.
- Special fully nested mode.
- Auto or Normal end of interrupt.
- Buffered or Non-buffered mode.

4.41 Write a program segment to initialize a single 8259 connected to 8085 processor.

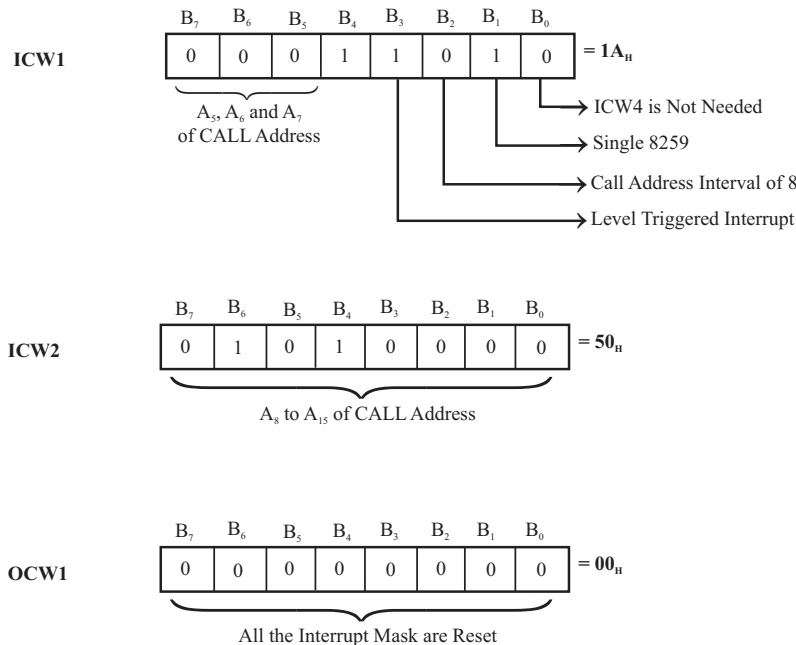
Let us assume that 8259 is IO-mapped in the system. The 8259 can be initialized by sending ICW1, ICW2 and OCW1. Let the 8-bit address when  $A_0 = 0$  be  $00_H$  and when  $A_0 = 1$  be  $01_H$ .

```

MVI A,ICW1 ; Move ICW1 to A-register.
OUT 00H    ; Send ICW1 to 8259.
MVI A,ICW2 ; Move ICW2 to A-register.
OUT 01H    ; Send ICW2 to 8259.
MVI A,OCW1 ; Move OCW1 to A-register.
OUT 01H    ; Send OCW1 to 8259.
HLT        ; Halt program execution.

```

4.42 Frame the Command words ICW1, ICW2 and OCW1 for initializing single 8259 with call address interval of 8 and for level triggered interrupt. Also, unmask all the interrupt inputs. The desired vector address is  $5000_H$ .



# ASSEMBLY LANGUAGE PROGRAMMING

## 5.1 LEVELS OF PROGRAMMING

Programs are a set of instructions or commands needed to perform a specific task by a programmable device such as a microprocessor. The programs needed for a programmable device can be developed at three different levels and they are as follows :

1. Machine level programming
2. Assembly level programming
3. High level programming

### Machine Level Programming

In machine level programming, instructions are written using binary codes which uses only two symbols '0' and '1'. The manufacturer of microprocessors will give a set of instructions for each microprocessor in binary codes, i.e., one binary code will represent one operation performed by the microprocessor. The language in which the instructions are represented by binary codes is called machine language. A microprocessor can understand and execute the machine language programs directly.

The binary instructions of one microprocessor will not be same as that of another microprocessor. Therefore, the machine language programs developed for one microprocessor cannot be used for another microprocessor i.e., the machine level programs are machine dependent. Moreover, it is highly tedious for a programmer to write programs in the machine language.

### Assembly Level Programming

In assembly level programming, instructions are written using mnemonics. A mnemonic comprises of a few letters of the English language which represent the operation performed by the instruction. For example, the mnemonic for the instruction which performs **addition** operation is **ADD**. The manufacturer of the microprocessors will provide a set of instructions in the form of a mnemonic for each microprocessor. Also, for each mnemonic a binary code will be specified by the manufacturer. If the program is developed using binary codes then it is called machine level programming and if the program is developed using mnemonics then it is called assembly level programming.

The language in which the instructions are represented by mnemonics is called assembly language. Microprocessors cannot execute the assembly language programs directly. The assembly language programs have to be converted to machine language for execution. This conversion is performed using a software tool called assembler.

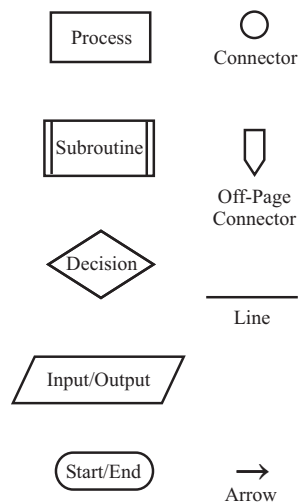
The mnemonics of one microprocessor will not be same as that of another microprocessor. Therefore, the assembly language programs developed for one microprocessor cannot be used for another microprocessor directly i.e. the assembly language programs are machine dependent. But certain manufacturers provide upward compatability for the same family of microprocessors. (i.e., the program developed for a the lower version of a microprocessor of a family can be run on the higher version without modifications.) For example, consider the INTEL 80x86 family of microprocessors. The program developed for 8086 microprocessor can be run on 80186, 80286, 80386 or 80486 microprocessor-based system without any modifications.

### High Level Programming

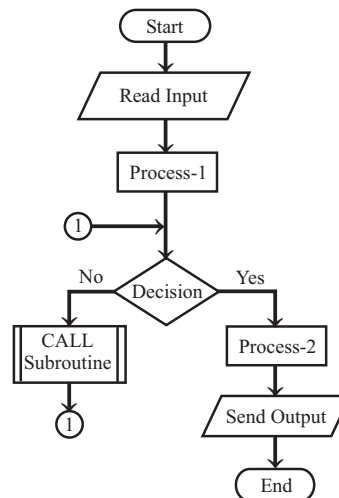
In high level programming the instructions will be in the form of statements written using symbols, English words and phrases. Each high level language will have its own vocabulary of words, symbols, phrases and sentences. Examples of high level languages are BASIC, C, C++, etc. The programs written in high level languages are easy to understand and machine independent. So they are known as portable programs. A high level language program has to be converted into machine language programs in order to be executed by the microprocessor. This conversion is performed by a software tool called compiler.

## 5.2 FLOWCHART

Flowchart is a graphical representation of the operation flow of a program. It is also the graphical form of an algorithm. Flowcharts can be a valuable aid in visualizing programs. The various symbols used for drawing flowcharts are shown in Fig. 5.1. The operations represented by various symbols of flowchart are explained in Table-5.1. A sample flowchart is shown in Fig. 5.2.


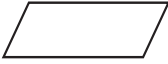


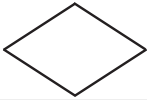






**Fig. 5.1 :** Symbols used in a flowchart.



**Fig. 5.2 :** A sample flowchart.

**TABLE - 5.1 : OPERATIONS REPRESENTED BY THE SYMBOLS USED IN FLOWCHART**

Symbol	Operation
Racetrack shape box 	A racetrack shaped symbol is used to indicate the beginning (start) or end of a program.
Parallelogram 	A parallelogram is used to represent input or output operation.
Rectangular box 	A rectangular box is used to represent simple operations other than input and output operations.
A rectangular box with double lines on vertical sides 	A rectangular box with double lines on vertical sides is used to represent a subroutine or procedure.
Diamond shaped box 	A diamond shaped box is used to represent a decision point or cross road in the programs
Small circle 	A small circle is used as a connector to show the connections between various parts of a flowchart within a page. Identical numbers are entered inside the circles that represent the same connecting points.
Five-sided box 	A five-sided box symbol is used as an off-page connector to show the connections between various sections of a flowchart in different pages. Identical numbers are entered inside the boxes that represent the same connecting point.
Line 	Lines are drawn between boxes and diamonds to indicate the program flow.
Arrow 	Arrows are placed on the lines to indicate the direction of program flow.

### 5.3 ASSEMBLY LANGUAGE PROGRAM DEVELOPMENT TOOLS

Development system is used by system designers to design and test the software and hardware of a microprocessor-based system before going for practical implementation (or fabrication). The microprocessor development system consists of a set of hardware and software tools. The hardware of a development system usually contain a standard PC (Personal Computer), printer and an emulator. The software tools are also called program development tools and they are editor, assembler, library builder, linker, debugger and simulator. These software tools can be run on a PC in order to write, assemble, debug, modify and test the assembly language programs.

### **Editor (Text Editor)**

Editor is a software tool which, when run on a PC, allows the user to type/enter and modify the assembly language program. The editor provides a set of commands for insertion, deletion and modification of letters, characters, statements, etc. The main function of an editor is to help the user to construct the assembly language program in the right format. The program created using editor is known as source program and it is usually saved with the file extension ".ASM". For example, if a program for addition is developed using editor then it can be saved as "ADDITION.ASM". Some examples of editors are NE (Norton Editor), EDIT (DOS Editor), etc.

### **Assembler**

The assembler is a software tool which, when run on a PC, converts the assembly language program to a machine language program. Several types of assemblers are available and they are one-pass assembler, two-pass assembler, macro assembler, cross assembler, resident assembler and meta assembler.

In one-pass assembler the source code is processed only once and we can use only backward reference. In a one-pass assembler as the source code is processed, any labels encountered are given an address and stored in a table. Whenever a label is encountered, the assembler may look backward to find the address of the label. If the label is not yet defined then it issues an error message (because the assembler will not look forward). Since only one pass is used to translate the source code, a one-pass assembler is very fast, but because of the forward reference problem, the one-pass assembler is not used often.

Most of the popularly used assemblers are the two-pass assemblers. In a two-pass assembler, the first pass is made through source code for the purpose of assigning an address to all the labels and to store this information in a symbol table. The second pass is made to actually translate the source code into machine code.

The input for the assembler is the source program which is saved with file extension ".ASM". The assembler usually generates two output files called object file and list file. The object file consist of relocatable machine codes of the program and it is saved with file extension ".OBJ". The list file contains the assembly language statements, the binary codes for each instruction and address of each instruction. The list file is saved with file extension ".LST".

The list file also indicates any syntax errors in the source program. The assembler will not identify the logical errors in the source program. In order to correct the errors indicated on the list file, the user have to use the editor again. The corrected source program is saved again and then reassembled. Usually, it may take several times through edit-assemble loop to eliminate the syntax errors from the source program.

Some examples of assemblers are TASM (Borland's Turbo Assembler), MASM (Microsoft's Macro Assembler), ASM86 (INTEL'S 8086 Assembler), ASM85 (INTEL'S 8085 Assembler), etc.

### **Advantages of the assembler**

1. The assembler translates mnemonics into binary code with speed and accuracy, thus eliminating human errors in looking up the codes.
2. The assembler assigns appropriate values to the variables used in a program. This feature offers flexibility in specifying jump locations.
3. It is easy to insert or delete instructions in a program and reassemble the entire program quickly with new memory locations and modified addresses for jump locations. This avoids rewriting the program manually.
4. The assembler checks syntax errors, such as wrong labels, opcodes, expressions, etc., and provides error messages. However, it cannot check logic errors in a program.
5. The assembler can reserve memory locations for data or results.
6. The assembler provides list file for documentation.

### **Library Builder**

The library builder is used to create library files which are a collection of procedures of frequently used functions. Actually a library file is a collection of assembled object files. While developing a software for a particular application, the programmers can link the library files in their programs. When the library file is linked with a program, only the procedure required by the program are copied from library file and added to the program.

The input to library builder is a set of assembled object files of program modules/procedures. The library builder combines the program modules/procedures into a single file known as library file and it is saved with file extension ".LIB". Some examples of library builder are microsoft's LIB, Borlands TLIB, etc.

### **Linker**

The linker is a software tool which is used to combine relocatable object files of program modules and library functions into a single executable file.

While developing program for a particular application it is much more efficient to develop the program in modules. The entire task of the program can be divided into smaller tasks and procedures for each task can be developed individually. These procedures are called program modules. For certain tasks we can use library files if they are available. Each module can be individually assembled, tested and debugged. Then the object files of program modules and the library files can be linked to get an executable file.

The linker also generates a link map file which contains the address information about the linked files. Some examples of linkers are microsoft's linker LINK, Borland's Turbo linker TLINK, etc.

### **Debugger**

Debugger is a software tool that allows the execution of a program in a single step or break-point mode under the control of user. The process of locating and correcting the errors in a program using a debugger is known as debugging.

The debugger allows the designer to load the object code program into the memory of the PC, execute the program and troubleshoot or debug it. The debugger allows the designer to look at the contents of registers and memory locations after running the program. It allows the system designer to change the contents of registers and memory locations and return the program.

Some debuggers allow the user to stop execution after each instruction so that the memory/register content can be checked or altered. A debugger also allows the user to set a breakpoint at any point in user program. When the user runs the program, the PC will execute instructions up to this breakpoint and stop. The user can then examine register and memory contents to see whether the results are correct upto that point. If the results are correct, the user can move the breakpoint to a later point in the program. If the results are not correct, the user can check the program up to that point to find out why they are not correct.

Debugger tools can help the user to isolate a problem in the program. Once the problem/errors are identified, the algorithm can be modified. Then the user can use the editor to correct the source program, reassemble the corrected source program, relink and run the program again.

### **Simulator**

The simulator is a program which can be run on the development system (Personal computer) to simulate the operations of the newly designed system. Some of the operations that can be simulated are given below:

- Execute a program and display result.
- Single step execution of a program.
- Break-point execution of a program.
- Display the contents of register/memory.

Simulator usually shows the contents of registers and memory locations on the screen of the computer and allows the system designer to perform all of the operations listed above, with the added advantage of watching the data change as the program operates. This feature saves considerable time because the register/memory contents do not have to be displayed using separate commands. The visual representation also gives the programmer a better feel for what is taking place in the program.

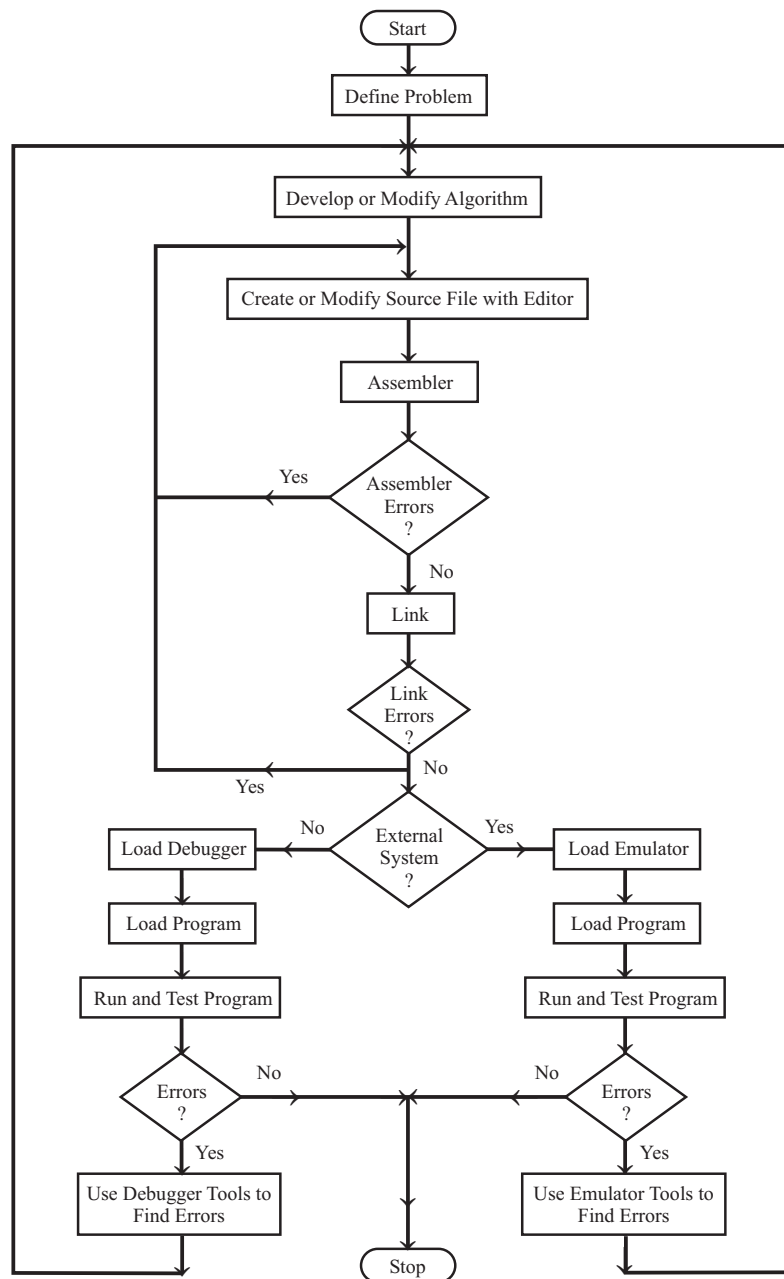
The simulators do not have the ability to perform actual IO or internal hardware operations such as timing or data transmission and reception.

### **Emulator**

An emulator is a mixture of hardware and software. It is usually used to test and debug the hardware and software of a newly designed microprocessor-based system. The emulator has a multicore cable which connects the PC of the development system and the newly designed hardware of the microprocessor system. A connector/plug at one end of the cable is plugged into new hardware in place of its microprocessor. The other end of cable is connected to parallel port of PC. Through this connection the software of the emulator allows the designer to download the object code program into RAM in the system being tested and run it.

Like a debugger, an emulator allows the system designer to load and run programs, examine and change the contents of registers, examine and change the contents of memory locations and insert breakpoints in the program.





**Fig. 5.3 :** Development process of an assembly language program.

The emulator also takes a snapshot of the contents of registers, activity on the address and data bus and the state of the flags as each instruction executes. Also, the emulator stores this trace data. The user can have a printout of the trace data to see the results that the program produced on a step-by-step basis. Another powerful feature of an emulator is the ability to use either development system memory or the memory on the hardware under test for the program that is being debugged.

### **Summary of the use of program Development Tools**

The various steps in the development of an assembly language program are given below and also as a flowchart in Fig. 5.3.

1. Define the problem carefully.
2. Use an editor to create the source file for assembly language program.
3. Assemble the source file with the assembler.
4. If assembler list file indicates errors then use editor and correct the errors.
5. Cycle through the edit - assemble loop until all errors indicated by assembler are cleared.
6. Use linker to link all object files of program modules and library files into a single executable file.
7. If the linker indicates any error then modify the source program, reassemble and relink to correct the errors.
8. If the developed program does not interact with any external hardware other than that directly connected to the system, then you can use the system debugger to run and debug your program.
9. If the designed program is intended to work with external hardware then use an emulator to run and debug the program.

## **5.4 VARIABLES AND CONSTANTS USED IN ASSEMBLERS**

The various characters used to construct assembler variables, constants and directives are the following :

Upper case English alphabets	: A to Z
Lower case English alphabets	: a to z
Numbers	: 0 to 9
Special characters	: @, \$, ?, _ (Underscore)

### **Variables**

Variables are symbols (or terms) used in assembly language program statements in order to represent the variable data and address. While running a program, a value has to be attached to each variable in the program. The advantage of using variables is that the value of the variable can be dynamically varied while running the program.

Usually a variable name is constructed such that it reflects the meaning of the value it holds. A variable name selected to represent the temperature of a device can be TEMP, a variable name selected to represent the speed of a motor can be M\_SPEED, etc. While constructing variable names, the numeric characters (0 to 9) should not be used as first character and the special characters \$ and ? should not be used.

### **Rules for framing variable names**

1. The variable name can have any of the following characters. A to Z, a to z, 0 to 9, @, \_ (underscore).
2. The first character in the variable name should be an alphabet ( A to Z or a to z) or an underscore.
3. The length of a variable name depends on the assembler and normally the maximum length of variable name is 32 characters.
4. Variable names are case insensitive. Therefore, the assembler does not distinguish between the upper and lower case letters/alphabets.

### Constants

The decimal, binary or hexadecimal number used to represent the data or address in assembly language program statement are called constants or numerical constants. When constants are used to represent the address/data then their values are fixed and cannot be changed while running a program. The binary, hexadecimal and decimal constants can be differentiated by placing a specific alphabet at the end of the constant.

A valid binary constant/number is framed using numeric characters 0 and 1 and the alphabet B is placed at the end.

A valid decimal (BCD) constant/number is framed using numeric characters 0 to 9 and the alphabet D is placed at the end. However, a constant/number which does not end with any alphabet is also treated as a decimal constant.

A valid hexadecimal constant/number is framed using numeric characters 0 to 9 and alphabets A to F and the alphabet H is placed at the end. A zero should be placed/inserted at the beginning of a hexadecimal number if the first digit is an alphabet character from A to F, otherwise the assembler will consider the constant starting with A to F as a variable.

#### Examples of valid constants

1011	-	Decimal (BCD) constant
1060D	-	Decimal constant
1101B	-	Binary constant
92ACH	-	Hexadecimal constant
0E2H	-	Hexadecimal constant

#### Examples of invalid constants

1131B	-	The character 3 should not be used in binary constant.
0E2	-	The character H at the end of hexadecimal number is missing.
C42AH	-	Zero is not inserted in the beginning of hexadecimal number and so it is treated as a variable.
1A65D	-	The character A should not be used in decimal constant.

## 5.5 ASSEMBLER DIRECTIVES

The assembler directives are the instructions to the assembler regarding the program being assembled. They are also called pseudo instructions or pseudo opcodes. The assembler directives are used to specify start and end of a program, attach value to variables, allocate storage locations for input/output data, to define start and end of segments, procedures, macros, etc.

The assembler directives control the generation of machine code and organization of the program. But no machine codes are generated for assembler directives. Some of the assembler directives that can be used for 8085 assembly language program development are listed in Table-5.2.

**TABLE - 5.2 : ASSEMBLER DIRECTIVES**

S.No	Assembler directives	Functions
1.	DB	Define Byte. Used to define byte type variable.
2.	DW	Define word. Used to define 16-bit variable.
3.	END	Indicates the end of the program.
4.	ENDM	End of macro. Indicates the end of a macro sequence.
5.	EQU	Equate. Used to equate numeric value or constant to a variable.
6.	MACRO	Defines the name, parameters and start of a macro.
7.	ORG	Origin. Used to assign the starting address for a program.

**DB (DEFINE BYTE)**

The directive DB is used to define a byte type variable. It reserves specific amount of memory to variables and stores the values specified in the statement as initial values in the allotted memory locations. The range of value that can be stored in a byte type variable is 0 to  $255_{10}$  ( $00_H$  to  $FF_H$ ) for unsigned value and  $-128_{10}$  to  $127_{10}$  for signed value ( $00_H$  to  $7F_H$  for positive values and  $80_H$  to  $FF_H$  for negative values).

The general form of the statement to define the byte variable is,

**variable DB value/values**

Examples :	
<b>AREA DB 45</b>	One memory location is reserved for the variable AREA and $45_{10}$ is stored as initial value in that memory location.
<b>LIST DB 7FH, 42H, 35H</b>	Three consecutive memory locations are reserved for the variable LIST and $7F_H$ , $42_H$ and $35_H$ are stored as initial value in the reserved memory location.

**DW (DEFINE WORD)**

The directive DW is used to define a word type (16-bit) variable. It reserves two consecutive memory locations to each variable and stores the 16-bit values specified in the statement as the initial value in the allotted memory locations. The range of values that can be stored in word type variable is 0 to  $65535_{10}$  ( $0000_H$  to  $FFFF_H$ ) for unsigned value, and  $-32768$  to  $+32767$  for signed value ( $0000_H$  to  $7FFF_H$  for positive value and  $8000_H$  to  $FFFF_H$  for negative value).

The general form of the statement to define the word type variable is,

**variable DW value/values**

Examples :	
<b>WEIGHT DW 1250</b>	Two consecutive memory locations are reserved for the variable WEIGHT and initialized with value $1250_{10}$ .
<b>ALIST DW 6512H, 0F251H, 0CDE2H</b>	Six consecutive memory locations are reserved for the variable ALIST and each 16-bit data specified in the instruction is stored in two consecutive memory locations.

**ORG, END AND EQU**

The directive ORG (Origin) is used to assign the starting address for a program. The directive END is used to terminate a program. The statements after the directive END will be ignored by the assembler.

The directive EQU (Equate) is used to attach a value to a variable.

Examples :	
<b>ORG 1000H</b>	This directive informs the assembler that the statements following ORG 1000H should be stored in memory starting with address $1000_H$ .
<b>PORT1 EQU 0F2H</b>	The value of variable PORT1 is $F2_H$ .
<b>LOOP EQU 10FEH</b>	The value of variable LOOP is $10FE_H$ .

## 5.6 PROCEDURE AND MACRO

### Procedure or Subroutine

When a group of instructions are to be used several times to perform a same function in a program, then we can write them as a separate subprogram called procedure or subroutine. Whenever required the procedures can be called in a program using CALL instructions.

Procedures are written and assembled as separate program modules and stored in memory. When a procedure is called in the main program, the program control is transferred to the procedure and after executing the procedure the program control is transferred back to the main program. In 8085 processor, the instruction CALL is used to call a procedure in the main program and the instruction RET is used to return the control to the main program.

The main advantage of using a procedure is that the machine codes for the group of instructions in the procedure has to be put in memory only once. The disadvantages of using the procedure are the need for a stack and the overhead time required to call the procedure and return to the calling program.

### Handling procedure

While executing a program, if the 8085 processor encounters a CALL the instruction, then it saves the content of the program counter in a stack and loads the subroutine address in the program counter. (The content of program counter that is saved in stack is the address of the instruction next to CALL in the main program. The subroutine address is the address given in the CALL instruction.)

When the subroutine address is loaded in the program counter, the processor starts executing the subroutine. The last instruction of the subroutine will be RET instruction and when it is executed, the processor moves the top of the stack memory to the program counter. (The top of stack memory is the address which is saved in stack before executing subroutine.) Now the program control (execution) is returned to main program.

The subroutine program may use the registers that are used by the main program. If in the main program the content of these registers are to be preserved then they have to be saved (PUSHed) in stack before calling the subroutine. After returning from subroutine, they can be retrieved (POPed) from the stack back to the respective register. In 8085 the type of stack is LIFO (Last-In-First-Out). Hence, the order of retrieving (POPing) should be opposite to that of storing (PUSHing). For example, if the content of register pair HL is stored first followed by DE then while retrieving the DE pair should be popped first followed by HL pair.

### Macro

When a group of instructions are to be used several times to perform a same function in a program and they are too small to be written as a procedure, then they can be defined as a macro. A macro is a small group of instructions enclosed by the assembler directives MACRO and ENDM. Macros are identified by their names and usually defined at the start of a program.

A macro is called by its name in the program. Whenever a macro is called in the program, the assembler will insert the defined group of instructions in place of the macro. In other words the macro call is like short hand expression which tells the assembler, *"Every time you see a macro name in the program, replace it with the group of instructions defined as macro"*. Actually the assembler generates machine codes for the group of instructions defined as macro, whenever it is called in the program. The process of replacing the macro with the instructions it represents is called expanding the macro. Hence macros are also known as open subroutines because, they get expanded at the point of macro invocation.

When macros are used, the generated machine codes are right-in-line with the rest of the program and so the processor does not have to go off to a procedure call and return. This results in avoiding the overhead time involved in calling and returning from a procedure. The disadvantage of using macro is that the program may take up more memory due to insertion of the machine codes in the program at the place of macros. Hence the macros should be used only when its body has a few program statements.

**TABLE - 5.3 : COMPARISON OF PROCEDURE AND MACRO**

Procedure	Macro
1. Accessed by CALL and RET mechanism during program execution.	1. Accessed during assembly with name given to macro when defined.
2. Machine code for instructions are stored in memory once.	2. Machine codes are generated for instructions in the macro each time it is called.
3. Parameters are passed in registers, memory locations or stack.	3. Parameters are passed as part of statement which calls macro.

## 5.7 DELAY ROUTINE

Delay routines are the subroutines used for maintaining the timings of various operations in a microprocessor. In control applications, certain equipment need to be ON/OFF after a specified time delay. In some applications, a certain operation has to be repeated after a specified time interval. In such cases simple time delay routines can be used to maintain the timings of the operations.

A delay routine is generally written as a subroutine (It need not be a subroutine always. It can even be a part of the main program.) In a delay routine a count (number) is loaded in a register of microprocessor. Then it is decremented by one and the zero flag is checked to verify whether the content of register is zero or not. This process is continued until the content of the register is zero. When it is zero the time delay is over and the control is transferred to the main program to carry out the desired operation.

The delay time is given by the total time taken to execute the delay routine. It can be computed by multiplying the total number of T states required to execute the subroutine and the time for one T-state of the processor. The total of number of T states can be computed from the knowledge of T states required for each instruction. The time for one T-state of the processor is given by the inverse of the internal clock frequency of the processor. For example, if the 8085 microprocessor has 5 MHz quartz crystal then,

$$\text{The internal clock frequency} = \frac{5}{2} = 2.5 \text{ MHz}$$

$$\text{Time for one T-state} = \frac{1}{2.5 \times 10^6} = 0.4 \text{ msec}$$

Two example delay routines that can be used in 8085 assembly language programs are presented in this section with details of timing calculations. For small time delays (< 0.5 millisecond) an 8-bit register can be used as counter, but for large time delays (< 0.5 second) 16-bit register should be used as counter. For very large time delays (> 0.5 second), a delay routine can be repeatedly called in the main program. The disadvantage in delay routines is that the processor time is wasted. An alternate solution is to use a dedicated timer like 8253/8254 to produce time delays or to maintain timings of various operations.

**EXAMPLE DELAY ROUTINE - 1**

*Write a delay routine to produce a time delay of 0.5 millisecond in 8085 processor-based system whose clock source is 6 MHz quartz crystal.*

**Solution**

The delay required is 0.5 millisecond, hence an 8-bit register of 8085 can be used to store a count value. The count is decremented by one and the zero flag is verified. If zero flag is set then decrement operation is terminated. The delay routine is written as a subroutine as shown below:

**Delay routine**

```

MVI D,N    ; Load the count value, N in D-register.
LOOP: DCR D ; Decrement the count.
      JNZ LOOP ; If count is not zero go to LOOP.
      RET      ; If count is zero return to main program.

```

The following table shows the T-state required for execution of the instructions in the subroutine.

Instruction	T-state required for execution of an instruction	Number of times the instruction is executed	Total T states
CALL addr16	18	1	$18 \times 1 = 18$
MVI D,N	7	1	$7 \times 1 = 7$
DCR D	4	N times	$4 \times N = 4N$
JNZ LOOP	10	(N-1) times	$10 \times (N-1) = 10N - 10$
	or 7	1	$7 \times 1 = 7$
RET	10	1	$10 \times 1 = 10$
Total T-state required for subroutine			$= 14N + 32$

**Calculation to find the count value, N**

External Clock frequency = 6 MHz

$$\text{Internal Clock frequency} = \frac{\text{External Clock}}{2} = \frac{6}{2} = 3 \text{ MHz}$$

$$\text{Time period of one T-state} = \frac{1}{\text{Internal Clock frequency}} = \frac{1}{3 \times 10^6} = 0.3333 \mu\text{s}$$

$$\left. \begin{array}{l} \text{Number of T states} \\ \text{required for 0.5 ms} \end{array} \right\} = \frac{\text{Required time delay}}{\text{Time for one T-state}} = \frac{0.5 \times 10^{-3}}{0.3333 \times 10^{-6}} = 1500.15 = 1500_{10}$$

On equating the total T states required for the subroutine and the number of T states for the required time delay, the count value, N can be calculated.

$$\therefore 14N + 32 = 1500_{10}$$

$$N = \frac{1500 - 32}{14} = 104.857_{10} \approx 105_{10} = 69_H$$

$$\therefore \text{Count value, } N = 69_H$$

If the above delay routine is called by a program and executed with count value of  $69_H$  then the delay produced will be 0.5 millisecond.

**Note :** The register used in the delay routine is D-register. Also the execution of delay routine will alter the flags. Hence, if the contents of these registers are to be preserved, the main program has to save them in the stack before calling the delay routine.

### EXAMPLE DELAY ROUTINE - 2

Write a delay routine to produce a time delay of 0.5 second in 8085 processor-based system whose internal clock frequency is 3 MHz.

#### Solution

The delay required is large, hence a 16-bit register can be used for storing the count value. The count is decremented one by one until it is zero. After each decrement operation we have to verify whether the content of register pair is zero or not. This can be performed by logically ORing the content of low order and high order register and then checking the zero flag. (Because the 16-bit increment/decrement instruction will not modify any flag.) The delay routine is written as a subroutine as shown below:

#### Delay Routine

```

LXI D,N ; Load the count value, N in DE-register pair.
LOOP: DCX D ; Decrement the count.
      MOV A,E ; Logically OR the content of
      ORA D ; E-register with D-register.
      JNZ LOOP ; If count is not zero, go to LOOP.
      RET ; If count is zero, return to main program.

```

The following table shows the T states required for execution of the instructions in the subroutine.

Instructions	T-state required for the execution of an instruction	Number of times the instruction is executed	Total T states
CALL addr16	18	1	$18 \times 1 = 18$
LXID,N	10	1	$10 \times 1 = 10$
DCXD	6	N times	$6 \times N = 6N$
MOVA,E	4	N times	$4 \times N = 4N$
ORAD	4	N times	$4 \times N = 4N$
JNZ LOOP	10	(N-1) times	$10 \times (N-1) = 10N - 10$
	or 7	1	$7 \times 1 = 7$
RET	10	1	$10 \times 1 = 10$
Total T states required for subroutine			$= 24N + 35$



**Calculation to find the count value, N**

$$\text{Internal Clock frequency} = 3 \text{ MHz}$$

$$\text{Time period of one T-state} = \frac{1}{\text{Internal Clock frequency}} = \frac{1}{3 \times 10^6} = 0.3333 \mu\text{s}$$

$$\left. \begin{array}{l} \text{Number of T states required} \\ \text{for 0.5 second} \end{array} \right\} = \frac{\text{Required time delay}}{\text{Time for one T-state}} = \frac{0.5 \text{ sec}}{0.3333 \times 10^{-6}}$$

$$= 1500150.015_{10} = 1500150_{10}$$

On equating the total T states required for the subroutine and the number of T states for the required time delay, the count value, N can be calculated.

$$\therefore 24N + 35 = 1500150_{10}$$

$$N = \frac{1500150 - 35}{24} = 62504.79_{10} \approx 62505_{10} = \text{F429}_{\text{H}}$$

$$\therefore \text{Count value, } N = \text{F429}_{\text{H}}$$

If the above delay routine is called by a program and executed with count value of F429<sub>H</sub> then the delay produced will be 0.5 second.

*Note : The registers used in the delay routine are A, D and E. Also the execution of delay routine will alter the flags. Hence if the contents of these register are to be preserved, then the main program has to save them in stack before calling the delay routine.*

**5.8 LIST AND ARRAY****List**

List is a linked data structure used in programming techniques. The linked data structure will have a number of components linked in a particular fashion. Each component will consist of a string data and a pointer to the next component. The basic idea of a linked data structure is that each component within the structure includes a pointer indicating where the next component can be found. Therefore, the relative order of the components can be changed by altering the pointers. In addition, individual components can be easily added or deleted, again by altering the pointer. As a result, a linked data structure is not confined to some maximum number of components, but whenever required the data structure can be expanded or contracted in size.

The different types of linked data structures are linear linked lists, linked lists with multiple pointers, circular linked lists and trees.

## Array

An array is a series of data of the same type stored in successive memory locations. Each value in the array is referred to as an element of the array. In programming techniques, array is created when we want to perform some operation on a series of data items.

### 5.9 STACK

The stack is a portion of RAM memory defined by the user for temporary storage and retrieval of data while executing a program. The microprocessor will have a dedicated internal register called **Stack Pointer (SP)** to hold the address of the stack. Also, the processor will have a facility to automatically decrement/increment the content of SP after every write/read operation into stack.

The user can initialize or create a stack by loading a RAM address in the **Stack Pointer (SP)**. Once an address is loaded in SP, the RAM memory locations below the address pointed by SP are reserved for stack. Typically 25 to 100 RAM memory locations are sufficient for stack. The user should take care that the reserved RAM memory locations for stack are not used for any other purpose.

The user has to create/implement a stack whenever the program consists of PUSH, POP, RST n, CALL and RET instructions. Also, the stack is needed whenever the system uses interrupt facility.

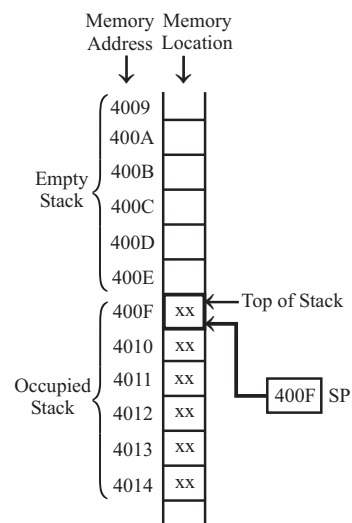
In a program, when the number of the available registers are not sufficient for storing intermediate result and data, then some of intermediate result and data can be stored in a stack using PUSH instruction and retrieved whenever required using POP instruction.

The CALL instruction and the interrupts store the return address (content of program counter) in the stack before executing the subroutine. Usually the subroutines are terminated with RET instruction. When RET instruction is executed, the top of stack is popped to program counter and the program control returns to the main program after the execution of subroutine.

#### 5.9.1 Stack in 8085 Microprocessor

In an 8085 processor, the stack is created by loading a 16-bit address in the stack pointer. Upon reset, the stack pointer is cleared to zero.

In an 8085 processor, for every write operation into stack, the SP is automatically decremented by two and for every read operation from stack, the SP is automatically incremented by two. Hence, data can be stored only in lower addresses from the address pointed by SP. Therefore, we can say that the SP holds the address of the top of stack. All the RAM addresses higher



**Fig. 5.4 :** Example of stack in an 8085.

than that pointed by the SP can be considered as occupied stack and all the RAM addresses lower than that pointed by the SP can be considered as empty stack as shown in Fig. 5.4. However, in practice only few memory locations are needed for stack.

In 8085 processor the content of register pairs can be stored in stack using PUSH instruction and the stored information can be retrieved back to register pair using POP instruction. When a number of register pairs have to be stored and retrieved in the stack, the order of retrieval should be reverse of that of the order of storage. For example, let BC pair be pushed to stack first and DE pair next. When the stored information has to be retrieved to appropriate registers, the top of stack should be popped to the DE pair first and then to the BC pair next. The storage and retrieval in stack are in reverse order, because the SP is decremented for every write operation into stack and SP is incremented for every read operation from stack. Therefore, the stack in 8085 is called Last-In-First-Out (LIFO) stack, i.e., the last stored information can be read first.

## 5.10 EXAMPLES OF 8085 ASSEMBLY LANGUAGE PROGRAMS

### EXAMPLE PROGRAM - 1: 8-Bit Addition

*Write an assembly language program to add two numbers of 8-bit data stored in memory locations 4200<sub>H</sub> and 4201<sub>H</sub> and store the result in 4202<sub>H</sub> and 4203<sub>H</sub>.*

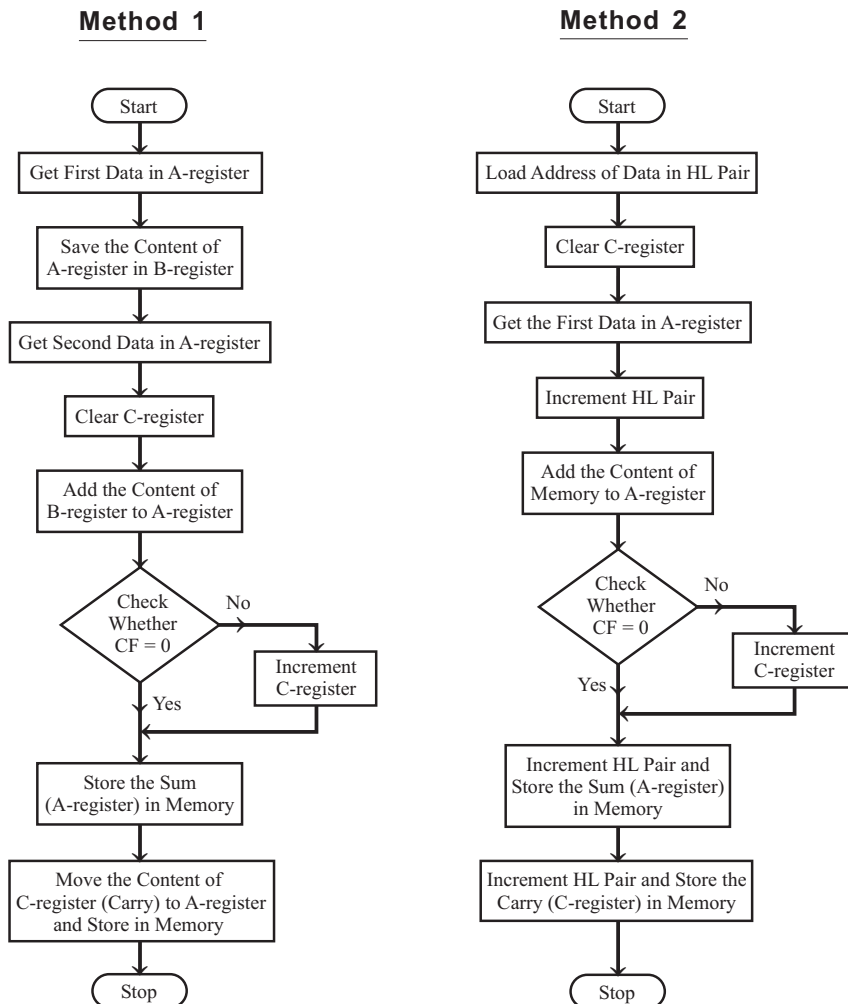
#### Problem Analysis

In order to perform addition in 8085, one of the data should be in accumulator and another data can be in any one of the general purpose register or in the memory. After addition, the sum is stored in the accumulator. The sum of two 8-bit data can be either 8 bits (sum only) or 9 bits (sum and carry). The accumulator can accommodate only the sum and if there is a carry, the 8085 will indicate by setting the carry flag. Hence, one of the registers is used to account for carry.

In method-1, direct addressing is used to address the data. But in method-2, register indirect addressing is used to the address data. Here, HL-register is used to hold the address of the data and it is called pointer.

#### Algorithm (Method-1)

1. Load the first data from memory to accumulator and move it to B-register.
2. Load the second data from memory to accumulator.
3. Clear C-register.
4. Add the content of B-register to accumulator.
5. Check for carry. If carry = 1, go to step 6 or if carry = 0, go to step 7.
6. Increment the C-register.
7. Store the sum in memory.
8. Move the carry to accumulator and store in memory.
9. Stop.

**Flowchart for example program 1****Algorithm (Method-2)**

1. Load the address of the data memory in HL pair (i.e., set HL pair as pointer for data).
2. Clear C-register.
3. Move the first data from memory to accumulator.
4. Increment the pointer (HL pair).

5. Add the content of memory addressed by HL with accumulator.
6. Check for carry. If carry = 1, go to step 7 or if carry = 0, go to step 8.
7. Increment the C-register.
8. Increment the pointer and store the sum.
9. Increment the pointer and store the carry.
10. Stop.

### Assembly language program (Method 1)

```
;PROGRAM TO ADD TWO 8-BIT DATA
;METHOD-1

ORG 4100H ;specify program starting address.

LDA 4200H ;Get 1st data in A and save in B.
MOV B,A
LDA 4201H ;Get 2nd data in A-register.
MVI C,00H ;Clear C-register to account for carry.
ADD B ;Get the sum in A-register.
JNC AHEAD ;If CF=0, go to AHEAD.
INR C ;If CF=1, increment C-register.
AHEAD: STA 4202H ;Store the sum in memory.
MOV A,C
STA 4203H ;Store the carry in memory.
HLT ;Halt program execution.

END ;Assembly end.
```

### Assembler listing for example program 1 (Method 1)

```
1 ;PROGRAM TO ADD TWO 8-BIT DATA
2 ;METHOD-1
3 0000
4 4100 ORG 4100H ;specify program starting address.
5
6 4100 3A 00 42 LDA 4200H ;Get 1st data in A and save in B.
7 4103 47 MOV B,A
8 4104 3A 01 42 LDA 4201H ;Get 2nd data in A-register.
9 4107 0E 00 MVI C,00H ;Clear C-register to account for carry.
10 4109 80 ADD B ;Get the sum in A-register.
11 410A D2 0E 41 JNC AHEAD ;If CF=0, go to AHEAD.
12 410D 0C INR C ;If CF=1, increment C-register.
13 410E 32 02 42 AHEAD: STA 4202H ;Store the sum in memory.
14 4111 79 MOV A,C
15 4112 32 03 42 STA 4203H ;Store the carry in memory.
16 4115 76 HLT ;Halt program execution.
17
18 4116 END ;Assembly end.
```

### Assembly language program (Method 2)

```
;PROGRAM TO ADD TWO 8-BIT DATA
;METHOD-2

ORG 4100H ;specify program starting address.

LXI H,4200H ;Set pointer for data.
MVI C,00H ;Clear C-register to account for carry.
MOV A,M ;Get 1st data in A-register.
INX H ;Add 2nd data which is available
ADD M ;in memory to A. Sum in A-register.
JNC AHEAD ;If CF=0, go to AHEAD.
INR C ;If CF=1, increment C-register.
AHEAD: INX H
MOV M,A ;Save the sum in memory.
```

```

    INX H
    MOV M,C      ;Save the carry in memory.
    HLT          ;Halt program execution.

    END          ;Assembly end.

```

### Assembler listing for example program 1 (Method 2)

```

1          ;PROGRAM TO ADD TWO 8-BIT DATA
2          ;METHOD-2
3
4 4100          ORG 4100H ;specify program starting address.
5
6 4100 21 00 42  LXI H,4200H ;Set pointer for data.
7 4103 0E 00      MVI C,00H ;Clear C-register to account for carry.
8 4105 7E          MOV A,M ;Get 1st data in A-register.
9 4106 23          INX H ;Add 2nd data which is available
10 4107 86          ADD M ;in memory to A. Sum in A-register.
11 4108 D2 0C 41   JNC AHEAD ;If CF=0, go to AHEAD.
12 410B 0C          INR C ;If CF=1, increment C-register.
13 410C 23          AHEAD: INX H
14 410D 77          MOV M, A ;Save the sum in memory.
15 410E 23          INX H
16 410F 71          MOV M,C ;Save the carry in memory.
17 4110 76          HLT ;Halt program execution.
18
19 4111          END ;Assembly end.

```

### Sample data

Input Data :    Data-1 = E2<sub>H</sub>  
                  Data-2 = 45<sub>H</sub>

Output data :    Sum    = 27<sub>H</sub>  
                  Carry = 01<sub>H</sub>

Memory address	Content
4200	E2
4201	45
4202	27
4203	01

### EXAMPLE PROGRAM - 2: 16-Bit Addition

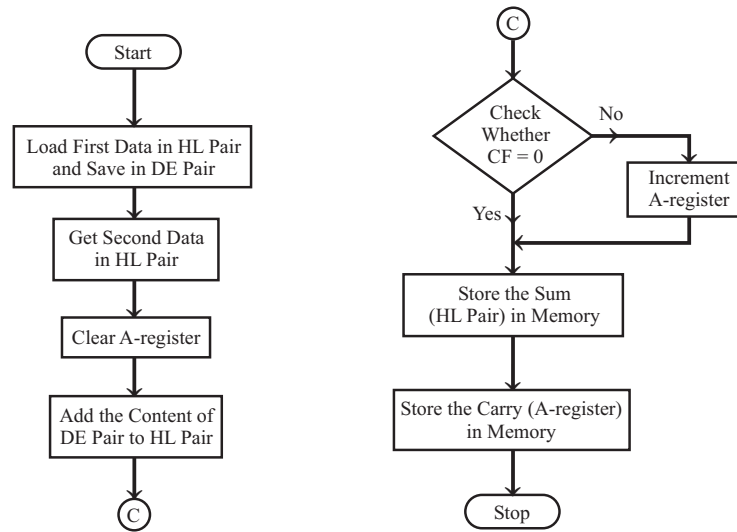
*Write an assembly language program to add two numbers of 16-bit data stored in memory locations from 4200<sub>H</sub> to 4203<sub>H</sub>. The data are stored such that the low byte first and then the high byte is stored. Store the result from 4204<sub>H</sub> to 4206<sub>H</sub>.*

### Problem Analysis

The 16-bit addition can be performed in 8085 either in terms of 8-bit addition or by using DAD instruction. In addition using DAD instruction, one of the data should be in HL pair and another data can be in another register pair. After addition the sum is stored in HL pair. If there is a carry in addition then that is indicated by setting a carry flag. Hence, one of the registers is used to account for carry.

### Algorithm

1. Load the first data in HL-register pair.
2. Move the first data to DE- register pair.
3. Load the second data in HL-register pair.
4. Clear A-register for carry.
5. Add the content of DE pair to HL pair.
6. Check for carry. If carry = 1, go to step 7 or If carry = 0, go to step 8.
7. Increment A-register to account for carry.
8. Store the sum and carry in memory.
9. Stop.

**Flowchart for example program 2****Assembly language program**

```

;PROGRAM TO ADD TWO 16-BIT DATA

ORG 4100H ;specify program starting address.

LHLD 4200H ;Get 1st data in HL pair.
XCHG      ;Save 1st data in DE pair.
LHLD 4202H ;Get 2nd data in HL pair.
XRA A     ;Clear A-register for carry.
DAD D     ;Get the sum in HL pair.
JNC AHEAD ;If CF=0, go to AHEAD.
INR A     ;If CF=1, increment A-register.
AHEAD: SHLD 4204H ;Store the sum in memory.
STA 4206H ;Store the carry in memory.
HLT       ;Halt program execution.

END        ;Assembly end.

```

**Assembler listing for example program 2**

```

1          ;PROGRAM TO ADD TWO 16-BIT DATA
2
3 4100          ORG 4100H ;specify program starting address.
4
5 4100 2A 00 42  LHLD 4200H ;Get 1st data in HL pair.
6 4103 EB       XCHG      ;Save 1st data in DE pair.
7 4104 2A 02 42  LHLD 4202H ;Get 2nd data in HL pair.
8 4107 AF       XRA A     ;Clear A-register for carry.
9 4108 19       DAD D     ;Get the sum in HL pair.
10 4109 D2 0D 41 JNC AHEAD ;If CF=0, go to AHEAD.
11 410C 3C       INR A     ;If CF=1, increment A-register.
12 410D 22 04 42 AHEAD: SHLD 4204H ;Store the sum in memory.
13 4110 32 06 42 STA 4206H ;Store the carry in memory.
14 4113 76       HLT       ;Halt program execution.
15
16 4114          END        ;Assembly end.

```

**Sample data**

Input Data : Data-1 = C254<sub>H</sub>  
                   Data-2 = 8A92<sub>H</sub>  
 Output Data : Sum = 4CE6<sub>H</sub>  
                   Carry = 01

Memory address	Content
4200	54
4201	C2
4202	92
4203	8A

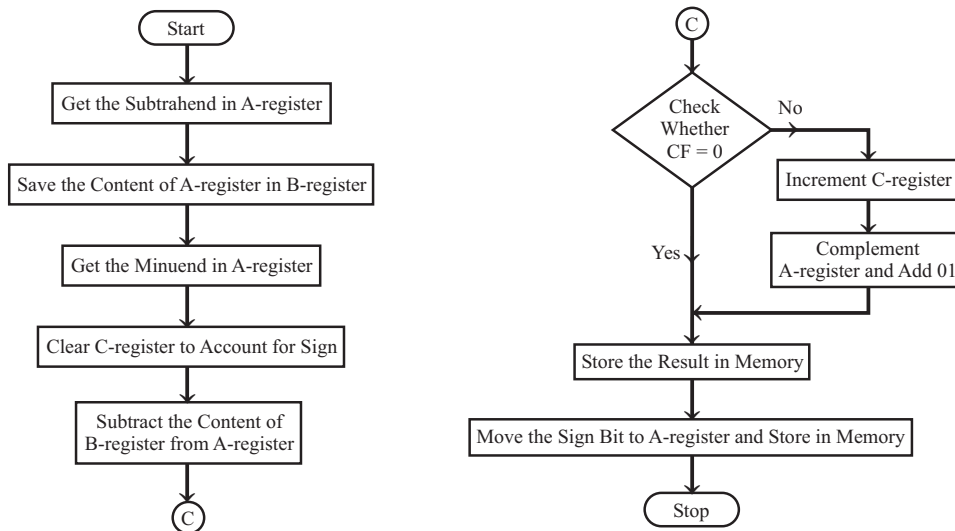
Memory address	Content
4204	E6
4205	4C
4206	01

**EXAMPLE PROGRAM - 3 : 8-Bit Subtraction**

Write an assembly language program to subtract two numbers of 8-bit data stored in memory locations 4200<sub>H</sub> and 4201<sub>H</sub>. Store the magnitude of the result in 4202<sub>H</sub>. If the result is positive store 00 in 4203<sub>H</sub> or if the result is negative store 01 in 4203<sub>H</sub>.

**Problem Analysis**

In order to perform subtraction in 8085, one of the data should be in accumulator and another data can be in any one of the general purpose register or in the memory. After subtraction the result is stored in the accumulator. The 8085 perform 2's complement subtraction and then complement the carry. Therefore, if the result is negative then the carry flag is set and the accumulator will have 2's complement of the result. One of the register is used to account for sign of the result. In order to get the magnitude of the result again take 2's complement of the result.

**Flowchart for example program 3****Algorithm**

1. Load the subtrahend (the data to be subtracted) from memory to accumulator and move it to B-register.
2. Load the minuend from memory to accumulator.
3. Clear C-register to account for sign of the result.
4. Subtract the content of B-register (subtrahend) from the content of accumulator (minuend).
5. Check for carry. If carry = 1, go to step 6 or if carry = 0, go to step 7.
6. Increment C-register. Complement the accumulator and add 01<sub>H</sub>.
7. Store the difference (accumulator) in memory.
8. Move the content of C-register (sign bit) to accumulator and store in memory.
9. Stop.



**Assembly language program**

```

;PROGRAM TO SUBTRACT TWO 8-BIT DATA

      ORG 4100H ;specify program starting address.

      LDA 4201H ;Get the subtrahend in B-register.
      MOV B,A
      LDA 4200H ;Get the minuend in A-register.
      MVI C,00H ;Clear C-register to account for sign.
      SUB B      ;Get the difference in A-register.
      JNC AHEAD  ;If CF=0, then go to AHEAD.
      INR C      ;If CF=1, then increment C-register.
      CMA        ;Get 2's complement of difference
      ADI 01H    ;(result) in A-register.
AHEAD: STA 4202H ;Store the result in memory.
      MOV A,C
      STA 4203H ;Store the sign bit in memory.
      HLT        ;Halt program execution.

      END        ;Assembly end.

```

**Assembler listing for example program 3**

```

1          ;PROGRAM TO SUBTRACT TWO 8-BIT DATA
2
3 4100          ORG 4100H ;specify program starting address.
4
5 4100 3A 01 42    LDA 4201H ;Get the subtrahend in B-register.
6 4103 47          MOV B,A
7 4104 3A 00 42    LDA 4200H ;Get the minuend in A-register.
8 4107 0E 00      MVI C,00H ;Clear C-register to account for sign.
9 4109 90          SUB B      ;Get the difference in A-register.
10 410A D2 11 41   JNC AHEAD  ;If CF=0, then go to AHEAD.
11 410D 0C          INR C      ;If CF=1, then increment C-register.
12 410E 2F          CMA        ;Get 2's complement of difference
13 410F C6 01      ADI 01H    ;(result) in A-register.
14 4111 32 02 42   AHEAD: STA 4202H ;Store the result in memory.
15 4114 79          MOV A,C
16 4115 32 03 42   STA 4203H ;Store the sign bit in memory.
17 4118 76          HLT        ;Halt program execution.
18
19 4119          END        ;Assembly end.

```

**Sample data**

```

Input Data : Minuend   = 5EH
              Subtrahend = 34H

Output Data : Difference = 2AH
              Sign bit   = 00H

```

Memory address	Content
4200	5E
4201	34
4202	2A
4203	00

**EXAMPLE PROGRAM - 4: 16-Bit Subtraction**

*Write an assembly language program to subtract two numbers of 16-bit data stored in memory locations from 4200<sub>H</sub> to 4203<sub>H</sub>. The data are stored such that the low byte is stored first and then the high byte is stored. Store the result in 4204<sub>H</sub> and 4205<sub>H</sub>.*

**Problem Analysis**

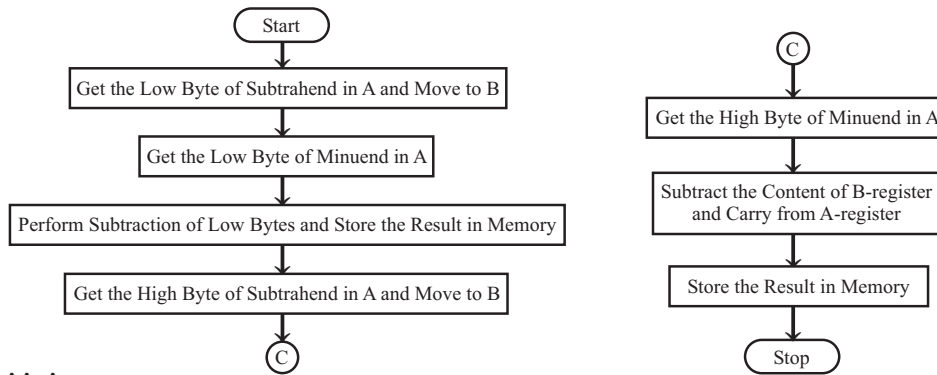
The 16-bit subtraction is performed in terms of 8-bit subtraction. First low bytes of the data are subtracted and the result is stored in memory. Then high bytes of the data are subtracted along with borrow (carry) in the previous subtraction and the result is stored in memory.

**Algorithm**

1. Load the low byte of subtrahend (the data to be subtracted) in accumulator from memory and move it to B-register.
2. Load the low byte of minuend in accumulator from memory.

3. Subtract the content of B-register (subtrahend) from the content of accumulator (minuend).
4. Store the low byte of result in memory.
5. Load the high byte of subtrahend in accumulator from memory and move it to B-register.
6. Load the high byte of minuend in accumulator from memory.
7. Subtract the content of B-register and the carry (borrow) from the content of accumulator.
8. Store the high byte of the result in memory.
9. Stop.

#### Flowchart for example program 4



#### Assembly language program

```

;PROGRAM TO SUBTRACT TWO 16-BIT DATA

ORG 4100H ;specify program starting address.

LDA 4202H
MOV B,A   ;Get low byte of subtrahend in B-register.
LDA 4200H ;Get low byte of minuend in A-register.
SUB B     ;Get difference of low bytes in A-register.
STA 4204H ;Store the result in memory.
LDA 4203H
MOV B,A   ;Get high byte of subtrahend in B-register.
LDA 4201H ;Get high byte of minuend in A-register.
SBB B     ;Get difference of high bytes in A-register.
STA 4205H ;Store the result.
HLT       ;Halt program execution.

END       ;Assembly end.

```

#### Assembler listing for example program 4

```

1          ;PROGRAM TO SUBTRACT TWO 16-BIT DATA
2
3 4100      ORG 4100H ;specify program starting address.
4
5 4100      3A 02 42 LDA 4202H
6 4103      47      MOV B,A   ;Get low byte of subtrahend in B-register.
7 4104      3A 00 42 LDA 4200H ;Get low byte of minuend in A-register.
8 4107      90      SUB B     ;Get difference of low bytes in A-register.
9 4108      32 04 42 STA 4204H ;Store the result in memory.
10 410B     3A 03 42 LDA 4203H
11 410E     47      MOV B,A   ;Get high byte of subtrahend in B-register.
12 410F     3A 01 42 LDA 4201H ;Get high byte of minuend in A-register.
13 4112     98      SBB B     ;Get difference of high bytes in A-register.
14 4113     32 05 42 STA 4205H ;Store the result.
15 4116     76      HLT       ;Halt program execution.
16
17 4117      END       ;Assembly end.

```

**Sample data**

Input Data : Minuend = B2AB<sub>H</sub>  
                   Subtrahend = 922C<sub>H</sub>  
 Output Data : Difference = 207F<sub>H</sub>

Memory address	Content
4200	AB
4201	B2
4202	2C
4203	92
4204	7F
4205	20

**EXAMPLE PROGRAM - 5 : 2-Digit BCD Addition**

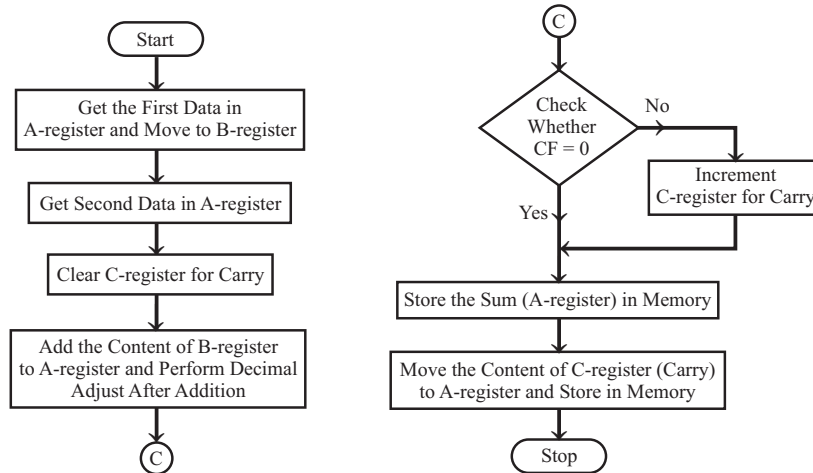
Write an assembly language program to add two numbers of 2-digit (8-bit) BCD data stored in memory locations 4200<sub>H</sub> and 4201<sub>H</sub>. Store the result in 4202<sub>H</sub> and 4203<sub>H</sub>.

**Problem Analysis**

The 8085 will perform only binary addition. Hence for BCD addition, the binary addition of BCD data is performed and then the sum is corrected to get the result in BCD. After binary addition the following correction should be made to get the result in BCD.

1. If the sum of lower nibbles exceeds 9 or if there is auxiliary carry then 6 is added to lower nibble.
2. If the sum of upper nibbles exceeds 9 or if there is carry then 6 is added to upper nibble.

The above correction is taken care by DAA (Decimal Adjust Accumulator) instruction. Therefore after binary addition, execute DAA instruction to do the above correction in the sum.

**Flowchart for example program 5****Algorithm**

1. Load the first data in accumulator and move it to B-register.
2. Load the second data in accumulator.
3. Clear C-register for storing carry.
4. Add the content of B-register to accumulator.
5. Execute DAA instruction.
6. Check for carry. If carry = 1, go to step 7 or if carry = 0, go to step 8.
7. Increment C-register to account for carry.
8. Store the sum (content of accumulator) in memory.
9. Move the carry (content of C-register) to accumulator and store in memory.
10. Stop.

**Assembly language program**

```

;PROGRAM TO ADD TWO 2-DIGIT BCD DATA

      ORG 4100H ;specify program starting address.

      LDA 4200H
      MOV B,A   ;Get 1st data in B-register.
      LDA 4201H ;Get 2nd data in A-register.
      MVI C,00H ;Clear C-register for accounting carry.
      ADD B
      DAA       ;Get the sum of BCD data in A-register.
      JNC AHEAD ;If CF=0, go to AHEAD.
      INR C     ;If CF=1, increment C-register.
AHEAD: STA 4202H ;Store the sum in memory.
      MOV A,C
      STA 4203H ;Store the carry in memory.
      HLT      ;Halt program execution.

      END      ;Assembly end.

```

**Assembler listing for example program 5**

```

1          ;PROGRAM TO ADD TWO 2-DIGIT BCD DATA
2
3 4100          ORG 4100H ;specify program starting address.
4
5 4100  3A 00 42      LDA 4200H
6 4103  47           MOV B,A   ;Get 1st data in B-register.
7 4104  3A 01 42      LDA 4201H ;Get 2nd data in A-register.
8 4107  0E 00         MVI C,00H ;Clear C-register for accounting carry.
9 4109  80           ADD B
10 410A  27          DAA       ;Get the sum of BCD data in A-register.
11 410B  D2 0F 41     JNC AHEAD ;If CF=0, go to AHEAD.
12 410E  0C          INR C     ;If CF=1, increment C-register.
13 410F  32 02 42     AHEAD: STA 4202H ;Store the sum in memory.
14 4112  79          MOV A,C
15 4113  32 03 42     STA 4203H ;Store the carry in memory.
16 4116  76          HLT      ;Halt program execution.
17
18 4117          END      ;Assembly end.

```

**Sample data**

Input Data : Data-1 = 72<sub>10</sub>  
               Data-2 = 99<sub>10</sub>  
 Output Data : Sum = 71<sub>10</sub>  
               Carry = 01<sub>10</sub>

Memory address	Content
4200	72
4201	99
4202	71
4203	01

**EXAMPLE PROGRAM - 6 : 4-Digit BCD Addition**

*Write an assembly language program to add two numbers of 4-digit BCD data stored in memory locations from 4200<sub>H</sub> to 4203<sub>H</sub> and store the result from 4204<sub>H</sub> to 4206<sub>H</sub>.*

**Problem Analysis**

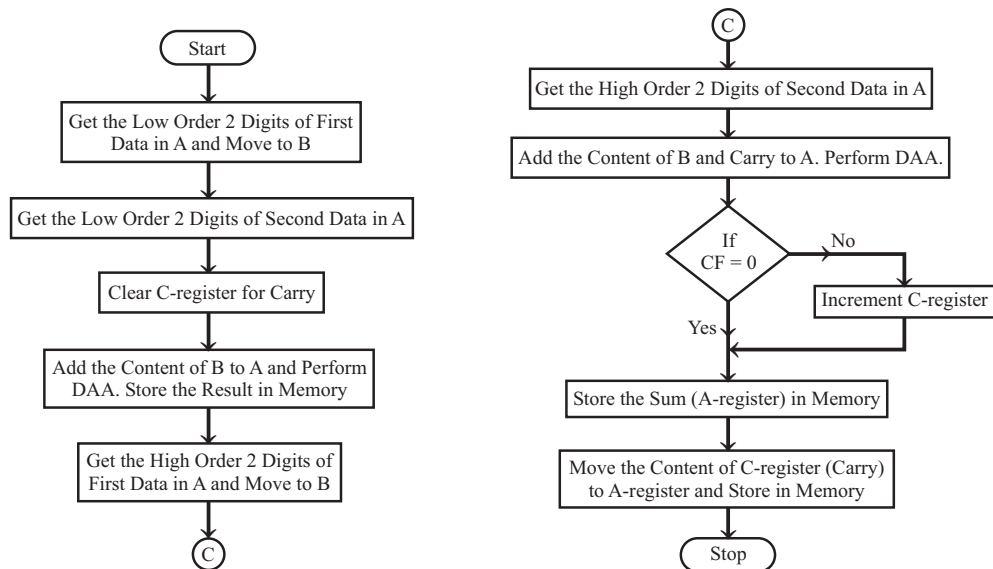
The 4-digit BCD addition is performed in terms of 2-digit BCD addition. First lower order two digits are added and the sum is stored in the memory. Then the higher order two digits are added along with previous carry and the sum and final carry are stored in the memory.

**Algorithm**

1. Load the low order two digits of first data in accumulator and move it to B-register.
2. Load the low order two digits of second data in accumulator.
3. Clear C-register for storing carry.
4. Add the content of B-register to accumulator.

5. Execute DAA instruction.
6. Store the low order two digits of the result in memory.
7. Load the high order two digits of first data in accumulator and move it to B-register.
8. Load the high order two digits of second data in accumulator.
9. Add the content of B-register and carry (from previous addition) to accumulator.
10. Execute DAA instruction.
11. Check for carry. If carry = 1, go to step 12 or if carry = 0, go to step 13.
12. Increment C-register to account for final carry.
13. Store the high order two digits of the result in memory.
14. Move the carry (content of C-register) to accumulator and store in memory.
15. Stop.

### Flowchart for example program 6



### Assembly language program

;PROGRAM TO ADD TWO 4-DIGIT BCD DATA

```

ORG 4100H ;specify program starting address.

LDA 4200H
MOV B,A ;Get low order 2 digits of 1st data in B.
LDA 4202H ;Get low order 2 digits of 2nd data in A.
MVI C,00H ;clear C-register to account for carry.
ADD B
DAA ;Get the sum of low order two digits in A
STA 4204H ;and store it in memory.
LDA 4201H
MOV B,A ;Get high order 2 digits of 1st data in B.
LDA 4203H ;Get high order 2 digits of 2nd data in A.
ADC B
DAA ;Get the sum of high order two digits in A
STA 4205H ;and store the same in memory.
JNC AHEAD
INR C ;If CF=1, increment C-register.

```

```

AHEAD:  MOV A,C
        STA 4206H ;Store the carry in memory.
        HLT      ;Halt program execution.

        END      ;Assembly end.

```

### Assembler listing for example program 6

```

1          ;PROGRAM TO ADD TWO 4-DIGIT BCD DATA
2
3  4100          ORG 4100H ;specify program starting address.
4
5  4100 3A 00 42  LDA 4200H
6  4103 47          MOV B,A ;Get low order 2 digits of 1st data in B.
7  4104 3A 02 42  LDA 4202H ;Get low order 2 digits of 2nd data in A.
8  4107 0E 00      MVI C,00H ;Clear C-register to account for carry.
9  4109 80          ADD B
10 410A 27          DAA ;Get the sum of low order 2 digits in A
11 410B 32 04 42  STA 4204H ;and store it in memory.
12 410E 3A 01 42  LDA 4201H
13 4111 47          MOV B,A ;Get high order 2 digits of 1st data in B.
14 4112 3A 03 42  LDA 4203H ;Get high order 2 digits of 2nd data in A.
15 4115 88          ADC B
16 4116 27          DAA ;Get the sum of high order 2 digits in A
17 4117 32 05 42  STA 4205H ;and store the same in memory.
18 411A D2 1E 41  JNC AHEAD
19 411D 0C          INR C ;If CF=1, increment C-register.
20 411E 79          AHEAD: MOV A,C
21 411F 32 06 42  STA 4206H ;Store the carry in memory.
22 4122 76          HLT ;Halt program execution.
23
24 4123          END ;Assembly end.

```

### Sample data

Input Data : Data-1 = 8067<sub>10</sub>  
                   Data-2 = 2892<sub>10</sub>  
 Output Data : Sum = 0959<sub>10</sub>  
                   Carry = 01<sub>10</sub>

Memory address	Content
4200	67
4201	80
4202	92
4203	28

Memory address	Content
4204	59
4205	09
4206	01

### EXAMPLE PROGRAM - 7: BCD Subtraction

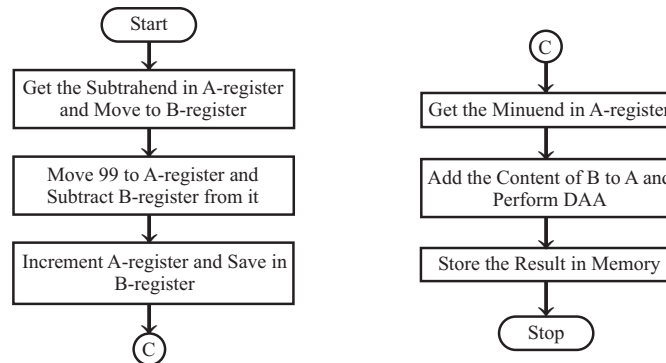
Write an assembly language program to subtract two numbers of 2-digit BCD data stored in memory locations 4200<sub>H</sub> and 4201<sub>H</sub> and store the result in 4202<sub>H</sub>.

### Problem Analysis

The 8085 will perform only binary subtraction. Hence for BCD subtraction, 10's complement subtraction is performed. First the 10's complement of the subtrahend is obtained and then added to the minuend. The DAA instruction is executed to get the result in BCD.

### Algorithm

1. Load the subtrahend in accumulator and move it to B-register.
2. Move 99 to accumulator and subtract the content of B-register from accumulator.
3. Increment the accumulator.
4. Move the content of accumulator to B-register.
5. Load the minuend in accumulator.
6. Add the content of B-register to accumulator.
7. Execute DAA instruction.
8. Store the result in memory.
9. Stop.

**Flowchart for example program 7****Assembly language program**

```

;PROGRAM TO SUBTRACT TWO BCD (2-DIGIT) DATA
    ORG 4100H ;specify program starting address.
    LDA 4201H
    MOV B,A   ;Get the subtrahend in B-register.
    MVI A,99H ;Get 10's complement of
    SUB B     ;subtrahend in A.
    INR A
    MOV B,A   ;Save 10's complement in B.
    LDA 4200H ;Get the minuend in A-register.
    ADD B     ;Get BCD sum of minuend and 10's
    DAA       ;complement of subtrahend. This sum
              ;is the difference between BCD data.
    STA 4202H ;Store the result in memory.
    HLT      ;Halt program execution.
    END      ;Assembly end.
  
```

**Assembler listing for example program 7**

```

1          ;PROGRAM TO SUBTRACT TWO BCD (2-DIGIT) DATA
2
3 4100      ORG 4100H ;specify program starting address.
4
5 4100 3A 01 42 LDA 4201H
6 4103 47      MOV B,A   ;Get the subtrahend in B-register.
7 4104 3E 99   MVI A, 99H ;Get 10's complement of
8 4106 90      SUB B     ;subtrahend in A.
9 4107 3C      INR A
10 4108 47     MOV B,A   ;Save 10's complement in B.
11 4109 3A 00 42 LDA 4200H ;Get the minuend in A-register.
12 410C 80     ADD B     ;Get BCD sum of minuend and 10's
13 410D 27     DAA       ;complement of subtrahend. This sum
14              ;is the difference between BCD data.
15 410E 32 02 42 STA 4202H ;Store the result in memory.
16 4111 76     HLT      ;Halt program execution.
17
18 4112      END      ;Assembly end.
  
```

**Sample data**

Input Data : Minuend = 95<sub>10</sub>  
                   Subtrahend = 32<sub>10</sub>

Output Data : Difference = 63<sub>10</sub>

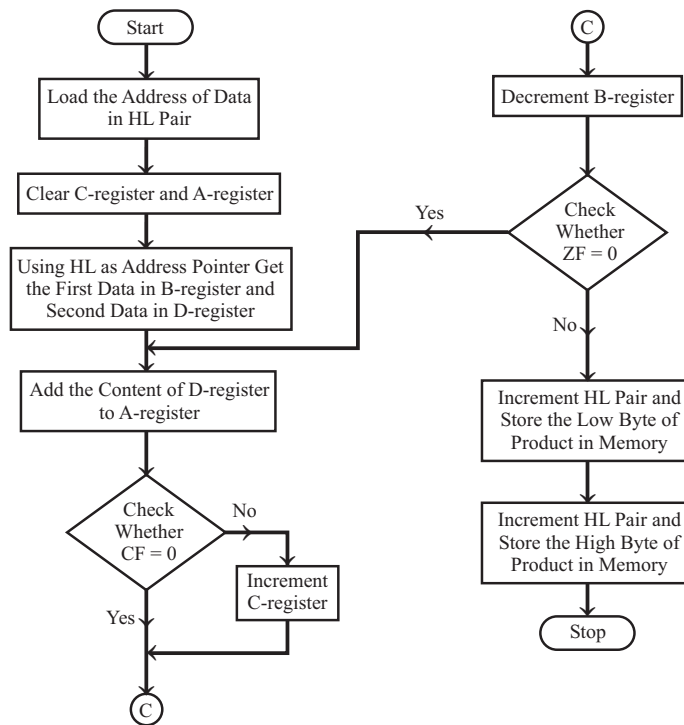
Memory address	Content
4200	95
4201	32
4202	63

**EXAMPLE PROGRAM - 8 : 8-Bit Multiplication**

Write an assembly language program to multiply two numbers of 8-bit data stored in memory locations 4200<sub>H</sub> and 4201<sub>H</sub> and store the product in 4202<sub>H</sub> and 4203<sub>H</sub>.

**Problem Analysis**

In 8085, the multiplication is performed as repeated additions. The initial value of sum is assumed as zero. One of the data is used as count (N) for the number of additions to be performed. Another data is added to the sum N times, where N is the count. The result of the product of two 8-bit data will be 16 bits. Hence, another register is used to account for the overflow.

**Flowchart for example program 8****Algorithm**

1. Load the address of the first data in HL pair (pointer).
2. Clear C-register for overflow (carry).
3. Clear the accumulator.
4. Move the first data to B-register (count).
5. Increment the pointer.
6. Move the second data to D-register (multiplicand).
7. Add the content of D-register to accumulator.
8. Check for carry. If carry = 1, go to step 9 or If carry = 0, go to step 10.
9. Increment C-register.
10. Decrement B-register (count).
11. Check whether count has reached zero. If ZF = 0 repeat steps 7 through 11, or if ZF = 1 go to next step.



12. Increment the pointer and store low byte of the product in memory.
13. Increment the pointer and store high byte of the product in memory.
14. Stop.

### Assembly language program

```
;PROGRAM TO MULTIPLY TWO NUMBERS OF 8-BIT DATA

      ORG 4100H      ;specify program starting address.

      LXI H,4200H    ;Set pointer for data.
      MVI C,00H      ;Clear C to account for overflow (Carry).
      XRA A          ;Clear accumulator(Initial sum = 0).
      MOV B,M        ;Get 1st data in B-register.
      INX H
      MOV D,M        ;Get 2nd data in D-register.
REPT: ADD D          ;Add D-register to accumulator.
      JNC AHEAD      ;If CF=1, increment C-register.
      INR C
AHEAD: DCR B
      JNZ REPT       ;Repeat addition until ZF=1.
      INX H
      MOV M,A        ;Store low byte of product in memory.
      INX H
      MOV M,C        ;Store high byte of product in memory.
      HLT            ;Halt program execution.

      END            ;Assembly end.
```

### Assembler listing for example program 8

```
1          ;PROGRAM TO MULTIPLY TWO NUMBERS OF 8-BIT DATA
2
3 4100          ORG 4100H      ;specify program starting address.
4
5 4100 21 00 42    LXI H,4200H ;Set pointer for data.
6 4103 0E 00      MVI C,00H   ;Clear C to account for overflow (Carry).
7 4105 AF         XRA A       ;Clear accumulator(Initial sum = 0).
8 4106 46         MOV B,M     ;Get 1st data in B-register.
9 4107 23         INX H
10 4108 56        MOV D,M     ;Get 2nd data in D-register.
11 4109 82        REPT: ADD D  ;Add D-register to accumulator.
12 410A D2 0E 41  JNC AHEAD
13 410D 0C        INR C       ;If CF=1, increment C-register.
14 410E 05        AHEAD: DCR B
15 410F C2 09 41  JNZ REPT    ;Repeat addition until ZF=1.
16 4112 23        INX H
17 4113 77        MOV M,A     ;Store low byte of product in memory.
18 4114 23        INX H
19 4115 71        MOV M,C     ;Store high byte of product in memory.
20 4116 76        HLT        ;Halt program execution.
21
22 4117          END          ;Assembly end.
```

### Sample data

Input Data : Data-1 = C7<sub>H</sub>  
                       Data-2 = 4A<sub>H</sub>

Output Data : Product = 3986<sub>H</sub>

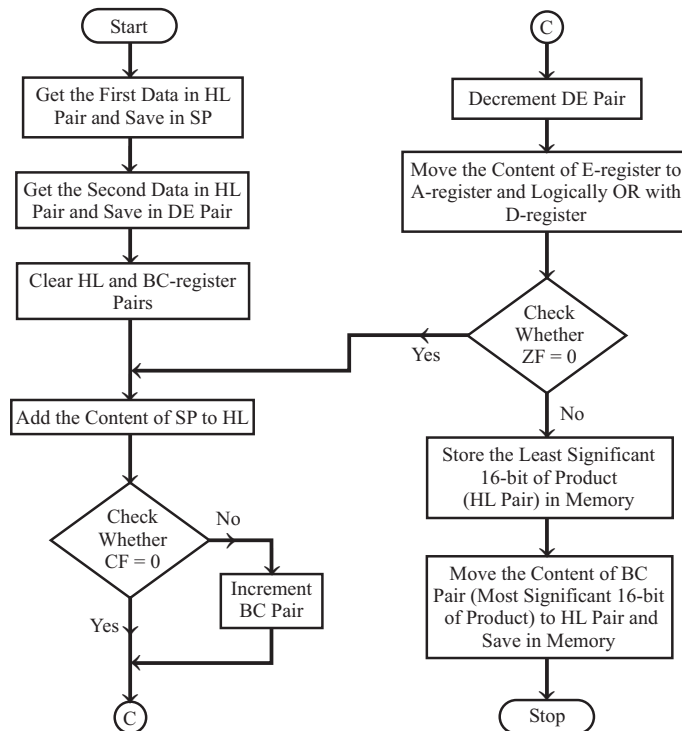
Memory address	Content
4200	C7
4201	4A
4202	86
4203	39

### EXAMPLE PROGRAM - 9: 16-Bit Multiplication

Write an assembly language program to multiply two numbers of 16-bit data stored in memory locations from 4200<sub>H</sub> to 4203<sub>H</sub>. Store the product in memory locations from 4204<sub>H</sub> to 4207<sub>H</sub>.

**Problem Analysis**

The 16-bit multiplication is performed as repeated 16-bit additions. The initial sum is assumed as zero. One of the data is stored in SP (Stack Pointer) and another data is stored in DE pair. The content of DE pair is used as count for number of additions. The content of SP is added to the sum N times, where N is the count. The maximum size of product will be 32-bit. Hence BC pair is used to account for overflow. In 16-bit decrement no flags are affected. Hence, to check zero of the count (DE pair), move E-register to A-register and logically ORed with D-register.

**Flowchart for example program 9****Algorithm**

1. Load the first data in HL pair and move to SP.
2. Load the second data in HL and move to DE (count).
3. Clear HL pair (Initial sum).
4. Clear BC pair for overflow (carry).
5. Add the content of SP to HL.
6. Check for carry. If carry=1, go to step 7 or If carry=0, go to step 8.
7. Increment BC pair.
8. Decrement the count.
9. Check whether count has reached zero.
10. To check for zero of the count, move the content of E-register to A-register and logically OR with D-register.
11. Check the zero flag. If ZF=0, repeat steps 5 through 11 or If ZF=1, go to next step.

12. Store the content of HL in memory. (Least significant 16 bits of the product).
13. Move the content of C to L and B to H and store HL in memory. (Most significant 16 bits of the product).
14. Stop.

### Assembly language program

```
;PROGRAM TO MULTIPLY TWO NUMBERS OF 16-BIT DATA

      ORG 4100H      ;specify program starting address.

      LHLD 4200H     ;Get 1st data in HL pair.
      SPHL          ;Save 1st data in SP.
      LHLD 4202H     ;Get 2nd data in HL pair.
      XCHG          ;Save 2nd data in DE pair.
      LXI H,0000H    ;Clear HL pair(initial sum=0).
      LXI B,0000H    ;Clear BC pair to account overflow.
NEXT:  DAD SP        ;Add the content of SP to sum(HL).
      JNC AHEAD
      INX B          ;If CF=1, increment BC pair.
AHEAD: DCX D
      MOV A,E        ;Check for zero in DE pair. This is done
      ORA D          ;by logically ORing D and E.
      JNZ NEXT       ;Repeat addition until count is zero.
      SHLD 4204H     ;Store lower 16-bit of product in memory.
      MOV L,C
      MOV H,B
      SHLD 4206H     ;Store upper 16-bit of product in memory.
      HLT            ;Halt program execution.

      END            ;Assembly end.
```

### Assembler listing for example program 9

```
1          ;PROGRAM TO MULTIPLY TWO NUMBERS OF 16-BIT DATA
2
3 4100          ORG 4100H      ;specify program starting address.
4
5 4100 2A 00 42    LHLD 4200H     ;Get 1st data in HL pair.
6 4103 F9          SPHL          ;Save 1st data in SP.
7 4104 2A 02 42    LHLD 4202H     ;Get 2nd data in HL pair.
8 4107 EB          XCHG          ;Save 2nd data in DE pair.
9 4108 21 00 00    LXI H,0000H    ;Clear HL pair(initial sum=0).
10 410B 01 00 00   LXI B,0000H    ;Clear BC pair to account overflow.
11 410E 39          NEXT: DAD SP   ;Add the content of SP to sum(HL).
12 410F D2 13 41   JNC AHEAD
13 4112 03          INX B          ;If CF=1, increment BC pair.
14 4113 1B          AHEAD: DCX D
15 4114 7B          MOV A,E        ;Check for zero in DE pair. This is done
16 4115 B2          ORA D          ;by logically ORing D and E.
17 4116 C2 0E 41   JNZ NEXT       ;Repeat addition until count is zero.
18 4119 22 04 42   SHLD 4204H     ;Store lower 16-bit of product in memory.
19 411C 69          MOV L,C
20 411D 60          MOV H,B
21 411E 22 06 42   SHLD 4206H     ;Store upper 16-bit of product in memory.
22 4121 76          HLT            ;Halt program execution.
23
24 4122          END            ;Assembly end.
```

### Sample data

Input Data : Data-1 = 5A 24<sub>H</sub>  
               Data-2 = 47C2<sub>H</sub>  
 Output Data : Product = 19444B48<sub>H</sub>

Memory address	Content	Memory address	Content
4200	24	4204	48
4201	5A	4205	4B
4202	C2	4206	44
4203	47	4207	19

**EXAMPLE PROGRAM - 10 : 8-Bit Division**

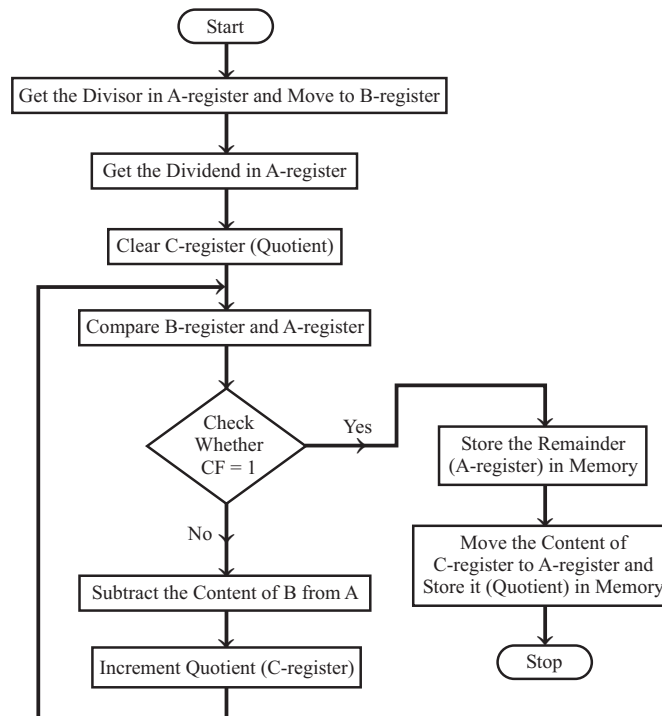
Write an assembly language program to divide two numbers of 8-bit data stored in memory locations 4200<sub>H</sub> and 4201<sub>H</sub>. Store the quotient in 4202<sub>H</sub> and the remainder in 4203<sub>H</sub>.

**Problem Analysis**

The division in 8085 is performed as repeated subtraction. The dividend is stored in A-register and divisor in B-register. The initial value of quotient is assumed as zero. Subtraction should be performed only when dividend is greater than divisor. So repeated subtraction is performed until dividend is lesser than the divisor. For each subtraction, the quotient is incremented by one. Then store the quotient and remainder in the memory.

**Algorithm**

1. Load the divisor in accumulator and move it to B-register.
2. Load the dividend in accumulator.
3. Clear C-register to account for quotient.
4. Check whether divisor is less than dividend. If divisor is less than dividend, go to step 8, otherwise go to next step.
5. Subtract the content of B-register from accumulator.
6. Increment the content of C-register (quotient).
7. Go to step 4.
8. Store the content of accumulator (remainder) in memory.
9. Move the content of C-register (quotient) to accumulator and store in memory.
10. Stop.

**Flowchart for example program 10**

**Assembly language program**

```

;PROGRAM TO DIVIDE TWO NUMBERS OF 8-BIT DATA

        ORG 4100H ;specify program starting address.

        LDA 4201H
        MOV B,A   ;Get the divisor in B-register.
        LDA 4200H ;Get the dividend in A-register.
        MVI C,00H ;Clear C-register for quotient.
AGAIN:   CMP B
        JC STORE ;If divisor is less than dividend go to store.
        SUB B    ;Subtract divisor from dividend.
        INR C    ;Increment quotient by one for each subtraction.
        JMP AGAIN
STORE:   STA 4203H ;Store the remainder in memory.
        MOV A,C
        STA 4202H ;Store the quotient in memory.
        HLT      ;Halt program execution.

        END      ;Assembly end.

```

**Assembler listing for example program 10**

```

1          ;PROGRAM TO DIVIDE TWO NUMBERS OF 8-BIT DATA
2
3  4100          ORG 4100H ;specify program starting address.
4
5  4100  3A 01 42    LDA 4201H
6  4103  47          MOV B,A   ;Get the divisor in B-register.
7  4104  3A 00 42    LDA 4200H ;Get the dividend in A-register.
8  4107  0E 00      MVI C,00H ;Clear C-register for quotient.
9  4109  B8          AGAIN: CMP B
10 410A  DA 12 41    JC STORE ;If divisor is less than dividend go to store.
11 410D  90          SUB B    ;Subtract divisor from dividend.
12 410E  0C          INR C    ;Increment quotient by one for each subtraction.
13 410F  C3 09 41    JMP AGAIN
14 4112  32 03 42    STORE: STA 4203H ;Store the remainder in memory.
15 4115  79          MOV A,C
16 4116  32 02 42    STA 4202H ;Store the quotient in memory.
17 4119  76          HLT      ;Halt program execution.
18
19 411A          END      ;Assembly end.

```

**Sample data**

Input Data : Dividend =  $C9_H$   
               Divisor =  $0A_H$   
 Output Data : Quotient =  $14_H$   
               Remainder =  $01_H$

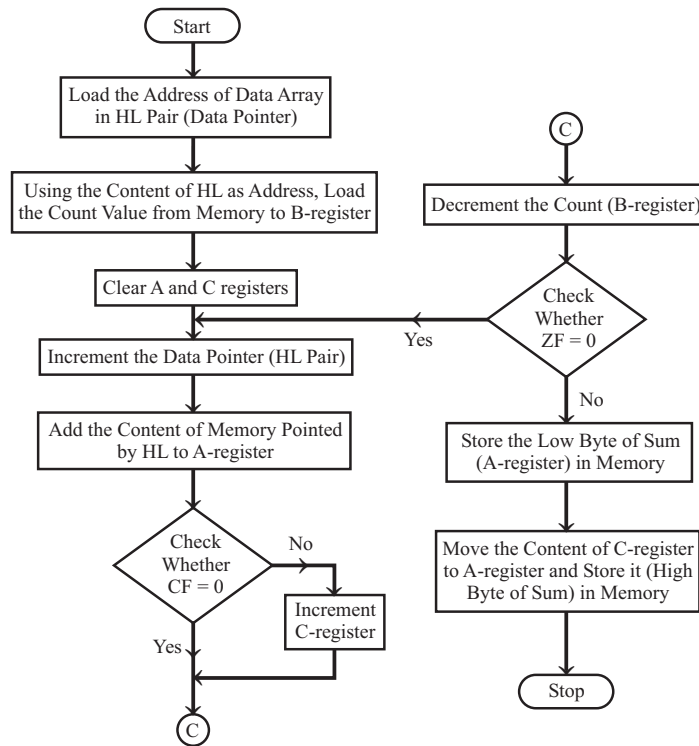
Memory address	Content
4200	$C9$
4201	$0A$
4202	$14$
4203	$01$

**EXAMPLE PROGRAM - 11: Sum of an Array**

*Write an assembly language program to add an array of data stored in memory from  $4200_H$  to  $4200_H + N$ . The first element of the array, gives the number of elements in the array. Store the result in  $4300_H$  and  $4301_H$ . Assume that the sum does not exceed 16-bit.*

**Problem Analysis**

The number of bytes (data)  $N$ , is used as count for number of additions. The initial sum is assumed as zero. The HL register pair is used as pointer for data. Each element of the array is added to sum and for accounting the overflow one of the registers is used.

**Flowchart for example program 11****Algorithm**

1. Load the address of the first element of the array in HL pair (pointer).
2. Move the count to B-register.
3. Clear C-register for carry.
4. Clear accumulator for sum.
5. Increment the pointer (HL pair).
6. Add the content of memory addressed by HL to accumulator.
7. Check for carry. If carry = 1, go to step 8, or If carry = 0, go to step 8.
8. Increment C-register.
9. Decrement the count.
10. Check for zero of the count. If ZF = 0 go to step 5 or If ZF = 1, go to next step.
11. Store the content of accumulator (low byte of sum).
12. Move the content of C-register (high byte of sum) to accumulator. Store the content of accumulator in memory.
13. Stop.

**Assembly language program**

```
; PROGRAM TO ADD AN ARRAY OF DATA
```

```

ORG 4100H    ;specify program starting address.

LXI H,4200H  ;Set pointer for data.
MOV B,M      ;Set count for number of data.
MVI C,00H    ;Clear C-register to account for carry.
XRA A        ;Clear accumulator. Initial sum=0.

```

```

REPT:   INX H
        ADD M      ;Add an element of the array to sum.
        JNC AHEAD
        INR C      ;If CF=1, increment C-register.
AHEAD:  DCR B
        JNZ REPT   ;Repeat addition until count is zero.
        STA 4300H  ;Store low byte of sum in memory.
        MOV A,C
        STA 4301H  ;Store high byte of sum in memory.
        HLT       ;Halt program execution.

        END       ;Assembly end.

```

**Assembler listing for example program 11**

```

1          ;PROGRAM TO ADD AN ARRAY OF DATA
2
3 4100          ORG 4100H ;specify program starting address.
4
5 4100 21 00 42  LXI H,4200H ;Set pointer for data.
6 4103 46        MOV B,M      ;Set count for number of data.
7 4104 0E 00     MVI C,00H    ;Clear C-register to account for carry.
8 4106 AF        XRA A        ;Clear accumulator. Initial sum=0.
9 4107 23        REPT: INX H
10 4108 86       ADD M      ;Add an element of the array to sum.
11 4109 D2 0D 41 JNC AHEAD
12 410C 0C       INR C      ;If CF=1, increment C-register.
13 410D 05       AHEAD: DCR B
14 410E C2 07 41 JNZ REPT   ;Repeat addition until count is zero.
15 4111 32 00 43 STA 4300H  ;Store low byte of sum in memory.
16 4114 79       MOV A,C
17 4115 32 01 43 STA 4301H  ;Store high byte of sum in memory.
18 4118 76       HLT       ;Halt program execution.
19
20 4119 END      ;Assembly end.

```

**Sample data**

Input Data : Count = 07<sub>H</sub>  
 Array = C2<sub>H</sub>  
           45<sub>H</sub>  
           B3<sub>H</sub>  
           F4<sub>H</sub>  
           7C<sub>H</sub>  
           ED<sub>H</sub>  
           16<sub>H</sub>

Output Data : Sum = 042D<sub>H</sub>

Memory address	Content	
4200	07	Count
4201	C2	
4202	45	Array
4203	B3	
4204	F4	
4205	7C	
4206	ED	
4207	16	
4300	2D	Sum
4301	04	

**EXAMPLE PROGRAM - 12 : Search for Smallest Data in an Array**

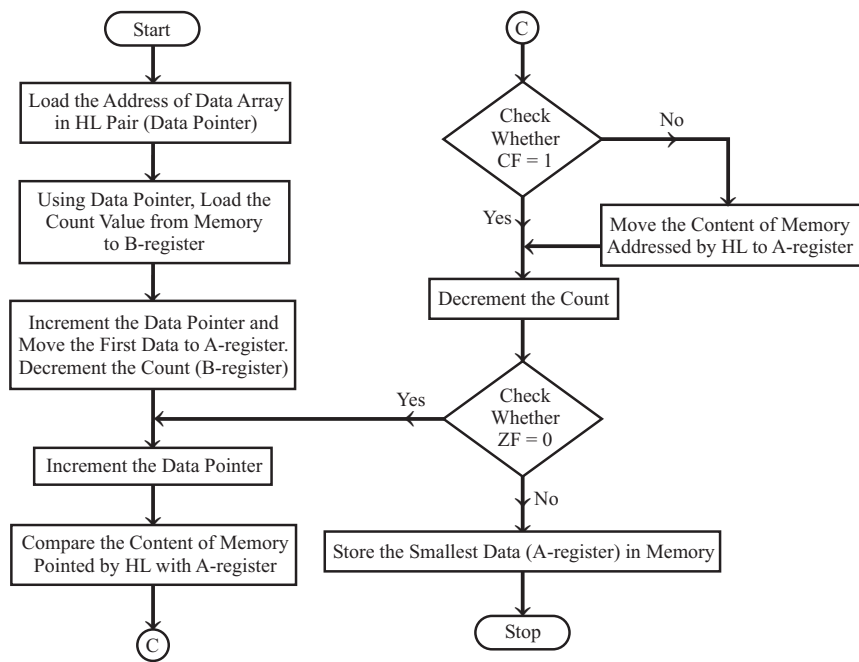
*Write an assembly language program to search the smallest data in an array of N data stored in memory locations from 4200<sub>H</sub> to (4200<sub>H</sub> + N). The first element of the array gives the number of data in the array. Store the smallest data in 4300<sub>H</sub>*

**Problem Analysis**

The HL register pair is used as pointer for the array. One of the general purpose register is used as count. A data in the array is moved to A-register and compared with next data. After each comparison, the smallest data is brought to accumulator. The comparisons are carried N-1 times. After N-1 comparisons, the smallest data will be in A-register and store it in memory.

**Algorithm**

1. Load the address of the first element of the array in HL register pair (pointer).
2. Move the count to B-register.
3. Increment the pointer.
4. Get the first data in accumulator.
5. Decrement the count.
6. Increment the pointer.
7. Compare the content of memory addressed by HL pair with that of accumulator.
8. If carry = 1, go to step 10 or If carry = 0, go to step 8.
9. Move the content of memory addressed HL to accumulator.
10. Decrement the count.
11. Check for zero of the count. If ZF = 0, go to step 6, or If ZF = 1 go to next step.
12. Store the smallest data in memory.
13. Stop.

**Flowchart for example program 12****Assembly language program**

;PROGRAM TO SEARCH SMALLEST DATA IN AN ARRAY

```

ORG 4100H    ;specify program starting address.

LXI H,4200H  ;Set pointer for array.
MOV B,M      ;Set count for number of elements in array.
INX H
MOV A,M      ;Set 1st element of array as smallest data.
DCR B        ;Decrement the count.

```



```

LOOP: INX H      ;Compare an element of array
      CMP M      ;with current smallest data.
      JC AHEAD   ;If CF=1, go to AHEAD.
      MOV A,M    ;If CF=0, then content of memory
                ;is smaller than A, hence if CF=0, make
                ;memory as smallest by moving to A.
AHEAD: DCR B
      JNZ LOOP   ;Repeat comparison until count is zero.
      STA 4300H  ;Store the smallest data in memory.
      HLT       ;Halt program execution.

      END       ;Assembly end.

```

### Assembler listing for example program 12

```

1          ;PROGRAM TO SEARCH SMALLEST DATA IN AN ARRAY
2
3 4100          ORG 4100H ;specify program starting address.
4
5 4100 21 00 42  LXI H,4200H ;Set pointer for array.
6 4103 46        MOV B,M    ;Set count for number of elements in array.
7 4104 23        INX H
8 4105 7E        MOV A,M    ;Set 1st element of array as smallest data.
9 4106 05        DCR B      ;Decrement the count.
10 4107 23      LOOP: INX H  ;Compare an element of array
11 4108 BE      CMP M      ;with current smallest data.
12 4109 DA 0D 41 JC AHEAD   ;If CF=1, go to AHEAD.
13 410C 7E      MOV A,M    ;If CF=0, then content of memory
14                ;is smaller than A, hence if CF=0, make
15                ;memory as smallest by moving to A.
16 410D 05      AHEAD: DCR B ;Decrement the count.
17 410E C2 07 41 JNZ LOOP   ;Repeat comparison until count is zero.
18 4111 32 00 43 STA 4300H  ;Store the smallest data in memory.
19 4114 76      HLT       ;Halt program execution.
20
21 4115          END       ;Assembly end.

```

### Sample data

```

Input Data : Count = 07H
            Array  = 42H
                3AH
                1CH
                24H
                B4H
                25H
                4FH
Output Data : Smallest data } = 1CH

```

Memory address	Content	
4200	07	Count
4201	42	
4202	3A	Array
4203	1C	
4204	24	
4205	B4	
4206	25	
4207	4F	
4300	1C	Smallest data

### EXAMPLE PROGRAM - 13 : Search for Largest Data in an Array

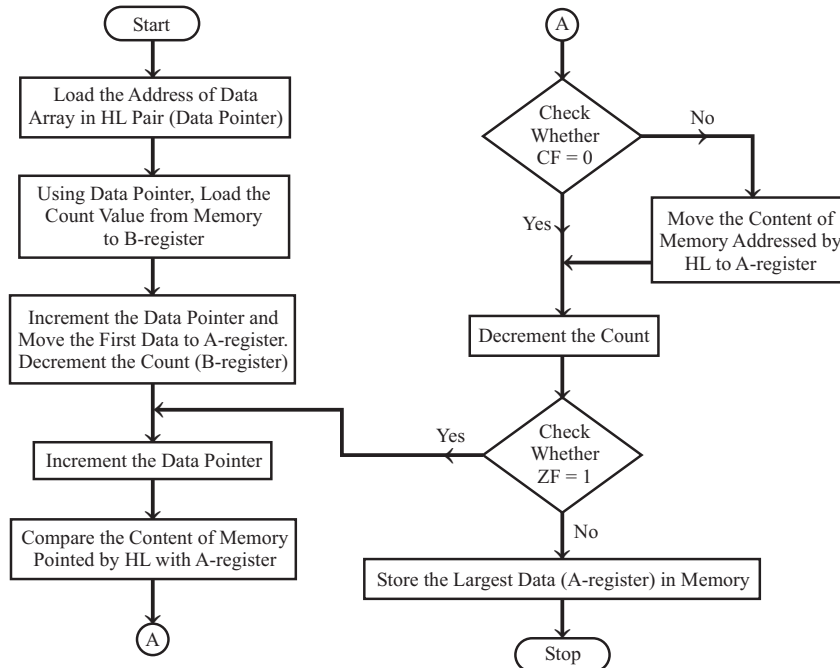
Write an assembly language program to search the largest data in an array of  $N$  data stored in memory locations from  $4200_H$  to  $4200_H + N$ . The first element of the array is the number of data ( $N$ ) in the array. Store the largest data in  $4300_H$ .

### Problem Analysis

The HL register pair is used as pointer for the array. One of the general purpose register is used as count. A data in the array is moved to A-register and compared with next data. After each comparison, the largest data is brought to A-register. The comparisons are performed  $N-1$  times. After  $N-1$  comparisons the largest data will be in A-register and store it in memory.

**Algorithm**

1. Load the address of the first element of the array in HL register pair (pointer).
2. Move the count to B-register.
3. Increment the pointer.
4. Get the first data in accumulator.
5. Decrement the count.
6. Increment the pointer.
7. Compare the content of memory addressed by HL pair with that of accumulator.
8. If carry = 0, go to step 10 or If carry = 1, go to step 8.
9. Move the content of memory addressed HL to accumulator.
10. Decrement the count.
11. Check for zero of the count. If ZF = 0, go to step 6, or If ZF = 1 go to next step.
12. Store the largest data in memory.
13. Stop.

**Flowchart for example program 13****Assembly language program**

```

;PROGRAM TO SEARCH LARGEST DATA IN AN ARRAY

ORG 4100H ;specify program starting address.

LXI H,4200H ;Set pointer for array.
MOV B,M ;Set count for number of elements in array.
INX H
MOV A,M ;Set 1st element of array as largest data.
DCR B ;Decrement the count.
LOOP: INX H ;Compare an element of array with
CMP M ;current largest data.
JNC AHEAD ;If CF=0, go to AHEAD.

```

```

        MOV A,M      ;If CF=1,then content of memory is larger
                    ;than accumulator.Hence if CF=1,
                    ;make memory content as current
                    ;largest by moving it to A-register.
AHEAD: DCR B
        JNZ LOOP     ;Repeat comparison until count is zero.
        STA 4300H    ;Store the largest data in memory.
        HLT          ;Halt program execution.

        END          ;Assembly end.

```

### Assembler listing for example program 13

```

1          ;PROGRAM TO SEARCH LARGEST DATA IN AN ARRAY
2
3 4100          ORG 4100H ;specify program starting address.
4
5 4100 21 00 42    LXI H,4200H ;Set pointer for array.
6 4103 46          MOV B,M      ;Set count for number of elements in array.
7 4104 23          INX H
8 4105 7E          MOV A,M      ;Set 1st element of array as largest data.
9 4106 05          DCR B        ;Decrement the count.
10 4107 23         LOOP: INX H   ;Compare an element of array with
11 4108 BE          CMP M       ;current largest data.
12 4109 D2 0D 41    JNC AHEAD   ;If CF=0, go to AHEAD.
13 410C 7E          MOV A,M     ;If CF=1,then content of memory is larger
14                    ;than accumulator. Hence if CF=1,
15                    ;make memory content as current
16                    ;largest by moving it to A-register.
17 410D 05         AHEAD: DCR B
18 410E C2 07 41    JNZ LOOP     ;Repeat comparison until count is zero.
19 4111 32 00 43    STA 4300H    ;Store the largest data in memory.
20 4114 76          HLT         ;Halt program execution.
21
22 4115          END          ;Assembly end.

```

### Sample data

Input Data : Count = 07<sub>H</sub>  
               Array = 62<sub>H</sub>  
                       7D<sub>H</sub>  
                       FC<sub>H</sub>  
                       24<sub>H</sub>  
                       C2<sub>H</sub>  
                       0F<sub>H</sub>  
                       92<sub>H</sub>

Output Data : Largest data } = FC<sub>H</sub>

Memory address	Content	
4200	07	Count Array
4201	62	
4202	7D	
4203	FC	
4204	24	
4205	C2	
4206	0F	
4207	92	
4300	FC	Laragest data

### EXAMPLE PROGRAM - 14 : Search for a Given Data in an Array

Write an assembly language program to search for a given data (stored in 4250<sub>H</sub>) in an array of data stored from 4200<sub>H</sub>. The end of the array is marked by 20<sub>H</sub>.

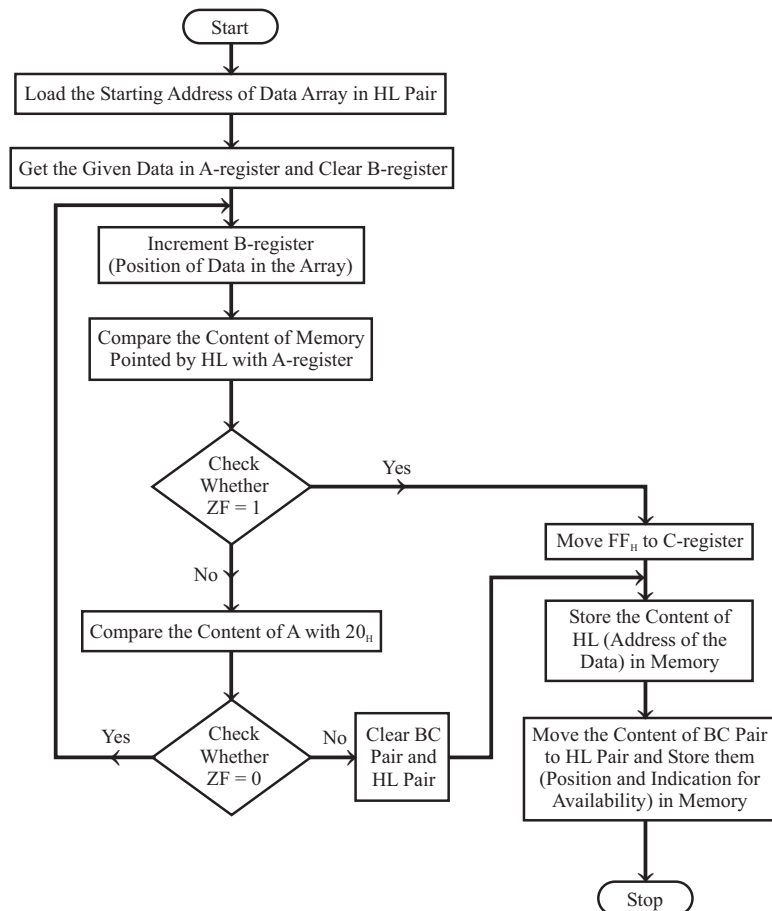
If the data is available store FF<sub>H</sub> in 4251<sub>H</sub>. Store the position of the data and its address in 4252<sub>H</sub>, 4253<sub>H</sub> and 4254<sub>H</sub> respectively. If the data is not available store 00<sub>H</sub> in memory locations from 4251<sub>H</sub> to 4254<sub>H</sub>.

### Problem Analysis

The HL pair is used as pointer for given data. B-register is used as pointer for position of the data. C-register is used to record the availability of given data. The given data is moved to A-register and compared with each element of the array one-by-one. If the data is available, terminate the comparison and store the position, address and FF<sub>H</sub> (for availability) in memory.

**Algorithm**

1. Load the address of the data array in HL register pair.
2. Load the given data in accumulator.
3. Clear B-register.
4. Increment B-register.
5. Compare the content of memory addressed by HL pair with that of accumulator.
6. If  $ZF = 0$ , go to next step or if  $ZF = 1$ , go to step 8.
7. Check for end of array by comparing the data with  $20_H$ . If  $ZF = 0$ , go to step 4, or If  $ZF = 1$ , go to next step.
8. Clear B, C, H, L registers and jump to step 10.
9. Move  $FF_H$  to C-register.
10. Store the content of H and L registers in memory.
11. Move C to L and B to H and store HL in memory.
12. Stop.

**Flowchart for example program 14**

**Assembly language program**

```

;PROGRAM TO SEARCH A GIVEN DATA IN AN ARRAY

        ORG 4100H    ;specify program starting address.

        LXI H,4200H  ;Set pointer for the data array.
        LDA 4250H    ;Load the given data in accumulator.
        MVI B,00H    ;Clear B-register to store the position.
LOOP:    INR B        ;Increment the position count.
        CMP M        ;Compare an element with given data.
        JZ AHEAD     ;If data is available,then ZF=1.
        MOV C,A      ;Save the given data in C-register.
        INX H
        MOV A,M      ;Get the next element of the array in A and
        CPI 20H      ;check for end of array.
        MOV A,C      ;Get the given data in A-register.
        JNZ LOOP     ;Repeat comparison until end of the array.
        LXI B,0000H  ;Clear B,C,H and L if given data
        LXI H,0000H  ;is not available in the array.
        JMP STORE
AHEAD:  MVI C,FFH    ;Move FFH to C, to indicate the
                ;availability of data.
STORE:  SHLD 4253H   ;Store the address of the data.
        MOV L,C
        MOV H,B
        SHLD 4251H   ;Store the position and indication
                ;for availability.
        HLT          ;Halt program execution.

        END          ;Assembly end.

```

**Assembler listing for example program 14**

```

1          ;PROGRAM TO SEARCH A GIVEN DATA IN AN ARRAY
2
3 4100          ORG 4100H    ;specify program starting address.
4
5 4100 21 00 42    LXI H,4200H ;Set pointer for the data array.
6 4103 3A 50 42    LDA 4250H  ;Load the given data in accumulator.
7 4106 06 00      MVI B,00H   ;Clear B-register to store the position.
8 4108 04          LOOP: INR B ;Increment the position count.
9 4109 BE          CMP M      ;Compare an element with given data.
10 410A CA 1F 41   JZ AHEAD   ;If data is available,then ZF=1.
11 410D 4F          MOV C,A    ;Save the given data in C-register.
12 410E 23          INX H
13 410F 7E          MOV A,M    ;Get the next element of the array in A and
14 4110 FE 20      CPI 20H     ;check for end of array.
15 4112 79          MOV A,C    ;Get the given data in A-register.
16 4113 C2 08 41   JNZ LOOP   ;Repeat comparison until end of the array.
17 4116 01 00 00   LXI B,0000H ;Clear B,C,H and L if given data
18 4119 21 00 00   LXI H,0000H ;is not available in the array.
19 411C C3 21 41   JMP STORE
20 411F 0E FF      AHEAD: MVI C,FFH ;Move FFH to C, to indicate the
21                ;availability of data.
22 4121 22 53 42   STORE: SHLD 4253H ;Store the address of the data.
23 4124 69          MOV L,C
24 4125 60          MOV H,B
25 4126 22 51 42   SHLD 4251H  ;Store the position and indication
26                ;for availability.
27 4129 76          HLT        ;Halt program execution.
28
29 412A          END          ;Assembly end.

```

**Sample data**

Input Data :  
 Array = 45<sub>H</sub>  
           72<sub>H</sub>  
           CA<sub>H</sub>  
           2F<sub>H</sub>  
           C2<sub>H</sub>  
           D1<sub>H</sub>  
           4F<sub>H</sub>  
           20<sub>H</sub>  
 Given Data = 2F<sub>H</sub>

Memory address	Content
4200	45
4201	72
4202	CA
4203	2F
4204	C2
4205	D1
4206	4F
4207	20

Output Data :  
 Availability = FF<sub>H</sub>  
 Position = 04<sub>H</sub>  
 Address = 4203<sub>H</sub>

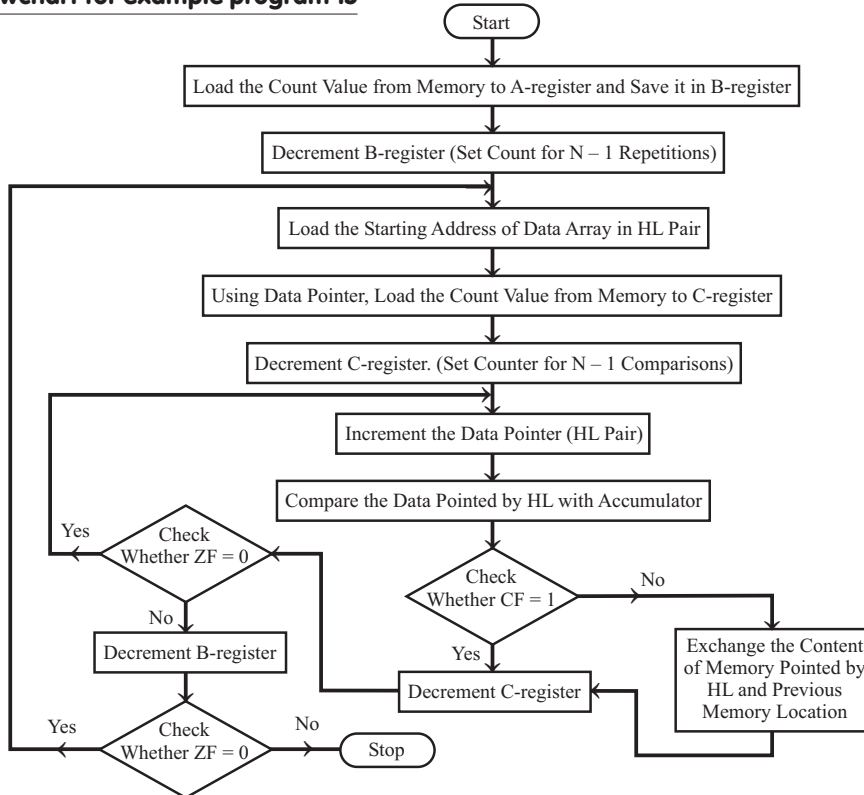
Memory address	Content
4250	2F
4251	FF
4252	04
4253	03
4254	42

**EXAMPLE PROGRAM - 15: Sorting an Array in Ascending Order**

Write an assembly language program to sort an array of data in ascending order. The array is stored in memory starting from 4200<sub>H</sub>. The first element of the array gives the count value for the number of elements in the array.

**Problem Analysis**

The algorithm for bubble sorting is given below. In bubble sorting of N-data, N-1 comparisons are carried by taking two consecutive data at a time. After each comparison, the data is rearranged such that smallest among the two is in the first memory location and the largest in the next memory location. (Here the data is rearranged within the two memory locations whose contents are compared). When we perform N-1 comparisons as mentioned above for N-1, times then the array consisting of N-data will be sorted in the ascending order.

**Flowchart for example program 15**

**Algorithm**

1. Load the count value from memory to A-register and save it in B-register.
2. Decrement B-register (B is counter for N-1 repetitions).
3. Set HL pair as data array address pointer.
4. Set C-register as counter for N-1 comparisons.
5. Load a data of the array in accumulator using the data address pointer.
6. Increment the HL pair (data address pointer).
7. Compare the data pointed by HL with accumulator.
8. If carry flag is set (If the content of accumulator is smaller than memory) then go to step 10, otherwise go to next step.
9. Exchange the content of memory pointed by HL and the accumulator.
10. Decrement C-register. If zero flag is reset go to step 6 otherwise go to next step.
11. Decrement B-register. If zero flag is reset go the step 3 otherwise go to next step.
12. Stop.

**Assembly language program**

```

;PROGRAM TO SORT AN ARRAY OF DATA IN ASCENDING ORDER

        ORG 4100H      ;specify program starting address.

        LDA 4200H      ;Load the count value in A-register.
        MOV B,A        ;Set count for N-1 repetitions
        DCR B          ;of N-1 comparisons.
LOOP2:  LXI H,4200H     ;Set pointer for array.
        MOV C,M        ;Set count for N-1 comparisons.
        DCR C
        INX H          ;Increment pointer.
LOOP1:  MOV A,M        ;Get one data of array in A.
        INX H
        CMP M          ;Compare next data with A-register.
        JC  AHEAD      ;If content of A is less than
                        ;memory then go to AHEAD.
        MOV D,M        ;If the content of A is greater than
        MOV M,A        ;the content of memory,
        DCX H          ;then exchange the content of memory
        MOV M,D        ;pointed by HL and previous location.
        INX H
AHEAD:  DCR C
        JNZ LOOP1      ;Repeat comparisons until C count is zero.
        DCR B
        JNZ LOOP2      ;Repeat N-1 comparisons until B count is zero.
        HLT            ;Halt program execution.

        END            ;Assembly end.

```

**Assembler listing for example program 15**

```

1          ;PROGRAM TO SORT AN ARRAY OF DATA IN ASCENDING ORDER
2
3 4100          ORG 4100H      ;specify program starting address.
4
5 4100 3A 00 42  LDA 4200H     ;Load the count value in A-register.
6 4103 47          MOV B,A     ;Set count for N-1 repetitions
7 4104 05          DCR B       ;of N-1 comparisons.
8 4105 21 00 42  LOOP2: LXI H,4200H ;Set pointer for array.
9 4108 4E          MOV C,M     ;Set count for N-1 comparisons.
10 4109 0D         DCR C
11 410A 23         INX H       ;Increment pointer.
12 410B 7E         LOOP1: MOV A,M ;Get one data of array in A.
13 410C 23         INX H
14 410D BE         CMP M      ;Compare next data with A-register.
15 410E DA 16 41   JC  AHEAD   ;If content of A is less than
16                ;memory then go to AHEAD.

```

```

17 4111 56          MOV D,M      ;If the content of A is greater than
18 4112 77          MOV M,A      ;the content of memory,
19 4113 2B          DCX H        ;then exchange the content of memory
20 4114 72          MOV M,D      ;pointed by HL and previous location.
21 4115 23          INX H
22 4116 0D          AHEAD: DCR C
23 4117 C2 0B 41     JNZ LOOP1    ;Repeat comparisons until C count is zero.
24 411A 05          DCR B
25 411B C2 05 41     JNZ LOOP2    ;Repeat N-1 comparisons until B count is zero.
26 411E 76          HLT          ;Halt program execution.
27
28 411F            END          ;Assembly end.

```

**Sample data**

Input Data: 07  
AB  
92  
84  
4F  
69  
F2  
34

Memory address	Content
4200	07
4201	AB
4202	92
4203	84
4204	4F
4205	69
4206	F2
4207	34

(Before sorting)

Output data: 07  
34  
4F  
69  
84  
92  
AB  
F2

Memory address	Content
4200	07
4201	34
4202	4F
4203	69
4204	84
4205	92
4206	AB
4207	F2

(After sorting)

**EXAMPLE PROGRAM - 16 : Sorting an Array in Descending Order**

*Write an assembly language program to sort an array of data in descending order. The array is stored in the memory location starting from 4200<sub>H</sub>. The first element of the array gives the count value for the number of elements in the array.*

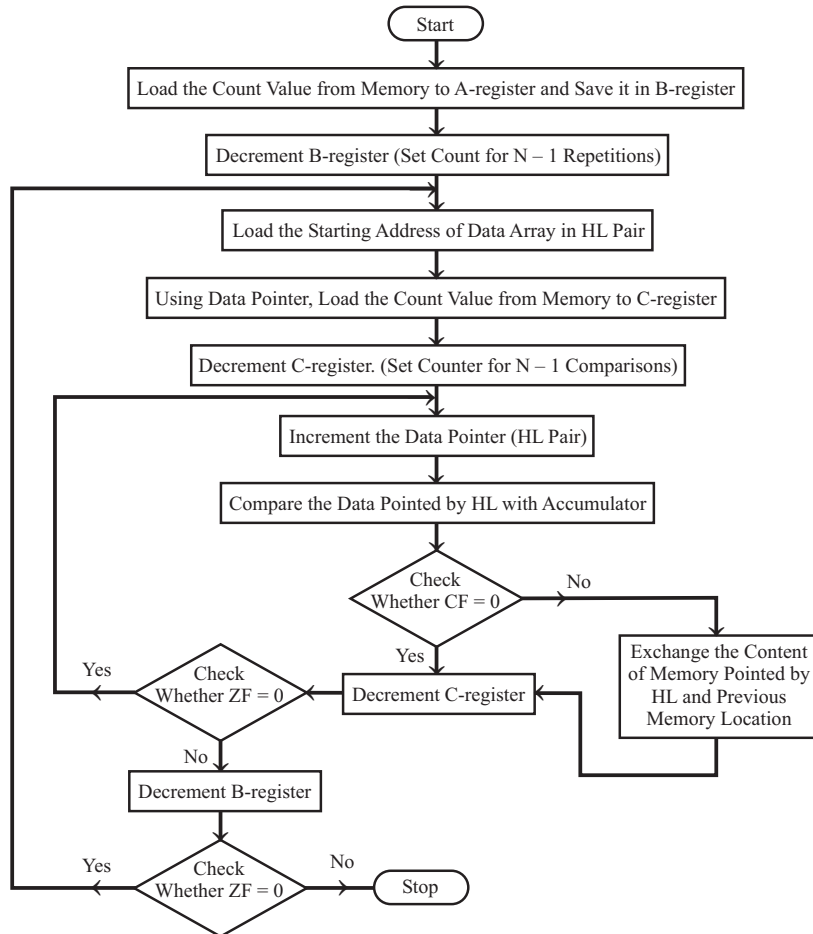
**Problem Analysis**

The algorithm for bubble sorting is given below. In bubble sorting of N-data, N-1 comparisons are carried by taking two consecutive data at a time. After each comparison, the data is rearranged such that largest among the two is in first memory location and the smallest in the next memory location. (Here the data is rearranged within the two memory locations whose contents are compared). When we perform N-1 comparisons as mentioned above for N-1 times, then the array consisting of N-data will be sorted in descending order.

**Algorithm**

1. Load the count value from memory to A-register and save it in B-register.
2. Decrement B-register (B is counter for N-1 repetitions).
3. Set HL pair as data array address pointer.
4. Set C-register as counter for N-1 comparisons.
5. Load a data of the array in accumulator using the data address pointer.
6. Increment the HL pair (data address pointer).
7. Compare the data pointed by HL with accumulator.
8. If carry flag is reset (If the content of accumulator is larger than memory) then go to step 10, otherwise go to next step.
9. Exchange the content of memory pointed by HL and the accumulator.
10. Decrement C-register. If zero flag is reset go to step 6 otherwise go to next step.
11. Decrement B-register. If zero flag is reset go the step 3 otherwise go to next step.
12. Stop.



**Flowchart for example program 16****Assembly language program**

```

;PROGRAM TO SORT AN ARRAY OF DATA IN DESCENDING ORDER

ORG 4100H      ;specify program starting address.

LDA 4200H      ;Load the count value in A-register.
MOV B,A        ;Set count for N-1 repetitions
DCR B          ;of N-1 comparisons.
LOOP2: LXI H,4200H ;Set pointer for array.
MOV C,M        ;Set count for N-1 comparisons.
DCR C
INX H          ;Increment the pointer.
LOOP1: MOV A,M   ;Get one data of array in A.
INX H          ;Compare the next data of array with
CMP M          ;the content of A-register.
JNC AHEAD      ;If content of A is greater than content
               ;of memory addressed by HL pair,
               ;then go to AHEAD.

```

```

MOV D,M      ;If the content of A is less than content
MOV M,A      ;of memory addressed by HL pair,
DCX H        ;then exchange content of memory pointed
MOV M,D      ;by HL and previous memory location.
INX H
AHEAD: DCR C
JNZ LOOP1    ;Repeat comparisons until C count is zero.
DCR B
JNZ LOOP2    ;Repeat N-1 comparisons until B count is zero.
HLT          ;Halt program execution.

END          ;Assembly end.

```

### Assembler listing for example program 16

```

1                ;PROGRAM TO SORT AN ARRAY OF DATA IN DESCENDING ORDER
2
3 4100            ORG 4100H      ;specify program starting address.
4
5 4100 3A 00 42    LDA 4200H     ;Load the count value in A-register.
6 4103 47          MOV B,A      ;Set counter for N-1 repetitions
7 4104 05          DCR B        ;of N-1 comparisons.
8 4105 21 00 42    LOOP2: LXI H,4200H ;Set pointer for array.
9 4108 4E          MOV C,M      ;Set count for N-1 comparisons.
10 4109 0D          DCR C
11 410A 23          INX H        ;Increment the pointer.
12 410B 7E          LOOP1: MOV A,M ;Get one data of array in A.
13 410C 23          INX H        ;Compare the next data of array with
14 410D BE          CMP M        ;the content of A-register.
15 410E D2 16 41    JNC AHEAD    ;If content of A is greater than content
16                ;of memory addressed by HL pair,
17                ;then go to AHEAD.
18 4111 56          MOV D,M      ;If the content of A is less than content
19 4112 77          MOV M,A      ;of memory addressed by HL pair,
20 4113 2B          DCX H        ;then exchange content of memory pointed
21 4114 72          MOV M,D      ;by HL and previous memory location.
22 4115 23          INX H
23 4116 0D          AHEAD: DCR C
24 4117 C2 0B 41    JNZ LOOP1    ;Repeat comparisons until C count is zero.
25 411A 05          DCR B
26 411B C2 05 41    JNZ LOOP2    ;Repeat N-1 comparison until B count is zero.
27 411E 76          HLT          ;Halt program execution.
28
29 411F            END          ;Assembly end.

```

### Sample data

Input Data :

	Memory address	Content
07	4200	07
C4	4201	C4
84	4202	84
9A	4203	9A
7B	4204	7B
E2	4205	E2
F4	4206	F4
B2	4207	B2

(Before sorting)

Output Data :

	Memory address	Content
07	4200	07
F4	4201	F4
E2	4202	E2
C4	4203	C4
B2	4204	B2
84	4205	84
7B	4206	7B
	4207	

(After sorting)

**EXAMPLE PROGRAM - 17 : Square Root of 8-Bit Binary Number**

*Write an assembly language program to find the square root of an 8-bit binary number. The binary number is stored in memory location 4200<sub>H</sub> and store the square root in 4201<sub>H</sub>.*

**Problem Analysis**

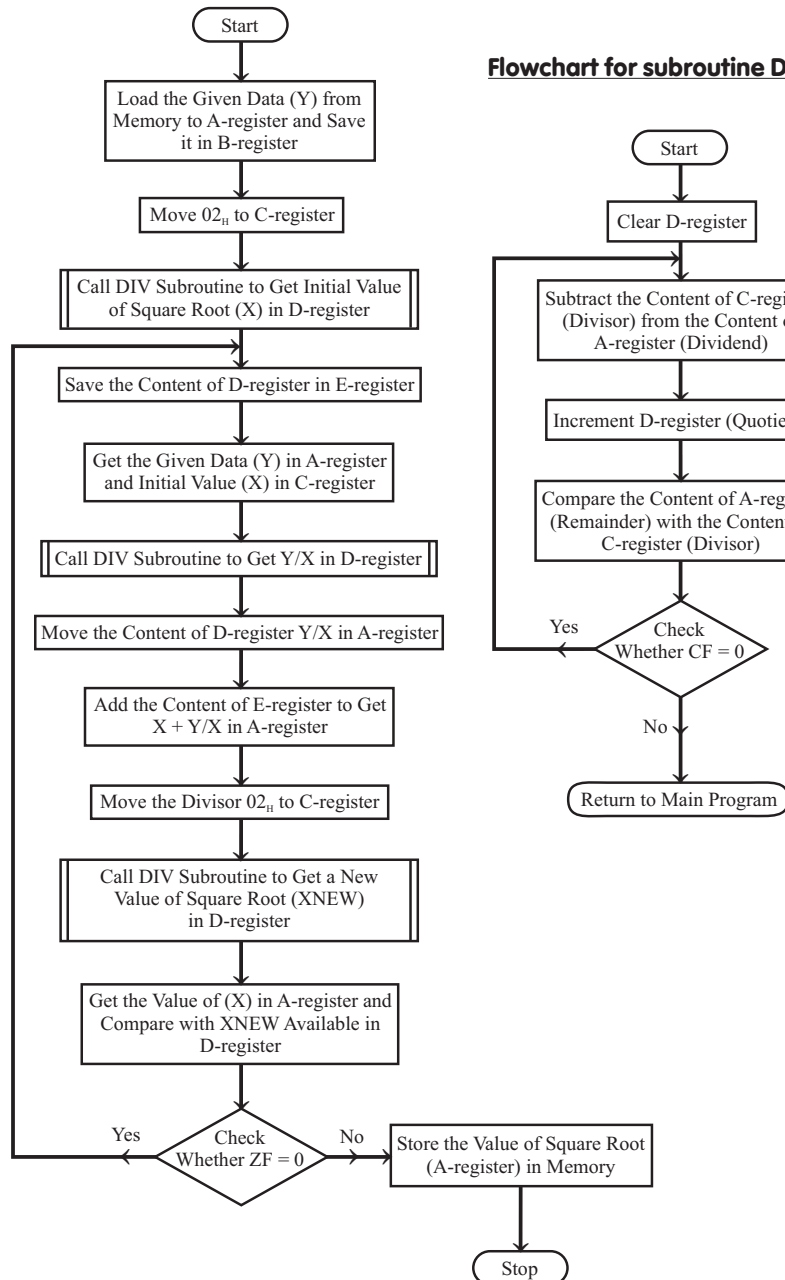
Square root can be computed by an iterative technique. First an initial value is assumed. Here the initial value of square root is taken as half the value of given number. The new value of square root is computed by using an expression,  $X_{NEW} = (X + Y/X)/2$  where X is the initial value of square root and Y is the given number. Then XNEW is compared with initial value. If they are not equal then the above process is repeated until X is equal to XNEW after taking XNEW as initial value, (i.e.,  $X \leftarrow X_{NEW}$ ).

**Algorithm**

1. Load the given data (Y) in A-register.
2. Save the content of A-register in B-register.
3. Move 02<sub>H</sub> (divisor) to C-register.
4. Call DIV subroutine to get initial value of square root (X) in D-register.
5. Save the content of D-register (initial value X) in E-register.
6. Move the given data (Y) from B-register to A-register.
7. Move the initial value (X) from D-register to C-register.
8. Call DIV subroutine to get Y/X in D-register.
9. Move the Y/X available in D-register to A-register.
10. Add the value of X in E-register to A-register to get  $X + Y/X$  in A-register.
11. Move 02<sub>H</sub> to C-register.
12. Call DIV subroutine to get new value of square root (XNEW) in D-register.
13. Compare X and XNEW.
14. If ZF = 1, go to next step. If ZF = 0, go to step 5.
15. Store the value of square root (A-register) in memory.
16. Stop.

**Algorithm for subroutine DIV**

1. Clear D-register.
2. Subtract the content of C-register (divisor) from the content of A-register (dividend).
3. Increment quotient (D-register).
4. Compare A-register and C-register.
5. If CF = 1, go to next step. If CF = 0 go to step 2.
6. Return to main program.

**Flowchart for example program 17**

**Assembly language program**

```

;PROGRAM TO FIND THE SQUARE ROOT OF 8-BIT BINARY NUMBER

        ORG 4100H ;specify program starting address.

        LDA 4200H ;Get the given data(Y) in A-register.
        MOV B,A   ;Save the data in B-register.
        MVI C,02H ;Get the divisor(02H) in C-register.
        CALL DIV  ;Call division subroutine to get initial
                  ;value(X) in the D-register.
REP:     MOV E,D   ;Save the initial value in E-register.
        MOV A,B   ;Get the dividend(Y) in A-register.
        MOV C,D   ;Get the divisor(X) in C-register.
        CALL DIV  ;Call division subroutine to get Y/X in D.
        MOV A,D   ;Move Y/X in A-register.
        ADD E     ;Get((Y/X)+X) in A-register.
        MVI C,02H ;Get the divisor (02H) in C-register.
        CALL DIV  ;Call division subroutine to get
                  ;XNEW in D-register.
        MOV A,E   ;Get X in A-register.
        CMP D     ;Compare X and XNEW.
        JNZ REP   ;If XNEW is not equal to X, then repeat.
        STA 4201H ;Save the square root in memory.
        HLT      ;Halt program execution.

;DIVISION SUBROUTINE
DIV:     MVI D,00H ;Clear D-register for quotient.
NEXT:    SUB C     ;Subtract the divisor from dividend.
        INR D     ;Increment the quotient.
        CMP C     ;Repeat subtraction until the divisor
        JNC NEXT  ;is less than dividend.
        RET      ;Return to main program.

        END      ;Assembly end.

```

**Assembler listing for example program 17**

```

1          ;PROGRAM TO FIND THE SQUARE ROOT OF 8-BIT BINARY NUMBER
2
3 4100          ORG 4100H ;specify program starting address.
4
5 4100 3A 00 42  LDA 4200H ;Get the given data(Y) in A-register.
6 4103 47          MOV B,A ;Save the data in B-register.
7 4104 0E 02      MVI C,02H ;Get the divisor(02H) in C-register.
8 4106 CD 1F 41   CALL DIV  ;Call division subroutine to get initial
9                  ;value(X) in the D-register.
10 4109 5A          REP: MOV E,D ;Save the initial value in E-register.
11 410A 78          MOV A,B ;Get the dividend(Y) in A-register.
12 410B 4A          MOV C,D ;Get the divisor(X) in C-register.
13 410C CD 1F 41   CALL DIV  ;Call division subroutine to get Y/X in D.
14 410F 7A          MOV A,D ;Move Y/X in A-register.
15 4110 83          ADD E   ;Get((Y/X)+X) in A-register.
16 4111 0E 02      MVI C,02H ;Get the divisor (02H) in C-register.
17 4113 CD 1F 41   CALL DIV  ;Call division subroutine to get
18                  ;XNEW in D-register
19 4116 7B          MOV A,E ;Get X in A-register.
20 4117 BA          CMP D   ;Compare X and XNEW.
21 4118 C2 09 41   JNZ REP  ;If XNEW is not equal to X, then repeat.
22 411B 32 01 42   STA 4201H ;Save the square root in memory.
23 411E 76          HLT     ;Halt program execution.
24
25
26          ;DIVISION SUBROUTINE
27
28 411F 16 00      DIV: MVI D,00H ;Clear D-register for quotient.
29 4121 91          NEXT: SUB C ;Subtract the divisor from dividend.

```

```

30 4122 14          INR D      ;Increment the quotient.
31 4123 B9          CMP C      ;Repeat subtraction until the divisor
32 4124 D2 21 41    JNC NEXT   ;is less than dividend.
33 4127 C9          RET        ;Return to main program.
34
35 4128             END        ;Assembly end.

```

**Sample data**Input Data : 64<sub>H</sub>Output Data : 0A<sub>H</sub>

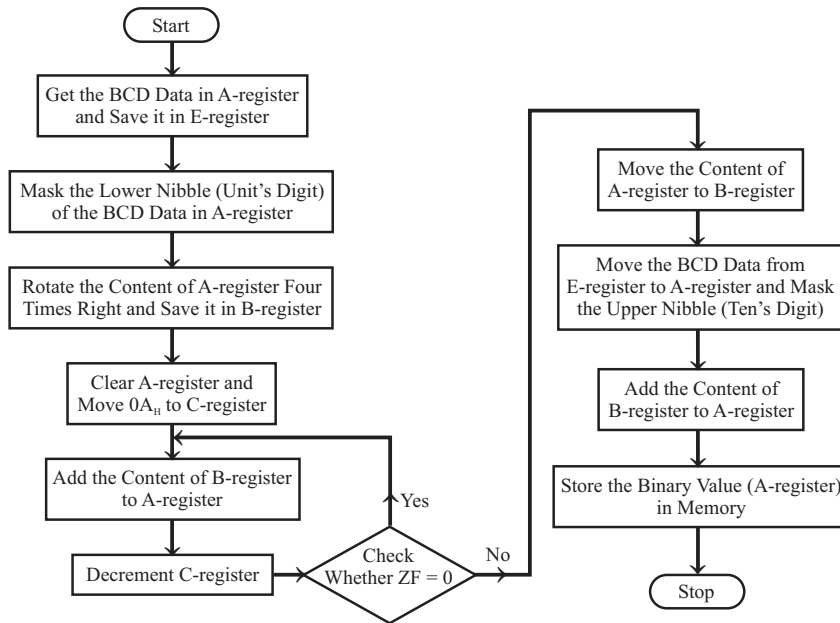
Memory address	Content
4200	64
4201	0A

**EXAMPLE PROGRAM - 18 : BCD to Binary Conversion**

Write an assembly language program to convert a two-digit BCD (8-bit) data to binary data. The BCD data is stored in 4200<sub>H</sub> and store the binary data in 4201<sub>H</sub>.

**Problem Analysis**

The 2-digit BCD data will have units digit and tens digit. When the tens digit (upper nibble) is multiplied by 0A<sub>H</sub> and the product is added to units digit (lower nibble), the result will be in binary, because the microprocessor performs binary arithmetic.

**Flowchart for example program 18****Algorithm**

1. Get the BCD data in A-register and save in E-register.
2. Mask the lower nibble (units) of the BCD data in A-register.
3. Rotate the upper nibble to lower nibble position and save in B-register.
4. Clear the accumulator.
5. Move 0A<sub>H</sub> to C-register.
6. Add B-register to A-register.
7. Decrement C-register. If ZF = 0 go to step 6. If ZF = 1, go to next step.
8. Save the product in B-register.

9. Get the BCD data in A-register from E-register and mask the upper nibble (tens).
10. Add the units (A-register) to product (B-register).
11. Store the binary value (A-register).
12. Stop.

### Assembly language program

```
;PROGRAM TO CONVERT 2-DIGIT BCD TO BINARY NUMBER

    ORG 4100H ;specify program starting address.

    LDA 4200H ;Get the data in A-register,
    MOV E,A   ;and save in E-register.
    ANI F0H   ;Mask the lower nibble (units digit).
    RLC       ;Rotate the upper nibble (tens digit)
    RLC       ;to lower nibble position and save in B.
    RLC
    RLC
    MOV B,A
    XRA A     ;Clear accumulator.
    MVI C,0AH ;Multiply tens digit by 0AH and
REP:  ADD B    ;get the product in A-register.
    DCR C
    JNZ REP
    MOV B,A   ;Save the product in B-register.
    MOV A,E   ;Get the BCD data in A-register.
    ANI 0FH   ;Mask the upper nibble (tens digit).
    ADD B     ;Get the binary data in A-register.
    STA 4201H ;Save the binary data in memory.
    HLT      ;Halt program execution.

    END      ;Assembly end.
```

### Assembler listing for example 18

```
1 ;PROGRAM TO CONVERT 2-DIGIT BCD TO BINARY NUMBER
2
3 4100 ORG 4100H ;specify program starting address.
4
5 4100 3A 00 42 LDA 4200H ;Get the data in A-register,
6 4103 5F MOV E,A ;and save in E-register.
7 4104 E6 F0 ANI F0H ;Mask the lower nibble (units digit).
8 4106 07 RLC ;Rotate the upper nibble (tens digit)
9 4107 07 RLC ;to lower nibble position and save in B.
10 4108 07 RLC
11 4109 07 RLC
12 410A 47 MOV B,A
13 410B AF XRA A ;Clear accumulator.
14 410C 0E 0A MVI C,0AH ;Multiply tens digit by 0AH and
15 410E 80 REP: ADD B ;get the product in A-register.
16 410F 0D DCR C
17 4110 C2 0E 41 JNZ REP
18 4113 47 MOV B,A ;Save the product in B-register.
19 4114 7B MOV A,E ;Get the BCD data in A-register.
20 4115 E6 0F ANI 0FH ;Mask the upper nibble (tens digit).
21 4117 80 ADD B ;Get the binary data in A-register.
22 4118 32 01 42 STA 4201H ;Save the binary data in memory.
23 411B 76 HLT ;Halt program execution.
24
25 411C END ;Assembly end.
```

### Sample data

Input Data : 45<sub>10</sub>  
 Output Data : 2D<sub>H</sub>

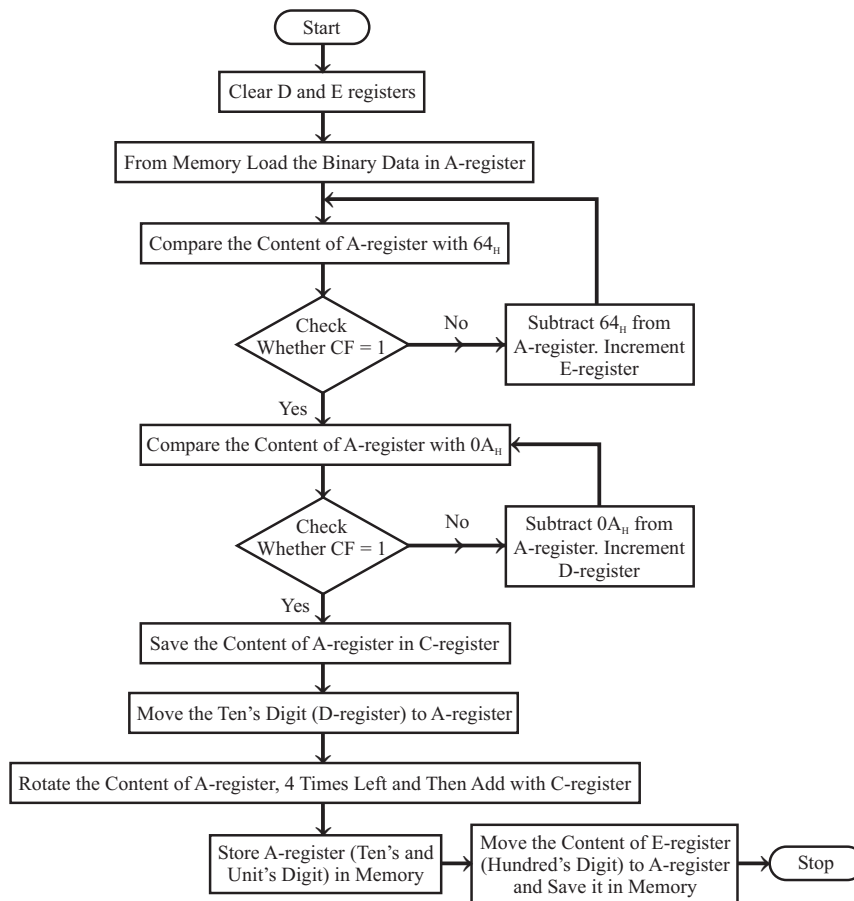
Memory address	Content
4200	45
4201	2D

**EXAMPLE PROGRAM - 19 : Binary to BCD Conversion**

Write an assembly language program to convert an 8-bit binary data to BCD. The binary data is stored in 4200<sub>H</sub>. Store the hundred's digit in 4251<sub>H</sub>. Store the ten's and unit's digits in 4250<sub>H</sub>.

**Problem Analysis**

The maximum value of 8-bit binary is  $FF_H = 256_{10}$ . Hence the maximum size of the data will have hundreds, tens and units. The algorithm given below uses two counters to count hundreds and tens. Initially counters are cleared. First let us subtract all hundreds from the binary data. For each subtraction, hundred's register is incremented by one. Then, let us subtract all tens. For each subtraction, ten's register is incremented by one. The remaining will be units. The tens and units are combined to form 2-digit BCD (8-bit binary).

**Flowchart for example program 19****Algorithm**

1. Clear D and E registers to account for hundreds and tens.
2. Load the binary data in A-register.
3. Compare A-register with 64<sub>H</sub>. If carry flag is set, go to step 7 otherwise go to next step.
4. Subtract 64<sub>H</sub> from A-register.
5. Increment E-register (Hundred's register).



6. Go to step 3.
7. Compare the A-register with  $0A_H$ . If carry flag is set, go to step 11, otherwise go to next step.
8. Subtract  $0A_H$  from A-register.
9. Increment D-register (ten's register).
10. Go to step 7.
11. Combine the units and tens to form 8-bit result.
12. Save the units, tens and hundreds in memory.

### Assembly language program

```
;PROGRAM TO CONVERT 8-BIT BINARY NUMBER TO BCD

      ORG 4100H ;specify program starting address.

      MVI E,00H ;Clear E-register for hundreds and
      MOV D,E   ;D-register for tens.
      LDA 4200H ;Get the binary data in A-register.
HUND: CPI 64H   ;Compare, whether data is less than 64H(100).
      JC TEN    ;If the content of A is less than
               ;100 or 64H then go to TEN.
      SUI 64H   ;Subtract all hundreds from the data and
      INR E     ;for each subtraction increment E-register.
      JMP HUND
TEN:   CPI 0AH   ;Compare whether the content of A
      JC UNIT   ;is less than 0AH or 10.If CF=1 go to UNIT.
      SUI 0AH   ;Subtract all tens from the data and for
      INR D     ;each subtraction increment D-register.
      JMP TEN
UNIT:  MOV C,A   ;Save the units in C-register.
      MOV A,D   ;Get tens in A-register.
      RLC      ;Rotate ten's digit to upper nibble position.
      RLC
      RLC
      RLC
      ADD C     ;Combine ten's and unit's digits.
      STA 4250H ;Save tens and units in memory.
      MOV A,E
      STA 4251H ;Save hundreds in memory.
      HLT      ;Halt program execution.

      END      ;Assembly end.
```

### Assembler listing for example program 19

```
1          ;PROGRAM TO CONVERT 8-BIT BINARY NUMBER TO BCD
2
3  4100          ORG 4100H ;specify program starting address.
4
5  4100 1E 00    MVI E,00H ;Clear E-register for hundreds and
6  4102 53      MOV D,E   ;D-register for tens.
7  4103 3A 00 42 LDA 4200H ;Get the binary data in A-register.
8  4106 FE 64    HUND: CPI 64H ;Compare, whether data is less than 64H(100).
9  4108 DA 11 41 JC TEN    ;If the content of A is less than
10                ;100 or 64H then go to TEN.
11 410B D6 64    SUI 64H   ;Subtract all hundreds from the data and
12 410D 1C      INR E     ;for each subtraction increment E-register.
13 410E C3 06 41 JMP HUND
14 4111 FE 0A    TEN:  CPI 0AH ;Compare whether the content of A
15 4113 DA 1C 41 JC UNIT   ;is less than 0AH or 10.If CF=1 go to UNIT.
16 4116 D6 0A    SUI 0AH   ;Subtract all tens from the data and for
17 4118 14      INR D     ;each subtraction increment D-register.
18 4119 C3 11 41 JMP TEN
19 411C 4F      UNIT: MOV C,A ;Save the units in C-register.
20 411D 7A      MOV A,D   ;Get tens in A-register.
21 411E 07      RLC      ;Rotate ten's digit to upper nibble position.
```

```

22  411F  07      RLC
23  4120  07      RLC
24  4121  07      RLC
25  4122  81      ADD C      ;Combine ten's and unit's digits.
26  4123  32 50 42 STA 4250H ;Save tens and units in memory.
27  4126  7B      MOV A,E
28  4127  32 51 42 STA 4251H ;Save hundreds in memory.
29  412A  76      HLT        ;Halt program execution.
30
31  412B                END      ;Assembly end.

```

**Sample Data**

Input Data : B9<sub>H</sub>  
Output Data : 0185<sub>10</sub>

Memory address	Content	
4200	B9	} BCD data
4250	85	
4251	01	

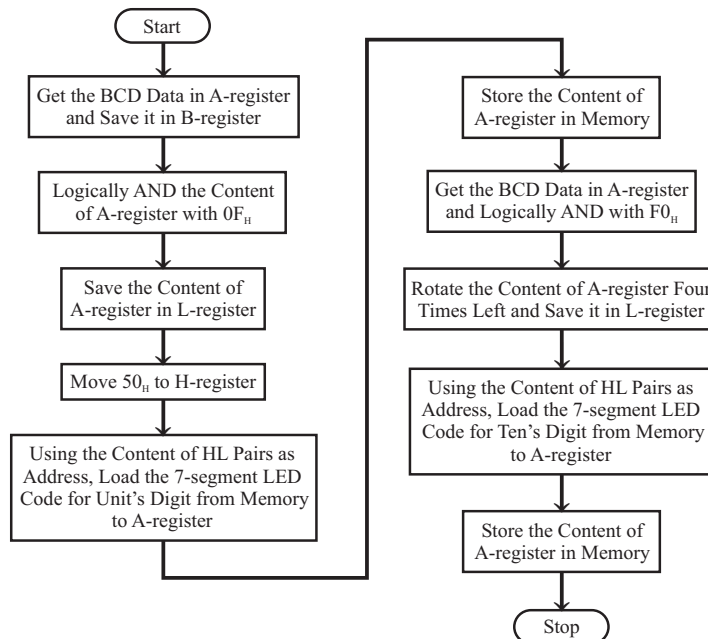
**EXAMPLE PROGRAM - 20 : BCD to 7-Segment LED Code**

*Write an assembly language program to find the 7-segment LED code for a 2-digit BCD data, by using the look up table. The BCD data is stored in 4200<sub>H</sub>. Store the 7-segment code in 4201<sub>H</sub> and 4202<sub>H</sub>.*

**Problem Analysis**

The 7-segment LED codes for decimal digit 0 to 9 are determined and stored in memory locations from 5000<sub>H</sub> to 5009<sub>H</sub> respectively. The look-up table is created such that the low order address is same as that of decimal digit. Hence, by this method the high order address is fixed (50) and the low order address is the decimal digit itself.

In order to find the 7-segment code, the BCD data is split into lower nibble and upper nibble. The code is determined by taking each nibble as low order address of the look up table.

**Flowchart for example program 20**

**Algorithm**

1. Load the BCD data in A-register and save in B-register.
2. Logically AND A-register with  $0F_H$  to mask upper nibble (ten's digit).
3. Move A-register to L-register and move  $50_H$  to H-register.
4. Get the LED code for lower nibble (unit's digit) in A-register and store in memory.
5. Move the BCD data from B-register to A-register and mask the lower nibble (unit's digit).
6. Rotate the upper nibble to lower nibble position.
7. Move A-register to L-register.
8. Get the LED code for ten's digit in A-register and store in memory.
9. Stop.

**Assembly language program**

```
;PROGRAM TO FIND THE 7-SEGMENT LED CODE FOR A BCD DATA
```

```
ORG 4100H ;specify program starting address.

LDA 4200H ;Get BCD data in A and save in B.
MOV B,A
ANI 0FH ;Mask the upper nibble (ten's digit).
MOV L,A ;Get memory address of LED code
MVI H,50H ;for unit's digit, in HL pair.
MOV A,M ;Get LED code for unit's digit in A
STA 4201H ;and store in memory.
MOV A,B ;Get the BCD data in A-register and
ANI 0FH ;mask the lower nibble (unit's digit).
RLC ;Rotate upper nibble to
RLC ;lower nibble position.
RLC
RLC
MOV L,A ;Get memory address of LED code
;for ten's digit in HL pair.
MOV A,M ;Get LED code for ten's digit in A
STA 4202H ;and store in memory.
HLT ;Halt program execution.

END ;Assembly end.
```

**Assembler listing for example program 20**

```
1 ;PROGRAM TO FIND THE 7-SEGMENT LED CODE FOR A BCD DATA
2
3 4100 ORG 4100H ;specify program starting address.
4
5 4100 3A 00 42 LDA 4200H ;Get BCD data in A and save in B.
6 4103 47 MOV B,A
7 4104 E6 0F ANI 0FH ;Mask the upper nibble (ten's digit).
8 4106 6F MOV L,A ;Get memory address of LED code
9 4107 26 50 MVI H,50H ;for unit's digit, in HL pair.
10 4109 7E MOV A,M ;Get LED code for unit's digit in A
11 410A 32 01 42 STA 4201H ;and store in memory.
12 410D 78 MOV A,B ;Get the BCD data in A-register and
13 410E E6 F0 ANI 0FH ;mask the lower nibble (unit's digit).
14 4110 07 RLC ;Rotate upper nibble to
15 4111 07 RLC ;lower nibble position.
16 4112 07 RLC
17 4113 07 RLC
18 4114 6F MOV L,A ;Get memory address of LED code
19 ;for ten's digit in HL pair.
20 4115 7E MOV A,M ;Get LED code for ten's digit in A
21 4116 32 02 42 STA 4202H ;and store in memory.
22 4119 76 HLT ;Halt program execution.
23
24 411A END ;Assembly end.
```

**Sample data 1 :****Loop-up table for Common cathode 7-segment LED**

Memory address	Content
5000	3F
5001	06
5002	5B
5003	4F
5004	66

Memory address	Content
5005	6D
5006	7D
5007	07
5008	7F
5009	6F

Input Data :  $45_{10}$   
Output Data :  $6D_H$   
 $66_H$

Memory address	Content
4200	45
4201	6D
4202	66

**Sample data 2 :****Loop-up table for Common anode 7-segment LED**

Memory address	Content
5000	C0
5001	F9
5002	A4
5003	80
5004	99

Memory address	Content
5005	92
5006	82
5007	F8
5008	80
5009	90

Input Data :  $45_{10}$   
Output Data :  $92_H$   
 $99_H$

Memory address	Content
4200	45
4201	92
4202	99

*Note : For 7-segment display code please refer Chapter-9 section 9.4.*

**EXAMPLE PROGRAM - 21 : Binary to ASCII Conversion**

*Write an assembly language program to convert an 8-bit binary (2-digit hexa) to ASCII code. The binary data is stored in  $4200_H$  and store the ASCII code in  $4201_H$  and  $4202_H$ .*

**Problem Analysis**

Each Hexa digit (4-bit binary) is represented by an 8-bit ASCII. The Hexa digit 0 through 9 are represented by  $30_H$  to  $39_H$  in ASCII. Hence, for Hexa 0 to 9, if we add  $30_H$ , we will get the corresponding ASCII. The Hexa digit A through F are represented by  $41_H$  to  $46_H$  in ASCII. Hence, for Hexa digit A to F if we add  $37_H$  we will get the corresponding ASCII.

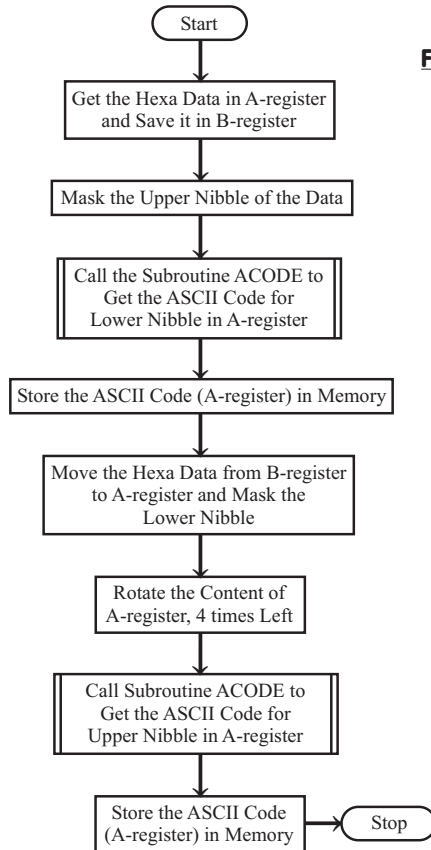
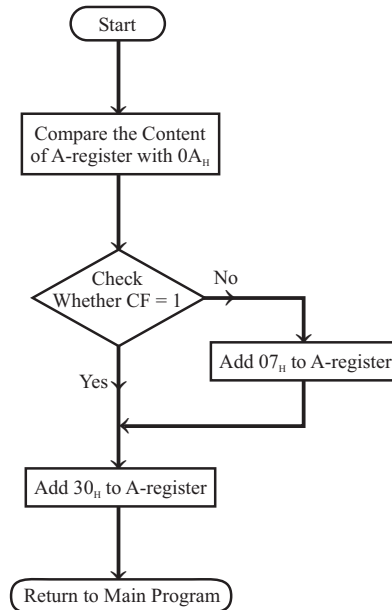
In the following algorithm the given 8-bit data is split into two nibbles. The ASCII code for each nibble is found by calling a subroutine, which takes care of adding  $30_H$  to the nibble if it is less than  $0A_H$ , or adding  $37_H$  if the nibble is greater than  $09_H$ .

**Algorithm**

1. Load the given data in A-register and move to B-register.
2. Mask the upper nibble of the binary (hexa) data in A-register.
3. Call subroutine ACODE to get ASCII code of the lower nibble and store in memory.
4. Move B-register to A-register and mask the lower nibble.
5. Rotate the upper nibble to lower nibble position.
6. Call subroutine ACODE to get the ASCII code of upper nibble and store in memory.
7. Stop.

**Algorithm for subroutine code**

1. Compare the content of A-register with  $0A_H$ .
2. If CF = 1, go to step 4. If CF = 0, go to next step.
3. Add  $07_H$  to A-register.
4. Add  $30_H$  to A-register.
5. Return to main program.

**Flowchart for example program 21****Flowchart for subroutine ACODE****Assembly language program**

;PROGRAM TO CONVERT 8-BIT BINARY TO ASCII CODE

```

ORG 4100H ;specify program starting address.

LDA 4200H ;Get binary data in A.
MOV B,A  ;Save the binary data in B-register.
ANI 0FH  ;Mask the upper nibble.
CALL ACODE ;Call subroutine to get ASCII code for
STA 4201H ;lower nibble in A and store in memory.
MOV A,B  ;Get data in A-register.
ANI F0H  ;Mask the lower nibble.
RLC      ;Rotate upper nibble to
RLC      ;lower nibble position.
RLC
RLC
CALL ACODE ;Call subroutine to get ASCII code for
STA 4202H ;upper nibble in A and store in memory.
HLT       ;Halt program execution.
  
```

```

;SUBROUTINE ACODE
ACODE: CPI 0AH ;If the content of A is less than 0AH,
      JC SKIP ;then add 30H to A otherwise
      ADI 07H ;add 37H to A-register.
SKIP:  ADI 30H
      RET      ;Return to main program.

      END      ;Assembly end.

```

**Assembler listing for example program 21**

```

1          ;PROGRAM TO CONVERT 8-BIT BINARY TO ASCII CODE
2
3  4100          ORG 4100H ;specify program starting address.
4
5  4100 3A 00 42  LDA 4200H ;Get binary data in A.
6  4103 47          MOV B,A ;Save the binary data in B-register.
7  4104 E6 0F      ANI 0FH ;Mask the upper nibble.
8  4106 CD 1A 41   CALL ACODE ;Call subroutine to get ASCII code for
9  4109 32 01 42   STA 4201H ;lower nibble in A and store in memory.
10 410C 78          MOV A,B ;Get data in A-register.
11 410D E6 F0      ANI F0H ;Mask the lower nibble.
12 410F 07          RLC      ;Rotate upper nibble to
13 4110 07          RLC      ;lower nibble position.
14 4111 07          RLC
15 4112 07          RLC
16 4113 CD 1A 41   CALL ACODE ;call subroutine to get ASCII code for
17 4116 32 02 42   STA 4202H ;upper nibble in A and store in memory.
18 4119 76          HLT      ;Halt program execution.
19
20
21          ;SUBROUTINE ACODE
22 411A          ACODE: CPI 0AH ;If the content of A is less than 0AH,
23 411A FE 09      JC SKIP ;then add 30H to A otherwise
24 411C DA 21 41   ADI 07H ;add 37H to A-register.
25 411F C6 07      SKIP: ADI 30H
26 4121 C6 30      RET      ;Return to main program.
27 4123 C9          END      ;Assembly end.
28
29 4124

```

**Sample data**

Input Data : E4<sub>H</sub>  
 Output Data : 34 (ASCII code for 4)  
               45 (ASCII code for E)

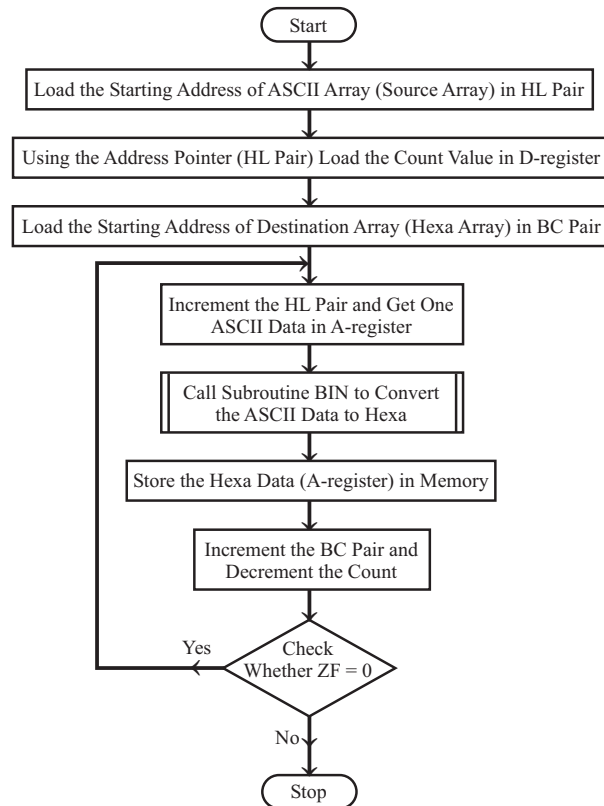
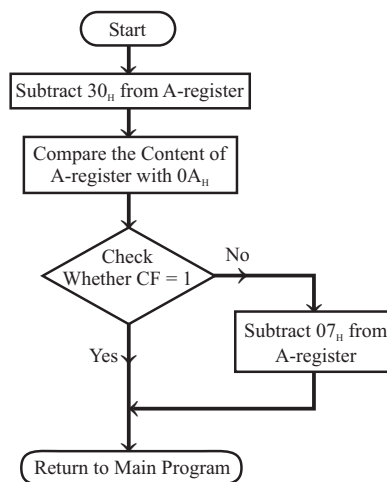
Memory address	Content
4200	E4
4201	34
4202	45

**EXAMPLE PROGRAM - 22 : ASCII to Binary Conversion**

*Write an assembly language program to convert an array of ASCII codes to corresponding binary (hexa) value. The ASCII array is stored starting from 4200<sub>H</sub>. The first element of the array gives the number of elements in the array.*

**Problem Analysis**

The hexa digit 0 through 9 are represented by 30<sub>H</sub> to 39<sub>H</sub> in ASCII. Hence, for ASCII code 30<sub>H</sub> to 39<sub>H</sub> if we subtract 30<sub>H</sub> then we will get the corresponding binary (hexa) value. The hexa digit A through F are represented by 41<sub>H</sub> to 46<sub>H</sub> in ASCII. Hence for ASCII code 41<sub>H</sub> to 46<sub>H</sub> we have to subtract 37<sub>H</sub> to get corresponding binary (hexa) value. In the following algorithm, a subroutine has been written to subtract either 30<sub>H</sub> or 37<sub>H</sub> from the given data.

**Flowchart for example program 22****Flowchart for subroutine BIN**

**Algorithm**

1. Set HL pair as pointer for ASCII array.
2. Set D-register as count for number of data in the array.
3. Set BC pair as pointer for binary (hexa) array.
4. Increment HL pair and move a data of ASCII array to A-register.
5. Call subroutine BIN to find the binary (hexa) value.
6. The binary (hexa) value available in A-register is stored in memory.
7. Increment BC pair.
8. Decrement D-register. If ZF = 0, then go to step 4. If ZF = 1, then stop.

**Algorithm for subroutine BIN**

1. Subtract 30<sub>H</sub> from A-register.
2. Compare the content of A-register with 0A<sub>H</sub>.
3. If CF = 1, go to step 5. If CF = 0, go to next step.
4. Subtract 07<sub>H</sub> from A-register.
5. Return to main program.

**Assembly language program**

```

;PROGRAM TO CONVERT ASCII CODE TO BINARY VALUE

    ORG 4100H    ;specify program starting address.

    LXI H,4200H ;Set pointer for ASCII array.
    MOV D,M     ;Set count for number of data.
    LXI B,4300H ;Set pointer for binary(hexa) array.
LOOP: INX H
    MOV A,M     ;Get an ASCII data in A-register.
    CALL BIN    ;Call subroutine to get binary
    STAX B      ;value in A and store in memory.
    INX B       ;Increment the binary array pointer.
    DCR D
    JNZ LOOP    ;Repeat conversion until count is zero.
    HLT         ;Halt program execution.

;SUBROUTINE BIN
BIN:  SUI 30H    ;subtract 30H from the data.
     CPI 0AH
     RC         ;If CF=1, Return to main program.
     SUI 07H    ;If data is greater than 0AH, then subtract
     RET       ;07H and return to main program.

    END         ;Assembly end.

```

**Assembler listing for example program 22**

```

1          ;PROGRAM TO CONVERT ASCII CODE TO BINARY VALUE
2
3 4100          ORG 4100H    ;specify program starting address.
4
5 4100 21 00 42    LXI H,4200H ;Set pointer for ASCII array.
6 4103 56          MOV D,M     ;Set count for number of data.
7 4104 01 00 43    LXI B,4300H ;Set pointer for binary(hexa) array.
8 4107 23          LOOP: INX H
9 4108 7E          MOV A,M     ;Get an ASCII data in A-register.
10 4109 CD 13 41    CALL BIN    ;Call subroutine to get binary
11 410C 02          STAX B      ;value in A and store in memory.
12 410D 03          INX B       ;Increment the binary array pointer.
13 410E 15          DCR D
14 410F C2 07 41    JNZ LOOP    ;Repeat conversion until count is zero.
15 4112 76          HLT         ;Halt program execution.

```



```

16 4113
17
18 ;SUBROUTINE BIN
19
20 4113 D6 30 BIN: SUI 30H ;Subtract 30H from the data.
21 4115 FE 0A CPI 0AH
22 4117 D8 RC ;If CF = 1, Return to main program.
23 4118 D6 07 SUI 07H ;If data is greater than 0AH then subtract
24 411A C9 RET ;07H and return to main program.
25
26 411B END ;Assembly end.

```

**Sample data**

Input Data :

Count : 07

ASCII Array: 31  
42  
35  
46  
43  
39  
38

Memory address	Content
4200	07
4201	31
4202	42
4203	35
4204	46
4205	43
4206	39
4207	38

Output Data :

Binary array= 01

0B  
05  
0F  
0C  
09  
08

Memory address	Content
4300	01
4301	0B
4302	05
4303	0F
4304	0C
4305	09
4306	08

**5.11 SUMMARY**

- A macro is a group of instructions written within brackets and identified by a name.
- A subroutine is defined as a group of instructions written separately from the main program to perform a function that occurs repeatedly in the main program.
- The instructions CALL addr16 and RET are used to implement a subroutine in 8085.
- Delay routines are subroutines used for maintaining the timings of various operations in a microprocessor.
- In a delay routine a count value is loaded in a register and it is decremented one by one until it becomes zero.
- List is a linked data structure used in programming techniques. The linked data structure will have a number of components linked in a particular fashion.
- The different types of linked data structures are linear linked lists, linked lists with multiple pointers, circular linked lists and trees.
- An array is a series of data of the same type stored in successive memory locations.
- Flowchart is graphical representation of the operation flow of the program.
- The microprocessor development system usually contains a microcomputer (PC), emulator and software tools like editor, assembler, linker, locator, debugger and simulator.
- An editor is a software which, when run on a microcomputer, allows the user to type and modify the assembly language program statements.
- An assembler is a software used to translate the assembly language program to machine language program.
- The assembler generates two output files. They are object file and list file.
- The different types of assembler are One-pass assembler, Two-pass assembler, Macro assembler, Cross assembler, Resident assembler and Meta assembler.
- The assembler directive are the instructions to the assembler regarding the program being assembled.
- An assembler identifies syntax errors but it cannot identify the logical errors.
- In one-pass assemblers, the source code is processed only once.
- In two-pass assembler labels are assigned addresses in the first pass. In the second pass the source codes are translated to machine codes.

- A macro assembler is a type of two-pass assembler that allows the programmer to write the source code in macros.
- Cross assemblers are assemblers developed in high-level languages and they are machine independent.
- A resident assembler is an assembler that can be run on a machine, which can execute the source code. For example, an 8085 assembler that is written in 8085 assembly language is a resident assembler.
- A meta assembler is an assembler that supports many different microprocessors.
- A linker is a program used to join together several object files into one large object file.
- A locator is a program used to assign the specific addresses to object codes which have to be loaded into memory.
- A debugger is a program which allows the user to troubleshoot or to identify and correct the errors in a program.
- The simulator is a program which can be run on the development system to simulate the operations of the newly designed systems.
- An emulator is usually used to test and debug the hardware and software of a newly designed system.

## 5.12 SHORT QUESTIONS AND ANSWERS

---

### 5.1 What is meant by a program?

A program is a set of instructions written to perform a certain task.

### 5.2 What is assembler, interpreter and compiler?

**(a) Assembler :** It is a software that converts assembly language program codes to machine language codes.

**(b) Compiler :** It is a software that converts the programs written in high level language to machine language.

**(c) Interpreter:** It is similar to a compiler but it converts the instructions one by one.

### 5.3 What is the need for a assembler?

An assembler is used to translate assembly language programs to machine language programs (i.e., in the executable format). Without the assembler it is very difficult to convert very large assembly language programs to machine codes.

### 5.4 What are the advantages of an assembler?

The advantages of an assembler are:

1. The assembler translates mnemonics into binary code with speed and accuracy.
2. It allows the programmer to use variables in the program.
3. It is easy to alter the program and reassemble.
4. The assembler identifies the syntax errors.
5. The assembler can reserve memory locations for data or result.
6. The assembler provides list file for documentation.

### 5.5 What are assembler directives or pseudo instructions?

Assembler directives are the instructions to the assembler regarding the program being assembled. They are also called pseudo instructions or pseudo opcodes.

The assembler directives will give information like start and end of a program, values of variables used in the program, storage locations for input and output data, etc.

5.6 List the assembler directives of a typical 8085 assembler.

The assembler directives of a typical 8085 assembler are the following :

Assembler directive	Function
DB	Define Byte. Used to define byte type variable.
DW	Define word. Used to define 16-bit variable.
END	Indicates the end of the program.
ENDM	End of macro. Indicates the end of a macro sequence.
EQU	Equate. Used to equate numeric value or constant to a variable.
MACRO	Defines the name, parameters and start of a macro.
ORG	Origin. Used to assign the starting address for a program.

5.7 What is a macro and when is it used?

A macro is a group of instructions written within brackets and identified by a name. A macro is written when a repeated group of instructions is too short or not appropriate to be written as a subroutine.

5.8 What is the meaning of expanding the macro?

While assembling a program, the assembler replaces the instructions represented by a macro in the place where macro is called. This is called expanding the macro.

5.9 What is the disadvantage in a macro?

The disadvantage in a macro is that, if it is expanded or used a number of times in a program then the program may occupy more memory.

5.10 What is a subroutine (or procedure)?

A subroutine (or procedure) is a group of instructions written separately from the main program to perform a function that occurs repeatedly in the main program.

5.11 What are the advantages of a subroutine?

1. Modular programming : The various tasks in a program can be developed as separate modules and called in the main program.
2. Reduction in the amount of work and program development time.
3. Reduces memory requirement for program storage.

5.12 How is a subroutine implemented in 8085?

A subroutine is written as a separate program (Procedure) and stored in a separate memory location. The subroutine program should be terminated by an RET instruction. It is called in the main program using CALL addr16 instruction. The addr16 should be the starting address of the subroutine program.

When the CALL instruction is executed, the processor saves the return address (address of next instruction) in stack, load the subroutine address in Program Counter(PC) and starts executing the subroutine. At the end of subroutine when the RET instruction is executed, the return address is retrieved from the stack and loaded to the PC.

## 5.13. What is a Delay routine?

A delay routine is a subroutine used for maintaining the timings of various operations in microprocessor.

## 5.14. Write a simple delay subroutine involving a single 8-bit register of 8085.

**Delay subroutine**

```

MVI C,d8 ;Load the count value (d8) in C-register.
LOOP: DCR C ;Decrement the count
      NOP
      NOP
      JNZ LOOP ;If ZF = 0, go to LOOP.
      RET ;Return to main program.

```

## 5.15. Write a simple delay subroutine involving a register pair of 8085.

**Delay subroutine**

```

LXI D,d16 ;Load the count value (d16) in DE register pair.
LOOP: DCX D ;Decrement the count.
      MOV A,E ;Logically OR the content of
      ORA D ;E-register with D-register.
      JNZ LOOP ;If ZF = 0, go to LOOP.
      RET ;Return to main program.

```

## 5.16. What is a list?

List is a linked data structure used in programming techniques. The linked data structure will have a number of components linked in a particular fashion. Each component will consist of a string data and a pointer next to the component.

## 5.17. What are the types of linked data structures?

The different types of linked data structures are linear linked lists, linked lists with multiple pointers, circular linked lists and trees.

## 5.18. What is an array?

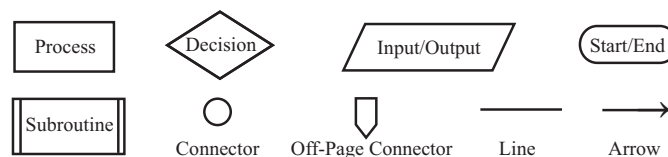
An array is a series of data of the same type stored in successive memory locations. Each value in the array is referred to as an element of the array.

## 5.19. What is a flowchart?

A flowchart is graphical representation of the operation flow of a program. It is the graphical (pictorial) form of an algorithm.

## 5.20. List the symbols used for drawing a flowchart.

The following are the symbols used for drawing flowchart:



**Fig. Q5.20 :** Symbols used in flowcharts.

5.21 *What is a development system? What are its components?*

A development system is a system used by the microprocessor-based system designer to design and test the software and hardware aspects of a new system under development.

The components of a development system are a microcomputer with standard accessories, emulator and program development tools like editor, assembler, linker, locator, debugger, simulator, etc.

5.22 *Write a short note on assembly language program development tools.*

The program development tools for an assembly language program are editor, assembler, linker, locator, debugger and simulator. These tools are softwares that can be run on the development system in order to write, assemble, debug, modify and test the assembly language programs.

5.23 *What is an Editor?*

An editor is a program which when run on a microcomputer system, allows the user to type and modify the assembly language program statements. The main function of an editor is to help the user to construct the assembly language program in the right format and save as a file.

5.24 *What is a one-pass assembler?*

A one-pass assembler is an assembler in which the source codes are processed only once. A one-pass assembler is very fast and in one-pass assembler only backward reference may be used.

5.25 *What is a two-pass assembler?*

The two-pass assembler is an assembler in which the source codes are processed two times. In the first pass, the assembler assigns addresses to all the labels and attach values to all the variables used in the program. In the second pass it converts the source code into machine code.

5.26 *What is the drawback of a one-pass assembler?*

The drawback of a one-pass assembler is that the program cannot have forward reference, because, a one-pass assembler issues an error message if it encounters a label or variable that is defined at a later part of a program.

5.27 *What is linker and locator?*

(a) A linker is a program used to join together several object files into one large object file.

(b) A locator is a program used to assign specific addresses to the object codes to be loaded into memory.

5.28 *What is debugging?*

The process of locating and correcting an error using a debugger is known as debugging.

5.29 *What is a debugger?*

A debugger is a software used to locate and troubleshoot errors in a program.

5.30 *What is a simulator?*

A simulator is a program which can be run on the development system to simulate the operations of the newly designed system. Some operations that can be simulated are given below:

- Execute a program and display the result.
- Single step execution of a program.
- Break-point execution of a program.
- Display the content of register/memory.

## 5.31 What is an emulator?

An emulator is a system that can be used to test the hardware and software of a newly developed microprocessor-based system.

## 5.32 What is the difference between an emulator and a simulator?

A simulator can be used to run and check the software of a newly developed microprocessor-based system but an emulator can be used to run and check both the hardware and software of a newly developed microprocessor-based system.

## 5.33 Write a subroutine to output the content of flag register to LED's connected to the port of a 8085 microprocessor-based system.

**Subroutine to display the content of flag register**

```
PUSH PSW ;Push the A-register and flag register to stack.
POP B ;POP the top of stack to BC pair.
MOV A,C ;Get the content of flag register in A-register.
OUT PORT ;Output the content of A-register to PORT.
RET ;Return to main program.
```

*Note : The A-register and Flag register are together called PSW (Program Status Word). In this, the A-register is high order register and Flag register is low order register.*

## 5.34 Write a simple program to find the smallest among the two data stored in memory.

**(Assume that data are stored in 4200<sub>H</sub> and 4201<sub>H</sub>. Store the result in 5000<sub>H</sub>)**

```
LDA 4200H ;Get first data in A-register.
MOV B,A ;Save first data in B-register.
LDA 4201H ;Get second data in A-register.
CMP B ;Compare the two data.
JC AHEAD ;If CF = 1, go to AHEAD.
MOV A,B ;If CF = 0, move B-register to A-register.
AHEAD: STA 5000H ;Store the smallest data in memory
HLT ;Stop
```

5.35 Write a simple program to multiply an 8-bit data stored at 4200<sub>H</sub> by 02<sub>H</sub> and store the result at 4300<sub>H</sub> and 4301<sub>H</sub>.

```
MVI B,00H ;Clear B-register.
XRA A ;Clear A-register and carry
LDA 4200H ;Get the data in A-register.
RAL ;Multiply the content of A by 02.
JNC AHEAD ;If CF = 0, go to AHEAD.
INR B ;If CF = 1, increment B-register.
AHEAD: STA 4300H ;Store the product in memory.
MOV A,B
STA 4301H ;Store the carry in memory.
HLT ;Halt program execution.
```

5.36 Write a simple program to divide an 8-bit data stored at 4200<sub>H</sub> by 02<sub>H</sub> and store the result at 4300<sub>H</sub> and 4301<sub>H</sub>.

```
MVI B,00H ;Clear B-register.
XRA A ;Clear A-register and carry
LDA 4200H ;Get the data in A-register.
```

```

        RAR          ;Divide the content of A by 02.
        JNC AHEAD    ;If CF = 0, go to AHEAD.
        INR B        ;If CF = 1, increment B-register.
AHEAD:  STA 4300H     ;Store the quotient in memory.
        MOV A,B
        STA 4301H     ;Store the remainder in memory.
        HLT          ;Halt program execution.

```

5.37 Write a simple program to split a hexa data into two nibbles and store in memory.

```

        LXI H,4200H  ;Set pointer for data array.
        MOV B,M      ;Get the data in B-register.
        MOV A,B      ;Copy the data to A-register.
        ANI 0FH      ;Mask the upper nibble.
        INX H
        MOV M,A      ;Store the lower nibble in memory.
        MOV A,B      ;Get the data in A-register
        ANI 0FH      ;Mask the lower nibble.
        RRC          ;Bring the upper nibble to lower nibble position.
        RRC
        RRC
        RRC
        INX H
        MOV M,A      ;Store the upper nibble in memory.
        HLT          ;Halt program execution.

```

5.38 Explain the mathematical functions performed by the following instructions.

```

        MVI A,07H
        RLC
        MOV B,A
        RLC
        RLC
        ADD B

```

The operations performed by each of the above given mathematical instruction are as follows:

1. An 8-bit data  $07_H$  is moved to A-register.
2. The content of A-register is multiplied by 02.
3. The content of A-register ( $07 \times 02 = 0E_H$ ) is copied to B-register.
4. The content of A-register is multiplied by 02.
5. The content of A-register is multiplied by 02.
6. The content of B-register is added to B-register.

The result of the above operations is that the content of A-register is multiplied by  $0A_H$ . Therefore, after executing the above instructions, the content of A-register will be  $46_H$ .

5.39 Write a subroutine to clear a flag register and an accumulator.

**Subroutine to clear flag register**

```
LXI SP,4200H ;Initialize stack.
LXI B,0000H ;Clear BC register pair.
PUSH B      ;Push the content of BC pair to stack.
POP PSW     ;Pop the top of stack to A-register and flag register (PSW).
RET         ;Return to main program.
```

**Note :** The A-register and Flag register are together called PSW (Program Status Word). The A-register is a high order register and the flag register is a low order register.

5.40 Write a subroutine program to exchange the content of BC pair and DE pair?

**Subroutine to exchange BC pair and DE pair**

```
LXI SP,4200H ;Initialize stack.
PUSH B      ;Store the content of BC pair in stack.
PUSH D      ;Store the content of DE pair in stack.
POP B       ;Move the content of DE pair stored in stack to BC pair.
POP D       ;Move the content of BC pair stored in stack to DE pair.
RET         ;Return to main program.
```



**CHAPTER 6**

---

**PERIPHERAL DEVICES AND INTERFACING**

---

**6.1 PROGRAMMABLE PERIPHERAL DEVICES**

---

Programmable peripheral devices are designed to perform various input/output functions and specific routine activities. Every programmable device will have one or more control registers. The programmable devices can be set up to perform specific functions by writing control words into the control registers. The control word is an instruction which informs the peripheral about the various functions it has to perform. The format of the control word will be specified by the manufacturer of the peripheral devices.

INTEL has developed a number of peripheral devices that can be used with 8085/8086/8088-based systems. Some of the peripheral devices developed by INTEL for 8085/8086/8088-based system are Parallel peripheral interface-8255, Serial communication interface-8251, Keyboard/Display controller-8279, Programmable Timer-8253/8254 and DMA controllers-8237 and 8257. A brief discussion about these devices and their interfacing with 8085 processor is presented in this chapter.

Parallel peripheral interface-8255 is used to interface a slow IO device to the fast processor and to achieve an efficient data transfer between them. The USART is used to provide serial communication between the processor and another system. The 8279 is used to relieve the processor from time consuming routine activities like keyboard scanning and display refreshing. Programmable timers are used to maintain various timings and to initiate time-based activities. DMA controllers are used to achieve very fast data transfer between memory and IO devices by bypassing the processor.

**6.2 PARALLEL DATA COMMUNICATION INTERFACE**

---

In microprocessor-based systems, digital information can be transmitted from one system to another system either by parallel or serial data transfer scheme.

In parallel data transfer, a group of bits (for eg., 8 bits) is transmitted from one device to another at any one time. To achieve parallel data transfer scheme, a group of data lines will be connecting the processor and peripheral devices. Normally in microprocessor-based systems the parallel data transfer schemes are adopted to transfer data between various devices inside the system.

Basically the microprocessor-based system has been fabricated on a PCB (**Printed Circuit Board**) in which a bus is formed with the required number of data lines and the bus connects all the devices in the system. The data transmitted over the bus in a PCB are highly reliable. In a well designed board, there will not be any loss of data and the data will not be corrupted.

When data has to be transmitted over longer distances (i.e., greater than 0.5m), we require high current signals to drive the data for longer distance. In such cases data is transmitted bit by bit through a single data line.

### 6.2.1 Parallel Data Transfer Schemes

Data transfer schemes refer to the method of data transfer between the processor and peripheral devices. In a typical microcomputer, data transfer takes place between any two devices: microprocessor and memory, microprocessor and IO devices, or memory and IO devices. For effective data transfer between these devices, the timing parameters of the devices should be matched. But most of the devices have incompatible timings. For example, an IO device may be slower than the processor due to which it cannot send data to the processor at the expected time.

Semiconductor memories are available with compatible timings. Moreover, slow memories can be interfaced using additional hardware to introduce wait states in machine cycles. The microprocessor system designer often faces difficulties while interfacing IO devices and magnetic memories (like floppy or hard disk) to achieve efficient data transfer to or from the microprocessor. Several data transfer schemes have been developed to solve the interfacing problems with IO devices.

The data transfer schemes have been broadly classified into the following two categories :

1. Programmed data transfer.
2. Direct memory access (DMA) data transfer.

In programmed data transfer, a memory resident routine (subroutine) requests the device for data transfer to or from one of the processor register.

Programmed data transfer scheme is used when relatively small amount of data is to be transferred. In these schemes, usually one byte or word of data is transferred at a time. Examples of devices using programmed data transfer are ADC, DAC, Hex-keyboard, 7-segment LEDs, etc.

Programmed data transfer scheme can be further classified into the following three types :

- a) Synchronous data transfer scheme.
- b) Asynchronous data transfer scheme.
- c) Interrupt driven data transfer scheme.

In DMA data transfer, the processor is forced to **HOLD** state (**high impedance** state) by an IO device until the data transfer between the device and the memory is complete. The processor does not execute any instruction during the **HOLD** period.

The DMA data transfer is used for large block of data transfer between IO device and memory. Typical examples of devices using DMA are CRT controller, floppy disk, hard disk, high speed line printer, etc.

The different types of DMA data transfer schemes are as follows :

- a) Cycle stealing DMA or Single transfer mode DMA.
- b) Block or Burst mode DMA.
- c) Demand transfer mode DMA.

Figure 6.1 shows the various types of data transfer schemes. All the data transfer schemes discussed above requires both software and hardware for their implementation. Within a microcomputer, more than one scheme can be used for interfacing different IO devices. However, some of these schemes require specific hardware features in the microprocessor for implementing the scheme.

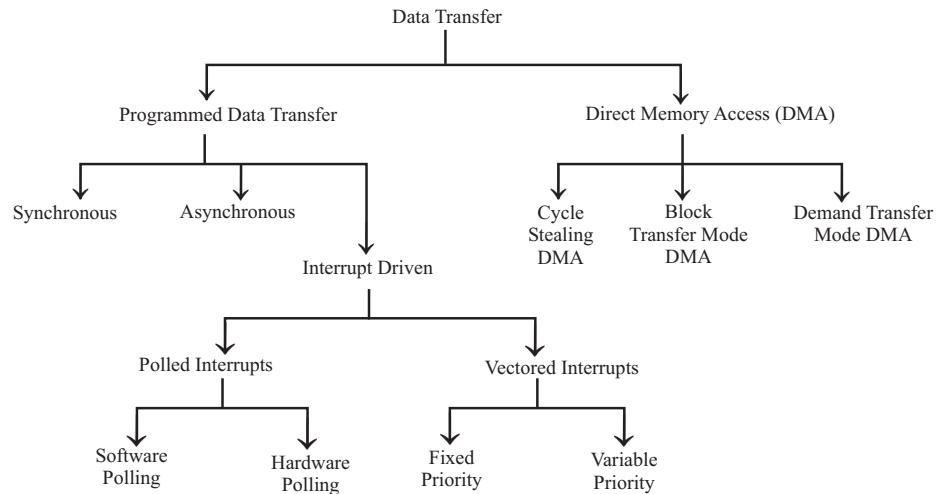


Fig. 6.1 : Types of data transfer schemes.

### Synchronous Data Transfer Scheme

The synchronous data transfer scheme is the simplest of all data transfer schemes. In this scheme the processor does not check the readiness of the device. The IO device or peripheral should have matched timing parameters. Whenever data is to be obtained from the device or transferred to the device, the user program can issue a suitable instruction for the device. At the end of the execution of this instruction, the transfer would have been completed.

The synchronous data transfer scheme can also be implemented with a small delay (if the delay is tolerable) after the request has been made. The sequence of operations for synchronous data transfer scheme is shown in Fig. 6.2. The mode-0 input/output in 8255 is an example of synchronous data transfer.

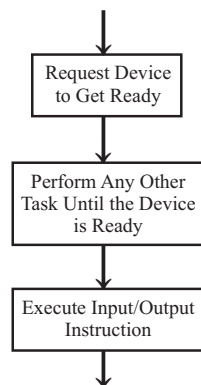


Fig. 6.2 : Synchronous data transfer scheme.

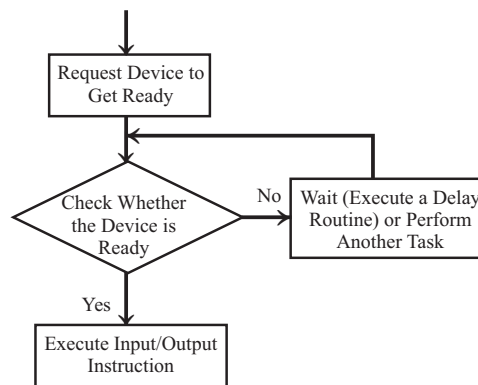


Fig. 6.3 : Asynchronous data transfer scheme.

### Asynchronous Data Transfer Scheme

The asynchronous data transfer scheme is employed when the speed of the processor and IO device does not match. In this scheme the processor sends a request to the device for read/write operation. Then the processor keeps on polling the status of the device. Once the device is ready, the processor executes a data transfer instruction to complete the process. To implement this scheme, the device should provide a signal which may be tested by the processor to ascertain whether it is ready or not.

The sequence of operations for asynchronous data transfer is shown in Fig. 6.3. The mode-1 and mode-2 handshake data transfer of 8255 without interrupt is an example of asynchronous data transfer.

### Interrupt Driven Data Transfer Scheme

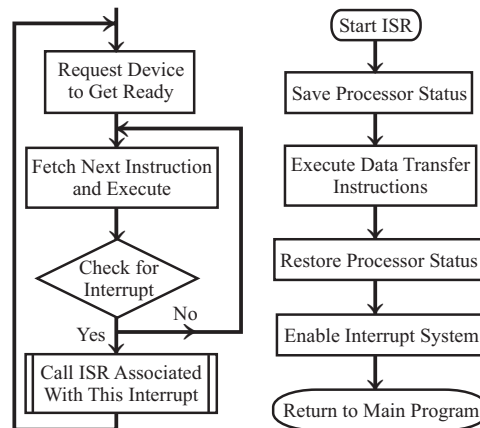
The interrupt driven data transfer scheme is the best method of data transfer for efficient utilization of processor time. In this scheme, the processor first initiates the IO device for data transfer. After initiating the device, the processor will continue the execution of instructions in the program. Also, at the end of every instruction the processor will check for a valid interrupt signal. If there is no interrupt then the processor will continue the execution.

*Note : The user/system designer need not write any subroutine/procedure to check for an interrupt. The logic of checking interrupt signals while executing each instruction is incorporated in the processor itself by the manufacturer of the processor.*

When the IO device is ready, it will interrupt the processor. On receiving an interrupt signal the processor will complete the current instruction execution and save the processor status in stack. Then the processor calls an **Interrupt Service Routine (ISR)** to service the interrupting device. At the end of ISR, the processor status is retrieved from the stack and the processor starts executing its main program. The sequence of operations for an interrupt driven data transfer scheme is shown in Fig. 6.4.

#### 6.2.2 INTEL 8212

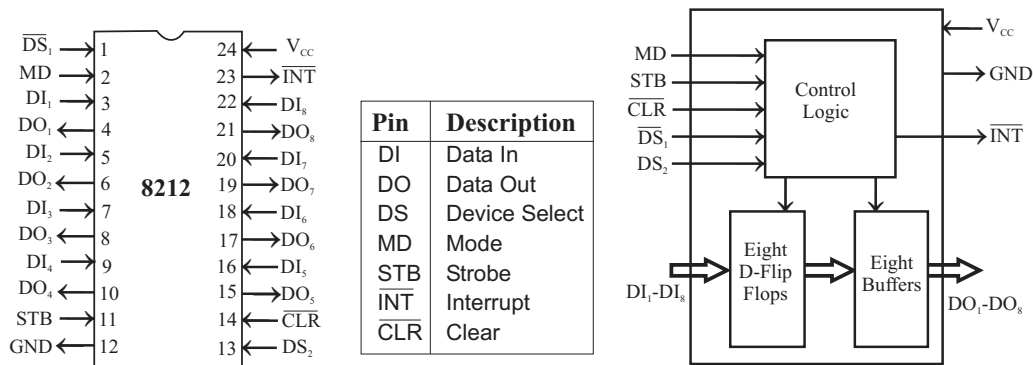
INTEL 8212 is a 24-pin IO device with eight number of D-type latches, each followed by a tristate buffer. It has eight input lines ( $DI_1$  to  $DI_8$ ) and eight output lines ( $DO_1$  to  $DO_8$ ). The 8212 can be used either as a latch or as a tristate buffer and the function is determined by pin MD (mode). The INTEL 8212 can be used either as input port or output port. When 8212 is used as output port, the MD pin is tied **high** and it will work as a latch. When 8212 is used as input port, the MD pin is tied **low** and it will work as a tristate buffer. In a system the 8212 is permanently connected to work either as input or as output and the function cannot be reversed. It has two device select signals  $\overline{DS}_1$  (active **low**) and  $DS_2$  (active **high**) and three control pins  $\overline{CLR}$  (clear),  $\overline{STB}$  (strobe) and  $\overline{INT}$  (Interrupt). The pin description of 8212 and its internal block diagram are shown in Fig. 6.5.



**Fig. a :** Main program execution sequence. **Fig. b :** ISR execution sequence.

**Fig. 6.4 :** Interrupt driven data transfer scheme.

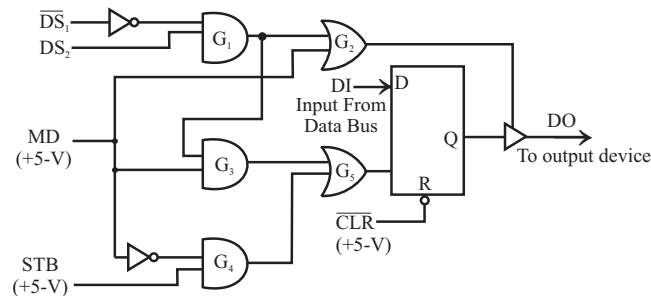
*Note : In microprocessor-based systems, the input port should be a tristate buffer and the output port should be a latch.*



**Fig. 6.5 :** 8212 pin description and internal (functional) block diagram.

The INTEL 8212 can be used for simple data transfer or data transfer with handshake signals. The strobe and interrupt signals are used for handshake data transfer. For simple data transfer, the STB is permanently tied **high** and  $\overline{\text{INT}}$  is not connected (not used) in the system.

The output logic of 8212 is shown in simplified form in the Fig. 6.6. The input lines  $DI_1$  to  $DI_8$  are connected to data bus of microprocessor system and the output lines  $DO_1$  to  $DO_8$  are connected to the output device. In the output mode, MD, STB and  $\overline{CLR}$  are **high**. When MD is **high**, the output of gate  $G_2$  is **high**, which enables the tristate buffer. The D-flip-flop functions as a latch. Now, the output of gate  $G_4$  is **low**, which makes the STB signal non-functional. When the device is selected by chip select circuit by making  $\overline{DS}_1 = 0$  and  $DS_2 = 1$ , the output signals of gates  $G_1$ ,  $G_3$  and  $G_5$  goes **high** and the clock signal of the flip-flop goes **high**. The data on pins DI flow to the output of the flip-flops and are latched when the clock pulse goes **low**.



**Fig. 6.6 : 8212 Output control logic.**

The INTEL 8212 functions as an input device when the mode signal is **low**. Figure 6.7 shows the simplified logic of the 8212 in the input mode. The input lines DI<sub>1</sub> to DI<sub>8</sub> are connected to input device and the output lines DO<sub>1</sub> to DO<sub>8</sub> are connected to data bus of microprocessor system. When the mode pin is **low**, all tristate buffers are disabled until the device is selected. However, when the STB is **high**, the output of G<sub>4</sub> and G<sub>5</sub> goes **high** and external data can be loaded into the flip-flops even if the 8212 is not selected. When the microprocessor selects the 8212, the tristate buffers are enabled and the data flow from output of D-latch (Q) to the data bus.

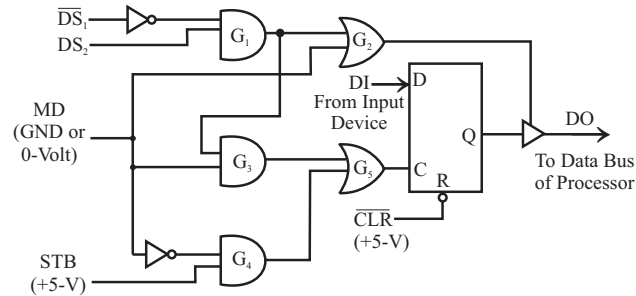


Fig. 6.7 : 8212 Input control logic.

### 6.2.3 Programmable IO Port and Timer - INTEL 8155/8156

The INTEL 8155 includes 256 bytes of RAM memory, three IO ports and a timer. The 8156 is identical with the 8155 except that the 8156 requires active high Chip Enable (CE).

Functionally 8155 can be viewed as two independent chips, one having static RAM and the other having IO ports and a timer. The IO section of 8155 includes 2 numbers of 8-bit parallel IO ports called port-A and port-B, one number of 6-bit port called port-C and a programmable timer.

All the ports can be configured as simple input or output ports. Ports A and B can be programmed in the handshake mode. In the handshake mode, each port uses three signals as handshake signals and the port-C pins are used for handshake signals. When some of the port-C pins are used for handshake signals, the remaining pins can be used as simple input or output lines.

The timer has a 14-bit counter which can be programmed to work in four operating modes. The internal block diagram of 8155 and its internal decoding logic are shown in Fig. 6.8.

The control logic of the 8155 is specifically designed to eliminate the need for external demultiplexing of  $AD_0 - AD_7$  and generating separate control signals for memory and IO. The ALE,  $IO/\overline{M}$ ,  $\overline{RD}$  and  $\overline{WR}$  signals from the 8085 can be connected directly to 8155.

The ports and the timer of 8155 are IO-mapped in the system. Hence an 8-bit address is used to select the internal devices. Actually the internal devices require a 3-bit address to select any one of the five internal devices as shown in Table-6.1. The remaining address lines are decoded to produce the chip select signal. (For interfacing of 8155 with 8085, please refer to Chapter-3, Design Example-6.)

**TABLE - 6.1 : INTERNAL ADDRESS  
OF 8155**

Internal device	Internal address		
	$A_2$	$A_1$	$A_0$
Control register/ Status register	0	0	0
Port-A	0	0	1
Port-B	0	1	0
Port-C	0	1	1
LSB timer	1	0	0
MSB timer	1	0	1

Ports are programmed to work as input or output port in simple or handshake modes. The timer is also programmed to work in any one of the four operating modes. The programming of the ports and the timer is accomplished by writing a control word in the control register. The control word is framed in the specified format as shown in Fig. 6.9 and then loaded in the control register. Each bit of control word defines a function as given in Table-6.2.

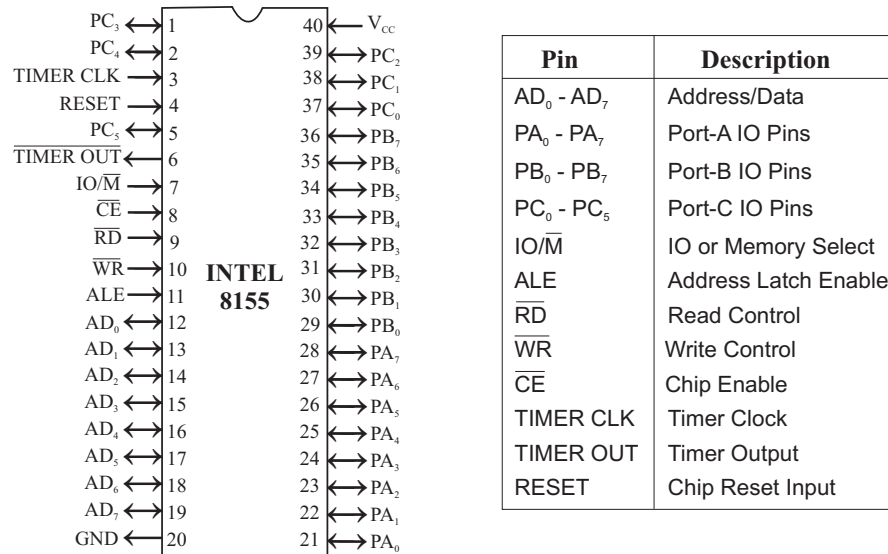


Fig. a : Pin description of 8155.

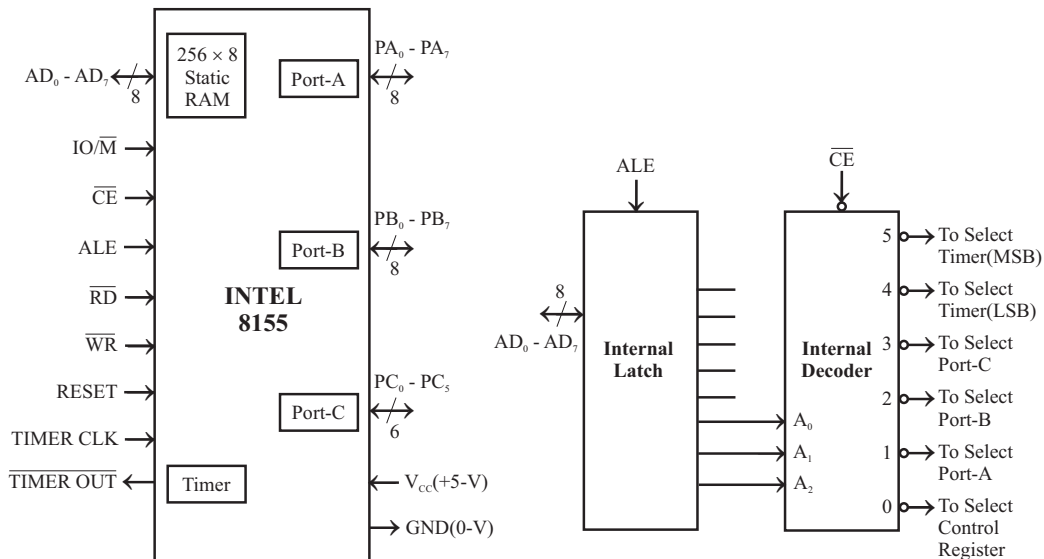
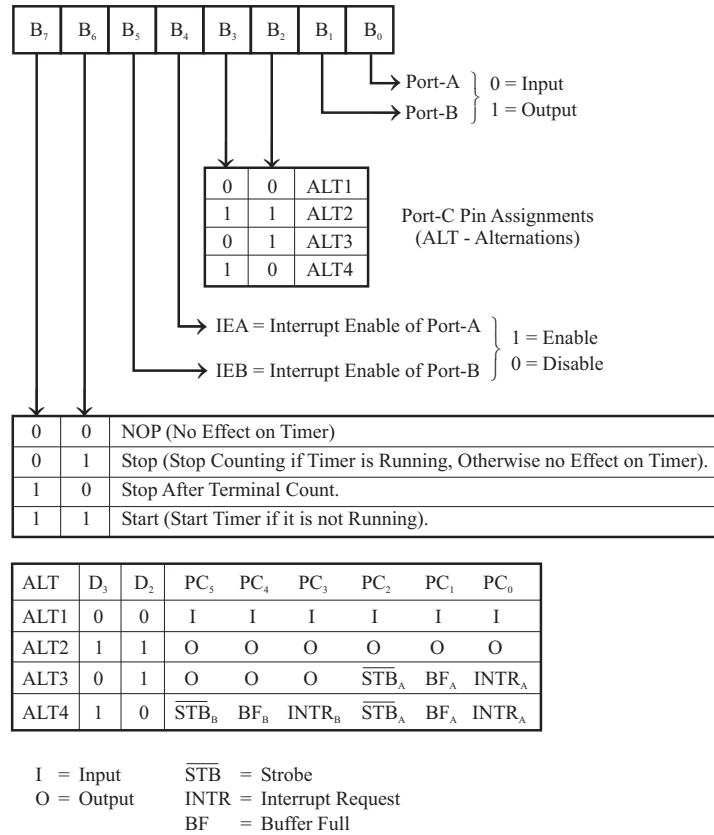


Fig. b : Internal block diagram of 8155.

Fig. c : Internal decoding logic of 8155.

Fig. 6.8 : Internal block diagram of 8155 and its internal decoding logic.



(The Subscript B denotes Port-B and the Subscript A denotes Port-A Signal).

**Fig. 6.9 : Control word format of 8155.**

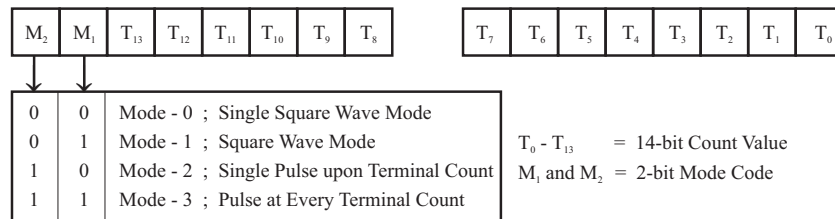
**TABLE - 6.2 : FUNCTIONS DEFINED BY CONTROL WORD**

Bit	Function defined
D <sub>0</sub>	Determine input or output function of port-A.
D <sub>1</sub>	Determine input or output function of port-B.
D <sub>2</sub> and D <sub>3</sub>	Determine the functions of port-C.
D <sub>4</sub> and D <sub>5</sub>	Used to enable or disable the internal flip-flop of 8155. If this internal flip-flop is enabled, then it generates an interrupt signal during handshake mode of IO.
D <sub>6</sub> and D <sub>7</sub>	Used for timer control.

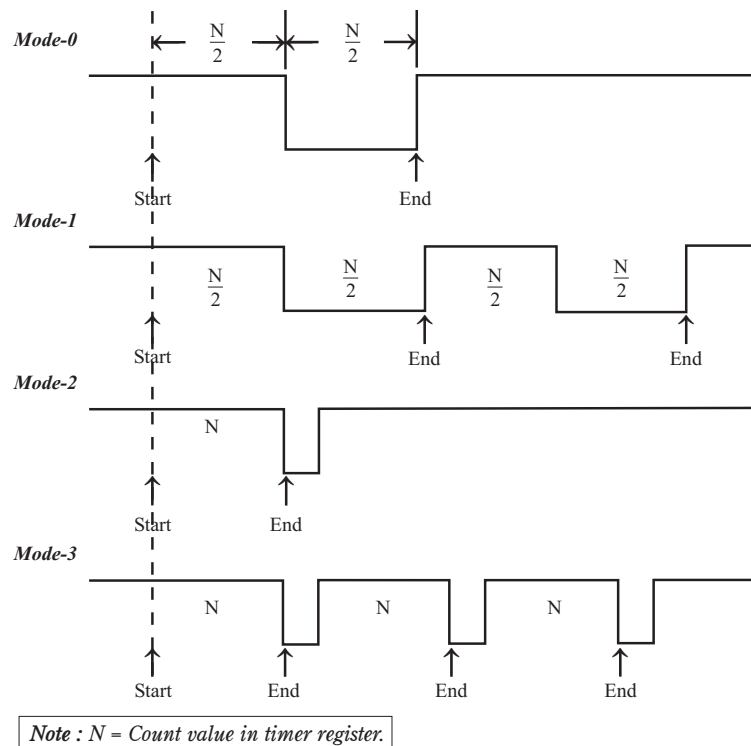


The timer section of the 8155 has two 8-bit registers as shown in Fig. 6.10 and in this register a 14-bit count value and a two bit mode code should be loaded. The timer requires an input clock signal and for every clock pulse, the timer decrements the count value by one. An appropriate control word starts the counter which decrements the count at each clock pulse. The timer output will vary according to the mode specified. The timer can be stopped either during counting or at the end of the count. In addition, the actual count at a given moment can be obtained by reading the status register.

The timer output normally remains **high** (while the timer is not running). When the timer starts running the output changes according to the mode of operation. The timer output for different operating modes are shown in Fig. 6.11.



**Fig. 6.10 :** Timer count value format.

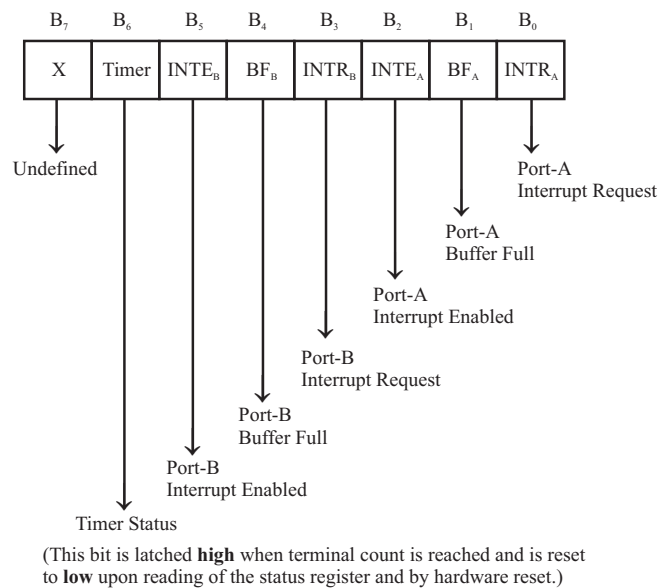


**Fig. 6.11 :** Output waveforms of the timer in 8155.

- Mode-0 : In this mode the timer output remains **high** for half the count and goes **low** for the remaining count, thus providing a single square wave.
- Mode-1 : In this mode, the timer output is a square wave. The initial timer count is automatically reloaded at the end of each count. The timer output remains **high** for half the count and remains **low** for the other count.
- Mode-2 : In this mode a single clock pulse is provided at the end of the count. The width of the pulse is equal to the time period of the input clock pulse.
- Mode-3 : This is similar to mode-2, except that the initial count is reloaded to provide a continuous waveform.

The INTEL 8155 has a status register which can be read by the processor by using the control register address. The control register and the status register have the same port address and they are differentiated by only  $\overline{RD}$  and  $\overline{WR}$  signals. (The processor can only write to control register and it can only read the status register.)

The processor can read the status register to check the status of the ports or the timer. The status register format is shown in Fig. 6.12.



**Fig. 6.12** : Format of an 8155 status register.

The 8155 ports can be configured as simple IO ports or handshake IO ports. In simple IO port no handshake signals are exchanged between the IO port and the IO device.

When a port is configured as a simple input port then the input device will load the data into the port without checking whether the previous data has been read by the processor or not.

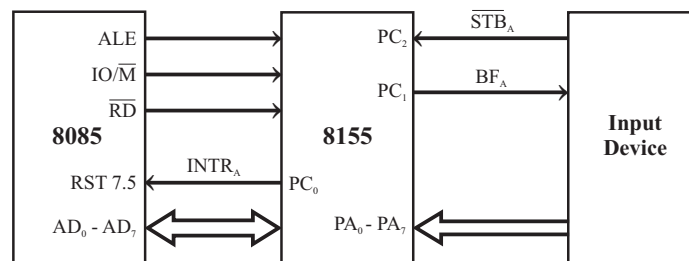
When a port is configured as a simple output port, then the processor loads the data to the output port without checking whether the previous data is accepted by the output device or not.

When a port is programmed as a handshake input port then handshake signals are used to transfer the data from input device to the port. When the port receives a data, it interrupts the processor for executing a subroutine for reading the data from the port and storing in appropriate place. Alternatively, the processor can check the status register of 8155 to know any data is available on the port, if a data is available on the port then the processor executes a subroutine to read the data and store it in appropriate place.

When a port is programmed as a handshake output port then the handshake signals are used to transfer the data from the port to output device. The processor first loads a data to the port. When the data is accepted by the output device the port will send an interrupt signal to the processor. Now the processor can load the next data to the port. Alternatively the processor can check the status register of 8155 before loading the next data.

### 8155 Handshake Input Port

The signals used for data transfer between input device and microprocessor using port-A of 8155 as handshake input port are shown in Fig. 6.13.



**Fig. 6.13 : Port-A of 8155 as handshake input port.**

1. The input device checks the **Buffer Full ( $BF_A$ )** signal. If  $BF_A$  is **low** then it places the data on port lines and asserts strobe ( $\overline{STB_A}$ ) **low**, to inform the port.
2. At the falling edge of  $\overline{STB_A}$ , the  $BF$  signal is asserted **high** to inform the input device that the port is full and it has to wait.
3. The input device asserts  $\overline{STB_A}$  signal as **high** after a predefined time and when it is asserted **high**, the 8155 generates an interrupt signal to the processor ( $INTR_A$ ).
4. On receiving an interrupt request the processor executes a subroutine to read the data from the port.
5. The data is read by the processor using  $\overline{RD}$  signal. At the rising edge of  $\overline{RD}$  the  $BF_A$  and  $INTR_A$  signal are asserted **low**, and now the input device can send the next data to the port.

*Note : Instead of interrupting the processor, the system can be designed to have data transfer by polling technique. In this method the processor polls the status register at regular intervals.*

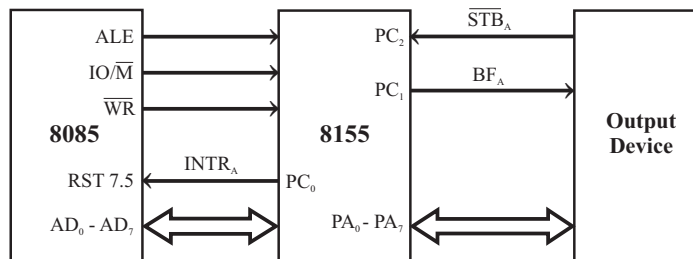
### 8155 Handshake Output Port

The signals used for data transfer between the output device and microprocessor using port-A of 8155 as handshake output port are shown in Fig. 6.14.

1. When the port is empty the processor writes a byte into the port.
2. For writing a data to the port, the processor asserts  $\overline{WR}$  signal as **low** and then **high**. At the falling edge of  $\overline{WR}$  the  $INTR_A$  is reset (asserted **low**) and at the rising edge of  $\overline{WR}$  the  $BF_A$  is asserted **high**.

3. The  $\overline{BF}_A$  signal informs the output device that the data is ready for it. If the output device accepts the data byte, then it asserts  $\overline{STB}_A$  **low** and then **high**.
4. When strobe is **low**, the  $\overline{BF}_A$  is reset to **low** and at the rising edge of the strobe the  $\overline{INTR}_A$  goes **high** to interrupt the processor.
5. When the processor is interrupted, it executes an interrupt service routine to load the next data in the output port.

*Note : The data transfer between the port and the processor can also be achieved by status check technique instead of using interrupt.*



**Fig. 6.14 : Port-A of 8155 as handshake output port.**

#### 6.2.4 Programmable Peripheral Interface - INTEL 8255

The INTEL 8255 is a device used to implement parallel data transfer between processor and slow peripheral devices like ADC, DAC, keyboard, 7-segment display, LCD, etc.

The 8255 has three ports: Port-A, Port-B and Port-C. The ports A and B are 8-bit parallel ports. Port-A can be programmed to work in any one of the three operating modes as input or output port. The three operating modes are :

- Mode-0 → Simple IO port.
- Mode-1 → Handshake IO port.
- Mode-2 → Bidirectional IO port.

Port-B can be programmed to work either in mode-0 or mode-1 as input or output port. Port-C pins (8 pins) have different assignments depending on the mode of ports A and B. If ports A and B are programmed in mode-0, then port-C can perform any one of the following function :

1. As 8-bit parallel port in mode-0 for input or output.
2. As two numbers of 4-bit parallel port in mode-0 for input or output.
3. The individual pins of port-C can be set or reset for various control applications.

If port-A is programmed in mode-1/mode-2 and port-2 is programmed in mode-1 then some of the pins of port-C are used for handshake signals and the remaining pins can be used as input/output lines or individually set/reset for control applications.

#### IO Modes of 8255

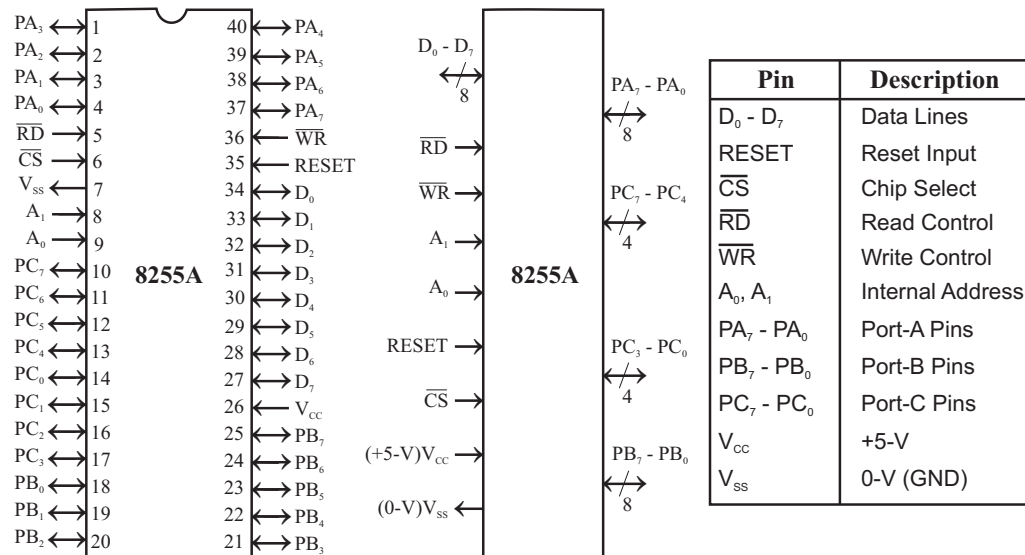
**Mode-0** : In this mode, all the three ports can be programmed either as input or output port. In mode-0, the outputs are latched and the inputs are not latched. The ports do not have handshake or interrupt capability. The ports in mode-0 can be used to interface DIP switches, Hexa-keypad, LEDs and 7-segment LEDs to the processor.

**Mode-1** : In this mode, only ports A and B can be programmed either as input or output port. In mode-1, handshake signals are exchanged between the processor and the peripherals prior to data transfer. The port-C pins are used for handshake signals. Input and output data are latched. Interrupt driven data transfer scheme is possible.

**Mode-2** : In this mode the port will be a bidirectional port (i.e., the processor can perform both read and write operations with an IO device connected to a port in mode-2). Only port-A can be programmed to work in mode-2. Five pins of port-C are used for handshake signals. This mode is used primarily in applications such as data transfer between two computers or floppy disk controller interface.

### Pins, Signals and Internal Block Diagram of 8255

The pin description of 8255 is shown in Fig. 6.15. It has 40 pins and requires a single +5-V supply. The internal block diagram of 8255 is shown in Fig. 6.16.



**Fig. 6.15** : Pin description of 8255.

The ports are grouped as Group A and Group B. The group A has port-A, port-C upper and its control circuit. The group B comprises of port-B, port-C lower and its control circuit. The read/write control logic requires six control signals. These signals are given below :

- $\overline{RD}$  (Read) : This control signal enables the read operation. When this signal is **low**, the microprocessor reads data from a selected IO port of the 8255A.
- $\overline{WR}$  (Write) : This control signal enables the write operation. When this signal goes **low**, the microprocessor writes into a selected IO port or the control register.

RESET : This is an active **high** signal. It clears the control register and set all ports in the input mode.

$\overline{CS}$ ,  $A_0$  and  $A_1$  : These are device select signals. The address lines  $A_0$  and  $A_1$  of 8255 can be connected to any two address lines of the processor to provide internal addresses. The  $A_0$  and  $A_1$  selects any one of the 4 internal devices as shown in Table-6.3. The 8255 will remain in **high impedance** state if the signal input to  $\overline{CS}$  is **high** and the device can be brought to normal logic by making the signal input to  $\overline{CS}$  as logic **low**.

TABLE - 6.3

Internal address		Device selected
$A_1$	$A_0$	
0	0	Port-A
0	1	Port-B
1	0	Port-C
1	1	Control Register

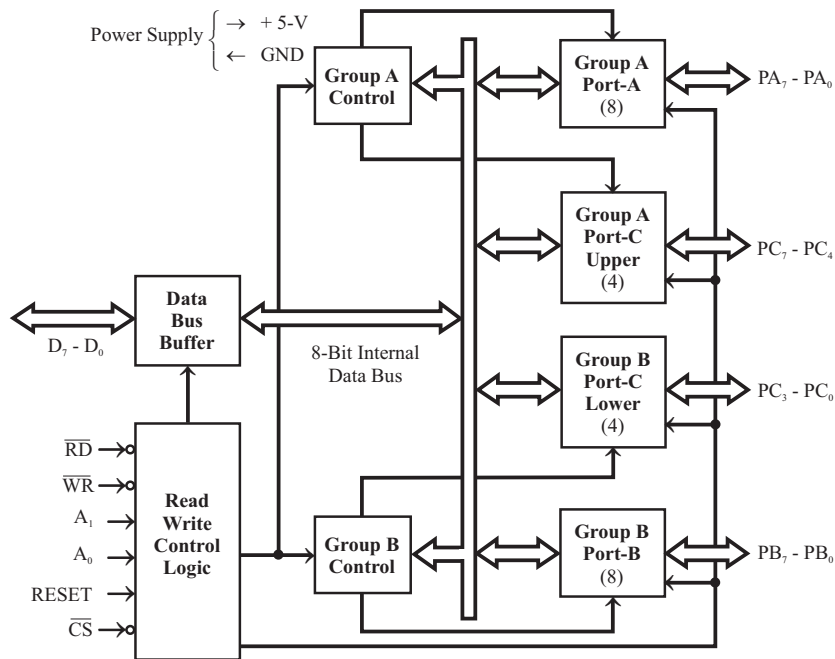


Fig. 6.16 : Internal block diagram of 8255.

### Interfacing of 8255 with 8085 Processor

A simple schematic for interfacing the 8255 with 8085 processor is shown in Fig. 6.17. The 8255 can be either memory-mapped or IO-mapped in the system. In the schematic shown in Fig. 6.17, the 8255 is IO-mapped in the system. The chip select signals for IO-mapped devices are generated by using a 3-to-8 decoder. The address lines  $A_4$ ,  $A_5$  and  $A_6$  are decoded to generate eight chip select signals (IOCS-0 to IOCS-7) and in this, the chip select IOCS-1 is used to select 8255. The address line  $A_7$  and the control signal  $\overline{IO/\overline{M}}$  are used as enable for the decoder.

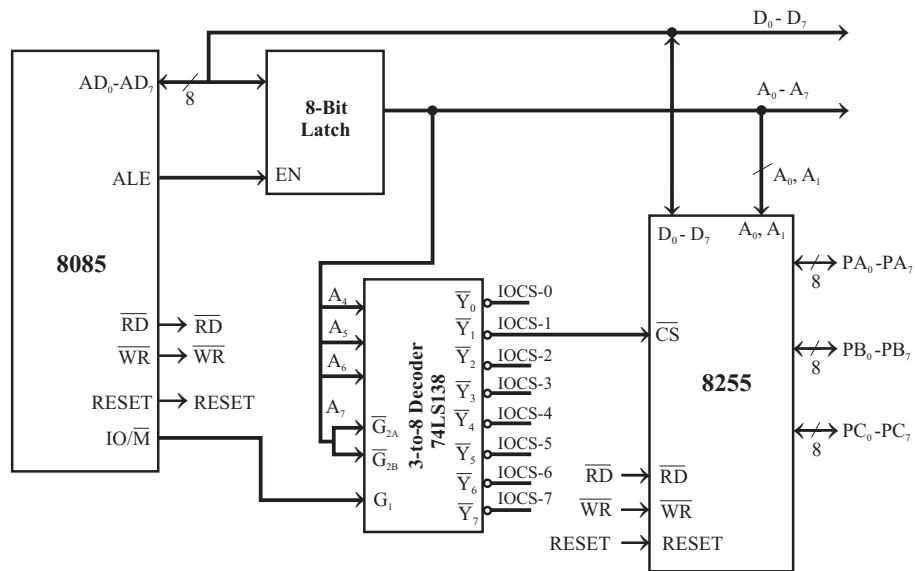


Fig. 6.17 : Interfacing 8255 with 8085 processor.

The address line  $A_0$  of 8085 is connected to  $A_0$  of 8255 and  $A_1$  of 8085 is connected to  $A_1$  of 8255 to provide the internal addresses. The IO addresses allotted to the internal devices of 8255 are listed in Table-6.4. The data lines  $D_0$ - $D_7$  are connected to  $D_0$ - $D_7$  of the processor to achieve parallel data transfer.

TABLE - 6.4 : IO ADDRESSES OF 8255

Internal device	Binary address								Hexa address
	Decoder input and enable				Input to address pins of 8255				
	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	
Port-A	0	0	0	1	x	x	0	0	10
Port-B	0	0	0	1	x	x	0	1	11
Port-C	0	0	0	1	x	x	1	0	12
Control Register	0	0	0	1	x	x	1	1	13

*Note : Don't care "x" is considered as zero.*

In the schematic shown in Fig. 6.17, the interrupt scheme is not included and so the data transfer can be performed only by checking the status of 8255 and not by interrupt method. For interrupt driven data transfer scheme, the interrupt controller 8259 has to be interfaced to system and the interrupts of port-A ( $PC_3$ ) and port-B ( $PC_0$ ) should be connected to two IR inputs of 8259.

### Programming (or Initializing) 8255

The 8255 has two control words: **IO Mode Set control Word (MSW)** and **Bit Set/Reset (BSR)** control word. The MSW is used to specify IO functions and BSR word is used to set/reset individual pins of port-C. Both the control words are written in the same control register. The control register differentiates them by the value of bit  $B_7$ . The BSR control word does not affect the functions of ports A and B.

Bit  $B_7$  of the control register specifies either the IO function or the bit set/reset function. If  $B_7 = 1$ , then the bits  $B_6$ - $B_0$  determine IO functions in various modes. If bit  $B_7 = 0$ , then the bits  $B_6$ - $B_0$  determine the pin of port-C to be set or reset.

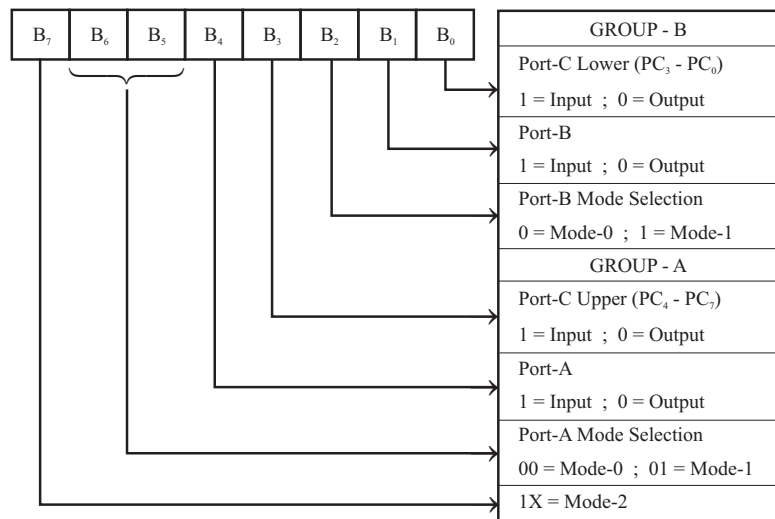


Fig. 6.18 : Format of IO mode set control word of 8255.

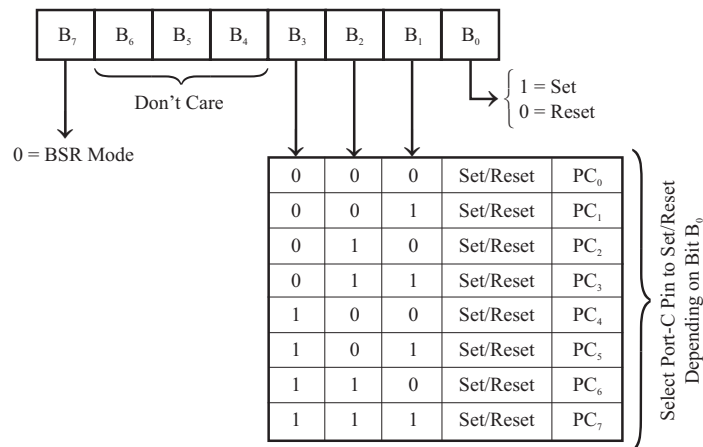


Fig. 6.19 : Format of Bit Set/Reset control word of 8255.



The 8255 ports are programmed (or initialized) by writing a control word in the control register. For setting IO functions and mode of operation the IO mode set control word is sent to control register. For setting/resetting a pin of port-C, the bit set/reset control word is sent to control register. The format of the IO mode set control word is shown in Fig. 6.18 and the format of bit set/reset control word is shown in Fig. 6.19. The various functions (assignments) of port-C pins during the different operating modes of ports A and B are listed in Table-6.5.

**TABLE - 6.5 : PORT-C PIN ASSIGNMENTS**

Functions of Ports A and B	PC <sub>7</sub>	PC <sub>6</sub>	PC <sub>5</sub>	PC <sub>4</sub>	PC <sub>3</sub>	PC <sub>2</sub>	PC <sub>1</sub>	PC <sub>0</sub>
Ports A and B in mode-0 Input/Output	IO	IO	IO	IO	IO	IO	IO	IO
Ports A and B in mode-1 Input ports	IO	IO	IBF <sub>A</sub>	$\overline{\text{STB}}_A$	INTR <sub>A</sub>	$\overline{\text{STB}}_B$	IBF <sub>B</sub>	INTR <sub>B</sub>
Ports A and B in mode-1 Output ports	$\overline{\text{OBF}}_A$	$\overline{\text{ACK}}_A$	IO	IO	INTR <sub>A</sub>	$\overline{\text{ACK}}_B$	$\overline{\text{OBF}}_B$	INTR <sub>B</sub>
Port-A in mode-2 Port-B in mode-0	$\overline{\text{OBF}}_B$	$\overline{\text{ACK}}_A$	IBF <sub>A</sub>	$\overline{\text{STB}}_A$	INTR <sub>A</sub>	IO	IO	IO

IO	- Input /Output line	$\overline{\text{OBF}}$	- Output Buffer Full
$\overline{\text{STB}}$	- Strobe	$\overline{\text{ACK}}$	- Acknowledge
IBF	- Input Buffer Full	The subscript A denotes port-A signal.	
INTR	- Interrupt Request	The subscript B denotes port-B signal.	

In handshake mode (i.e., in mode-1 and mode-2) the data transfer between the processor and the port can be implemented either by interrupt method or by checking the status of 8255 ports. In interrupt driven data transfer scheme when the port is ready it interrupts the 8085 processor through any one of the interrupt pin for a read/write operation. In status check technique, the 8085 processor can check the status of ports A and B by reading port-C. When the port is ready for data transfer, the processor executes a read/write cycle.

The 8255 has two internal flip-flops as interrupt enables (INTE<sub>A</sub> and INTE<sub>B</sub>) for port-A and port-B interrupt signals. In interrupt driven data transfer scheme the 8255 generates an interrupt signal only if these flip-flops are enabled by using BSR control word. The INTE<sub>A</sub> is enabled by setting PC<sub>4</sub> to **high** and INTE<sub>B</sub> is enabled by setting PC<sub>2</sub> to **high** using BSR control word. The interrupt signal can be disabled by resetting these two bits to zero using BSR control word.

When port-A and port-B are programmed in handshake mode (i.e., in mode-1 and mode-2), port-C can be read to know the readiness of the ports for data transfer. The format of the status word read from port-C is shown in Fig. 6.20.

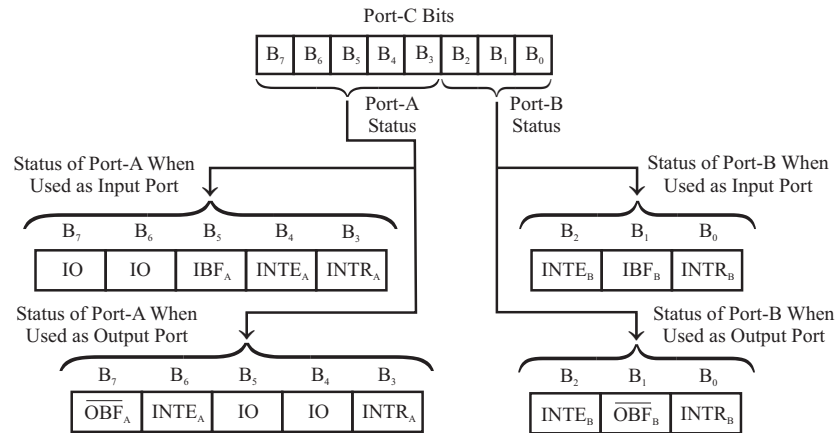


Fig. 6.20 : Format of status word of 8255 for handshake input and output operation.

### 8255 Handshake Input Port (Mode-1)

The signals used for data transfer between input device and 8085 microprocessor using port-A of 8255 as handshake input port (Mode-1) are shown in Fig. 6.21.

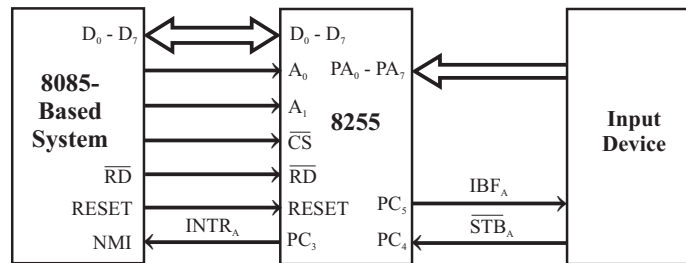


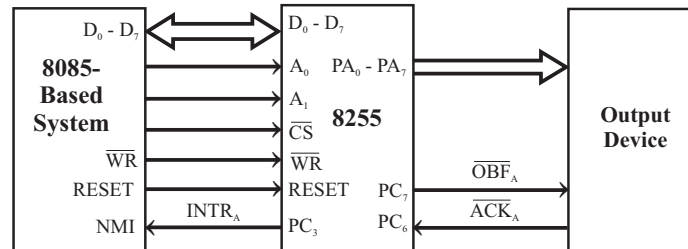
Fig. 6.21 : Port-A of 8255 as handshake input port (Mode-1).

1. The input device checks IBF<sub>A</sub> signal, if it is **low** then the input device places the data on the port lines PA<sub>0</sub>-PA<sub>7</sub> and asserts  $\overline{\text{STB}}_A$  **low** and after a delay time  $\overline{\text{STB}}_A$  is asserted **high**.
2. When  $\overline{\text{STB}}_A$  is **low** the 8255 asserts IBF signal **high** and at the rising edge of  $\overline{\text{STB}}_A$  the data is latched to the port and INTR<sub>A</sub> is set **high**.
3. When INTR<sub>A</sub> goes **high** the processor is interrupted through RST 5.5 input pin to execute a subroutine for reading the data from the port. For a read operation, the processor asserts  $\overline{\text{RD}}$  **low** and then **high**.
4. When  $\overline{\text{RD}}$  is **low**, INTR<sub>A</sub> is reset (asserted **low**) by 8255 and at the rising edge of  $\overline{\text{RD}}$ , IBF is asserted **low** and the input device can send the next data.

*Note : For port-B as input port in mode-1, same operations are performed, but for handshake signals PC<sub>0</sub>, PC<sub>1</sub> and PC<sub>2</sub> are used.*

### 8255 Handshake Output Port (Mode-1)

The signals used for data transfer between output device and 8085 microprocessor using port-A of 8255 as handshake output port (Mode-1) are shown in Fig. 6.22.



**Fig. 6.22 :** Port-A of 8255 as handshake output port (Mode-1).

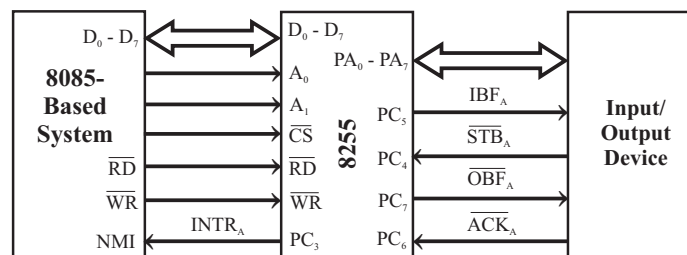
1. When the port is empty, the processor writes a byte in the port.
2. For writing a data to port, the processor asserts  $\overline{WR}$  **low** and then **high**. At the rising edge of  $\overline{WR}$ , both the  $INTR_A$  and  $\overline{OBF}_A$  are asserted **low** by the 8255.
3. The  $\overline{OBF}_A$  signal informs the output device that the data is ready. If the output device accepts the data then it sends an acknowledge signal by asserting  $\overline{ACK}_A$  **low** and then **high**.
4. When  $\overline{ACK}_A$  is **low**, the  $\overline{OBF}_A$  is asserted **high** by the 8255. When  $\overline{ACK}_A$  is **high** the  $INTR_A$  is set (asserted **high**), to interrupt the processor.
5. When  $INTR_A$  goes **high**, the processor is interrupted through RST 5.5 input pin to execute an interrupt service routine to load next data in the output port.

*Note : For port-B as output port in mode-1, same operations are performed, but for handshake signals  $PC_0$ ,  $PC_1$  and  $PC_2$  are used.*

### 8255 Bidirectional Port (Mode-2)

The signals used for data transfer between the IO device and 8085 microprocessor using port-A of 8255 as bidirectional port (Mode-2) are shown in Fig. 6.23.

*Note : Only port-A can work in mode-2.*



**Fig. 6.23 :** Port-A of 8255 as bidirectional port (Mode-2).

In mode-2 the port can be used either as an input port or as an output port. At any one time the processor will perform either read or write operation. In mode-2 the read operation can be followed by write or write operation can be followed by read. The signals involved and the operations performed for read operation are similar to mode-1 input port. The signals involved and the operations performed for write operation are similar to mode-1 output port.

### 6.2.5 Programmable IO Port and Memory - INTEL 8355

The INTEL 8355 is a ROM and IO port chip that can be used in the 8085A and 8088 microprocessor systems. The ROM portion has 2048 (2 k) locations with a word size of 8 bits. It has a maximum access time of 400 ns so that the device can be used without wait states in the 8085A CPU. The 8355-2 has 300 ns access time for compatibility with the 8085A-2 and full speed 5 MHz 8088 microprocessors. The internal block diagram and the pin description of 8355 are shown in Fig. 6.24.

The 8355 is a 40-pin IC available in DIP. It has multiplexed address and data lines. It has an internal address latch to demultiplex the address and data lines using the signal ALE. The IO portion consists of two general purpose IO ports. Each IO port has eight port lines and each IO port line is individually programmable as input or output.

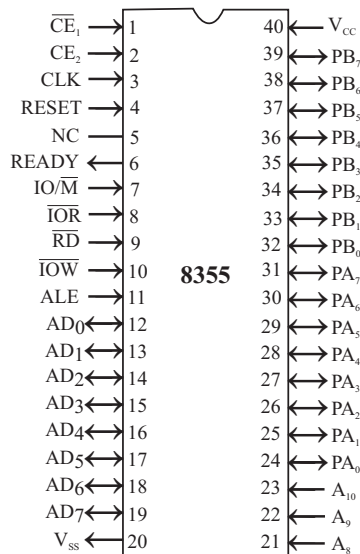


Fig. a : Pin description.

Fig. 6.24 : Pin description and internal block diagram of 8355.

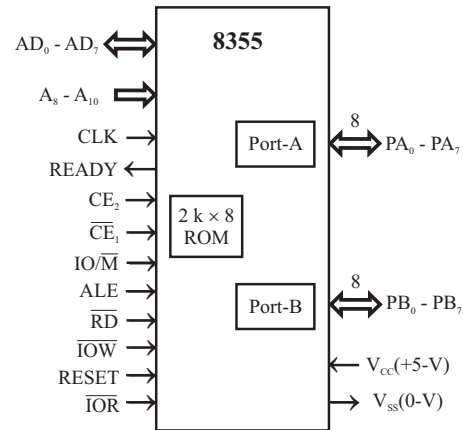


Fig. b : Internal block diagram of 8355.

TABLE - 6.6

Internal address		Device selected
A <sub>1</sub>	A <sub>0</sub>	
0	0	Port-A
0	1	Port-B
1	0	DDRA
1	1	DDR B

Pin	Description
AD <sub>0</sub> -AD <sub>7</sub>	Bidirectional address/data
A <sub>8</sub> -A <sub>10</sub>	High order bits of ROM address
ALE	Address latch enable
CE <sub>1</sub>	Active <b>low</b> chip enable
CE <sub>2</sub>	Active <b>high</b> chip enable
IO/M	IO or memory select
RD	Memory read control
IOW	IO write control
IOR	IO read control
RESET	RESET input
READY	Wait state request
CLK	Clock input
V <sub>cc</sub>	Power supply (+5-V)
V <sub>ss</sub>	Ground (0-V)
PA <sub>0</sub> -PA <sub>7</sub>	Port-A IO lines
PB <sub>0</sub> -PB <sub>7</sub>	Port-B IO lines

The ports are programmed as input or output port by loading an 8-bit word in the **Data Direction Register (DDR)** of the concerned port. Each line of the port can be individually used as input or output line. A **zero** loaded in the DDR register makes the corresponding line of port as input line. A **one** loaded in the DDR register makes the corresponding line of port as output line.

The ports and data direction registers are selected by two bit internal address,  $A_0$  and  $A_1$  as shown in Table-6.6. When the device is reset by applying a **high** signal at RESET pin, the DDRs are cleared and the ports are initialized to input mode.

### 6.2.6 Programmable IO Port and Memory - INTEL 8755

The INTEL 8755 is an EPROM and IO chip that can be used in the 8085A and 8088 microprocessor systems. The EPROM portion has 2048 (2 k) locations with a word size of 8 bits. It has a maximum access time of 450 ns so that the device can be used without wait states in an 8085A CPU.

The pin description and internal block diagram of 8755 are shown in Fig. 6.25. The 8755 is a 40-pin IC available in DIP. It has multiplexed address and data lines. It has an internal latch to demultiplex the address and data lines using the signal ALE. The IO portion consists of two general purpose IO ports. Each IO port has eight port lines and each IO port line is individually programmable as input or output.

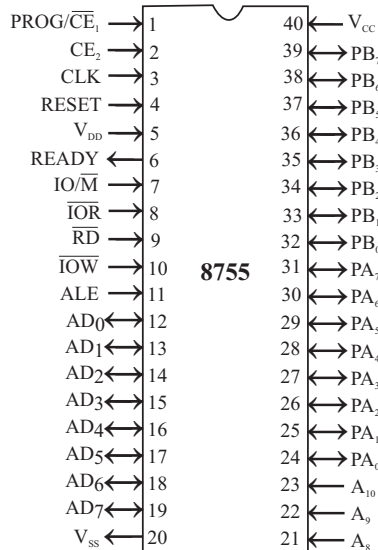


Fig. a : Pin description.

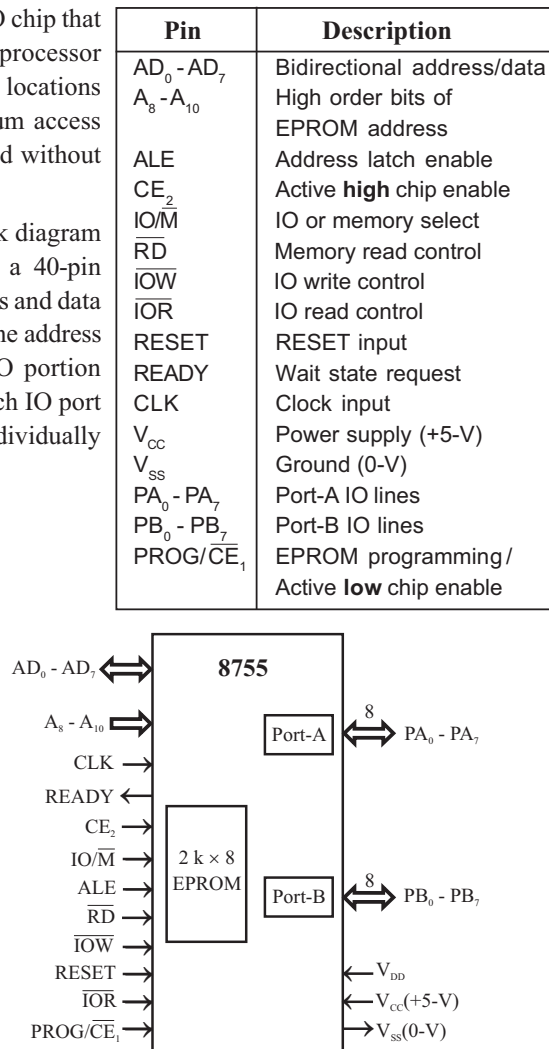


Fig. b : Internal block diagram of 8755.

Fig. 6.25 : Pin description and block diagram of 8755.

The ports are programmed as input or output port by loading an 8-bit word in the **Data Direction Register (DDR)** of the concerned port. Each line of the port can be individually used as input or output lines. A **zero** loaded in the DDR register makes the corresponding line of the port as input line. A **one** loaded in the DDR register makes the corresponding line of the port as output line.

The ports and data direction registers are selected by two bit internal address,  $A_0$  and  $A_1$  as shown in Table-6.7. When the device is reset by applying a **high** signal at RESET pin, the DDRs are cleared and the ports are initialized in the input mode. The pin  $\text{PROG}/\overline{\text{CE}}_1$  is used to program the EPROM.

**TABLE - 6.7**

Internal address		Device selected
$A_1$	$A_0$	
0	0	Port-A
0	1	Port-B
1	0	DDRA
1	1	DDR B

### 6.2.7 DMA Data Transfer Scheme

Normally the data transfer from memory to IO device or IO device to memory can be achieved only through microprocessor. When data has to be transferred from memory to IO device, first the processor sends address and control signals to memory to read the data from memory. Then the processor sends address and control signals to IO device to write data to IO device.

Similarly, when the data has to be transferred from IO device to memory, first the processor sends address and control signals to IO device to read data from IO device. Then the processor sends address and control signals to memory device to write data to memory.

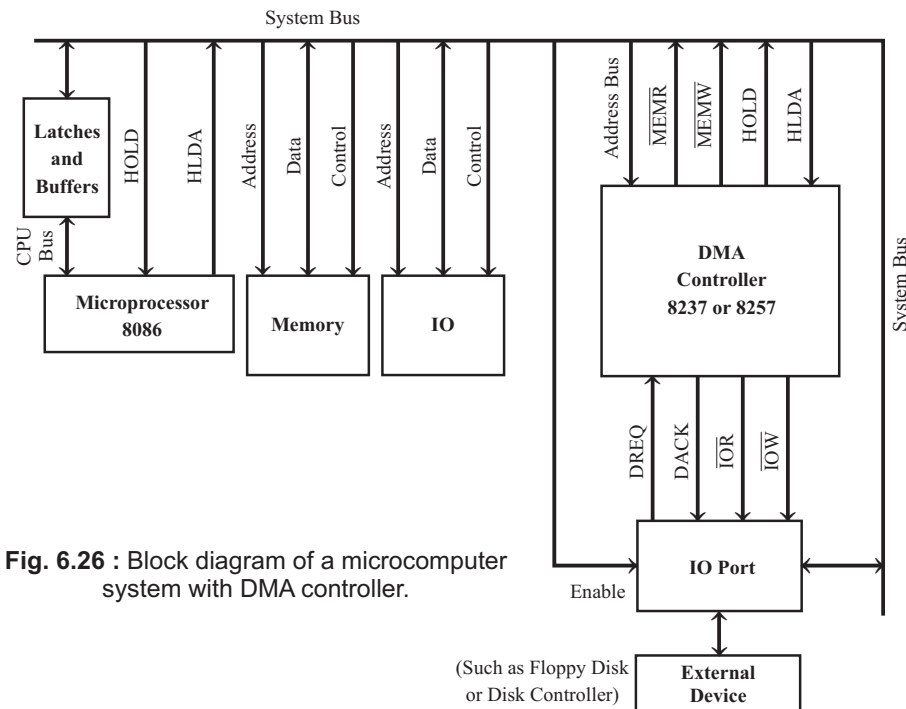
In the data transfer method described above, data cannot be directly transferred between memory and IO device, even though they are connected to common bus. The above process is inevitable, because the processor cannot simultaneously select two devices. Hence, a scheme called **Direct Memory Access (DMA)** has been developed in which the IO device can access the memory directly for data transfer. The DMA data transfer will be useful to transfer large amount of data between memory and IO device in a short time.

For direct data transfer between IO device and memory a dedicated hardware device called **Direct Memory Access controller (DMA controller)** is used. A DMA controller temporarily borrows the address bus, data bus and control bus from the microprocessor, and transfers the data bytes directly from the IO ports to a series of memory locations or vice versa. Some DMA controllers can also perform memory-to-memory transfer.

### A Microcomputer System with a DMA Controller

The simplified diagram of a microcomputer system with a DMA controller is shown in Fig. 6.26. In the system shown in Fig. 6.26 the DMA controller has one channel, which serves for one IO device. In actual DMA controller we may have more than one channel and each channel may service an IO device independently. Each channel contains an address register, a control register and a count register. For the sake of simplicity let us consider one channel DMA controller.

The DMA controller can work as a slave or as a master. In the slave mode, the microprocessor loads the address register with starting address of the memory, loads the count register with number of bytes to be transferred and loads the control register with control information.



**Fig. 6.26 :** Block diagram of a microcomputer system with DMA controller.

For performing DMA operation the processor has to initialize or program the IO device and DMA controller. Consider an example of transferring bulk data from floppy to memory by DMA. In this case the processor initializes both DMA controller and floppy controller, so that DMA controller is informed about address, type of DMA and number of bytes to be transferred and the floppy controller is informed to go for a DMA.

When the IO device needs a DMA transfer it sends a **DMA request** signal (DREQ) to the DMA controller. When the DMA controller receives a DMA request, it sends a HOLD request to the processor. At the end of the current instruction execution, the processor relieves the bus by asserting all its data, address and control pins to **high impedance** state. Then the processor sends an acknowledge (HLDA) signal to the DMA controller.

When the controller receives an acknowledge signal it takes control of the system bus and begins to work as a master. The DMA controller sends a **DMA acknowledge** signal (DACK) to IO device. The DACK signal will inform the device to get ready for DMA transfer.

For a read operation, the DMA controller outputs the memory address on the address bus and asserts  $\overline{\text{MEMR}}$  and  $\overline{\text{IOW}}$  signals. The DMA read refers to read data from memory. Hence for a read operation, the memory outputs the data on the data bus and this data will be written into IO port.

For a write operation, the DMA controller outputs the memory address on the address bus and asserts  $\overline{\text{MEMW}}$  and  $\overline{\text{IOR}}$  signals. The DMA write refers to write data to memory. Hence for a write operation, the IO device outputs the data on the data bus and this data will be written into memory. When the data transfer is complete the DMA controller unasserts its HOLD request signal to the processor and the processor takes control of the system bus.

The DMA transfer may be performed to transfer a byte at a time or in blocks. In cycle stealing DMA or single transfer mode, the DMA controller will perform one byte transfer in between instruction cycles. In burst mode or block transfer mode, the DMA controller will transfer a block of data.

### 6.2.8 DMA Controller - INTEL 8237

The DMA controller-8237 has been developed for 8085/8086/8088 microprocessor-based system. It is a device dedicated to perform a high speed data transfer between the memory and the IO device. The 8237 has four channels. So, it can be used to provide DMA to four IO devices. When more than four devices require DMA, a number of 8237 can be connected in cascade to increase the DMA channels.

For each DMA channel, a set of registers has been dedicated to store the memory address and the count value for number of bytes to be read/write by DMA. These registers are base address, current address, base word count, current word count and mode registers. Apart from these dedicated registers, the 8237 has temporary registers, status, command, mask and request registers.

The 8237 is a 40-pin IC and is available in a **Dual In-line Package (DIP)**. The pin configuration of 8237 is shown in Fig. 6.27. A brief description about the pins and signals of 8237 are listed in Table-6.8. The functional block diagram of 8237 is shown in Fig. 6.28.

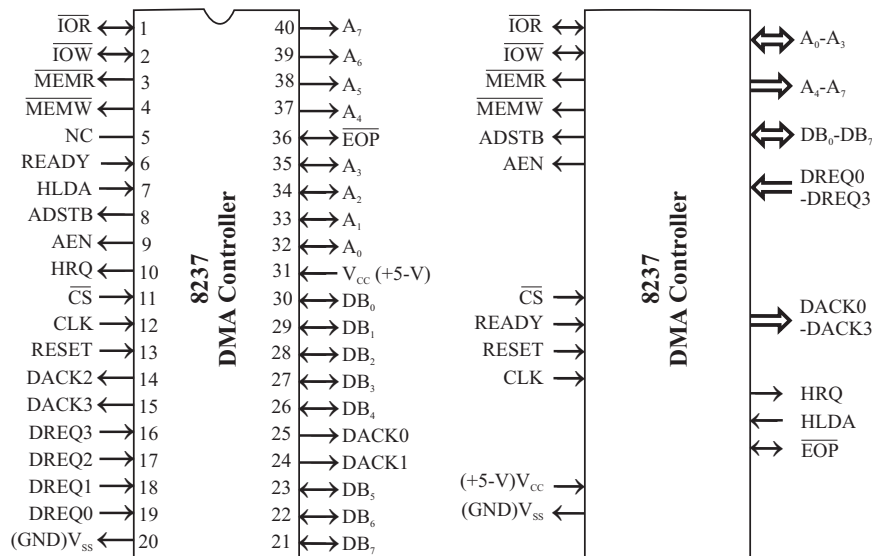


Fig. 6.27 : Pin configuration of 8237.

#### Features of 8237

- It has four independent DMA channels to service four IO devices.
- Number of channels can be increased by cascading any number of 8237.
- Each channel can be independently programmable to transfer upto 64 kb of data by DMA.
- Each channel can independently perform read transfer, write transfer and verify transfer.
- Channel-0 and channel-1 are used to perform memory-to-memory transfer.
- Each channel can be independently programmable to perform demand transfer DMA, single transfer DMA and block transfer DMA.



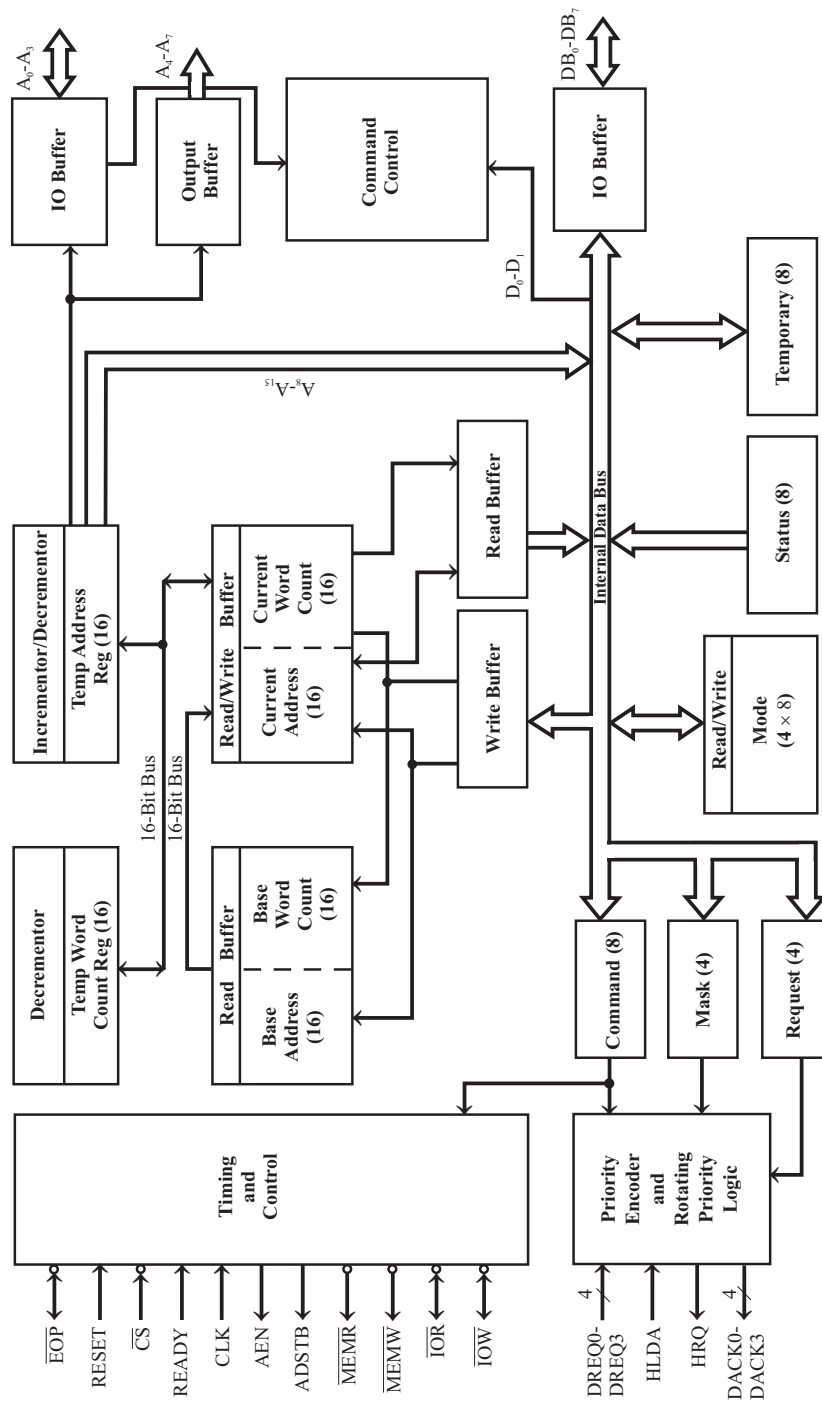


Fig. 6.28 : Functional block diagram of an 8237.

**TABLE - 6.8 : PIN DESCRIPTION OF 8237**

Pin	Description
CLK	Clock input to 8237. Maximum clock frequency is 5 MHz. In 8085 system, the processor clock is inverted and applied to CLK of 8237.
$\overline{\text{CS}}$	Logic <b>low</b> chip select signal. It is input signal to select 8237 during programming mode.
RESET	Reset input to 8237. Connected to system reset, when the RESET signal goes <b>high</b> the command, status, request and temporary registers are cleared. It also clears first-last flip-flop and sets the mask register.
READY	Ready input signal and it is tied to $V_{cc}$ for normal timings. When READY input is tied <b>low</b> , the 8237 enters a wait state. This is used to get extra time in DMA machine cycles to transfer data between slow memory and IO devices.
HRQ	Hold request output signal. It is the hold request signal sent by 8237 to the processor HOLD pin, to make a request for bus to perform DMA transfer.
HLDA	Hold acknowledge input signal. It is the hold acknowledge signal to be sent by the processor to inform the acceptance of hold request.
DREQ3 to DREQ0	DMA request inputs (Four channel inputs). Used by IO devices to request for DMA transfer.
DACK3 to DACK0	DMA acknowledge output signals. These are output signals from 8237 to the IO devices to inform the acceptance of DMA request. These outputs are programmable as either active <b>high</b> or active <b>low</b> signals.
$\text{DB}_7$ to $\text{DB}_0$	Data bus lines. These pins are used for data transfer between processor and DMA controller during programming mode. During DMA mode, these lines are used as multiplexed high order address and data lines.
$\overline{\text{IOR}}$	Bidirectional IO read control signal. It is input control signal for reading DMA controller during programming mode and output control signal for reading IO device during DMA (memory) write cycle.
$\overline{\text{IOW}}$	Bidirectional IO write control signal. It is input control signal for writing DMA controller during programming mode and output control signal for writing IO device during DMA (memory) read cycle.
$\overline{\text{EOP}}$	End of process. It is a bidirectional low active signal. It is used either as an input to terminate a DMA process or as an output to inform the end of the DMA transfer to the processor. This output can be used as interrupt to terminate DMA.

Table - 6.8 continued...

Pin	Description
A <sub>3</sub> to A <sub>0</sub>	Four bidirectional address lines. Used as input address during programming mode to select internal registers. During DMA mode the low order four bits of memory address are output by 8237 on these lines.
A <sub>7</sub> to A <sub>4</sub>	Four unidirectional address lines. Used to output the memory address bits A <sub>7</sub> to A <sub>4</sub> during DMA mode.
AEN	Address enable output signal. It is used to enable the address latch connected to DB <sub>7</sub> - DB <sub>0</sub> pins of 8237. It is also used to disable any buffers in the system connected to the processor.
ADSTB	Address strobe output signal. It is used to latch the high byte memory address issued through DB <sub>7</sub> to DB <sub>0</sub> lines by 8237 during DMA mode into an external latch.
$\overline{\text{MEMR}}$	Memory read control signal. It is an output control signal issued during DMA read operation.
$\overline{\text{MEMW}}$	Memory write control signal. It is an output control signal issued during DMA write operation.

The various internal registers of 8237 are listed in Table-6.9. The processor can read or write into these registers. But with certain registers the processor can perform only the read operation and with certain registers the processor can perform only the write operation. The internal registers are selected by a 4-bit address supplied through A<sub>0</sub>-A<sub>3</sub> lines of 8237. The addresses of internal registers and the operations (read/write) that can be performed on these registers are listed in Table-6.10.

**TABLE - 6.9 : INTERNAL REGISTERS OF 8237**

Name of the register	Size of register in bits	Number of registers available
Base address register	16	4
Base word count register	16	4
Current address register	16	4
Current word count register	16	4
Temporary address register	16	1
Temporary word count register	16	1
Status register	8	1
Command register	8	1
Temporary register	8	1
Mode register	8	4
Mask register	4	1
Request register	3	1

**TABLE - 6.10 : ADDRESS OF INTERNAL REGISTERS OF 8237**

Name of the register	Operation performed	Binary address			
		A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>
Channel-0 Base and Current address	Write	0	0	0	0
Channel-0 Current address	Read	0	0	0	0
Channel-0 Base and Current word count	Write	0	0	0	1
Channel-0 Current word count	Read	0	0	0	1
Channel-1 Base and Current address	Write	0	0	1	0
Channel-1 Current address	Read	0	0	1	0
Channel-1 Base and Current word count	Write	0	0	1	1
Channel-1 Current word count	Read	0	0	1	1
Channel-2 Base and Current address	Write	0	1	0	0
Channel-2 Current address	Read	0	1	0	0
Channel-2 Base and Current word count	Write	0	1	0	1
Channel-2 Current word count	Read	0	1	0	1
Channel-3 Base and Current address	Write	0	1	1	0
Channel-3 Current address	Read	0	1	1	0
Channel-3 Base and Current word count	Write	0	1	1	1
Channel-3 Current word count	Read	0	1	1	1
Command register	Write	1	0	0	0
Status register	Read	1	0	0	0
Request register	Write	1	0	0	1
Write single mask register bit	Write	1	0	1	0
Mode register	Write	1	0	1	1
Clear byte pointer flip-flop	Write	1	1	0	0
Temporary register	Read	1	1	0	1
Master clear	Write	1	1	0	1
Clear mask register	Write	1	1	1	0
Write all mask register bits	Write	1	1	1	1

The 16-bit internal registers of 8237 are read/write through 8-bit data bus. The 8237 has an internal first-last flip-flop which has to be cleared to zero for reading/writing low byte first and then high byte. The first last flip-flop can be set to one for reading/writing high byte first and then low byte. (However the 8237 does not have facility to directly set the first-last flip-flop, but it has the facility to directly clear the first-last flip-flop.) After each read or write operation, the state of flip-flop automatically toggles.

### **Internal Registers Of 8237**

#### **Current address (CA) register**

It is used to hold the 16-bit memory address of the next memory location to be accessed by DMA. The 8237 outputs the content of the CA-register as memory address and increments/decrements it by one. Each channel has its own CA-register. Initially the starting address of memory is loaded in CA-register from the base address register.

#### **Current word count (CWC) register**

It holds the count value of the number of bytes to be transferred by DMA. Initially the count value is loaded to the CWC register from the base count register. After each byte transfer by DMA, the count value is decremented by one. Therefore at any one time it holds the count value for the number of bytes (pending) to be transferred by the DMA.

#### **Base address (BA) register**

It is used to hold the starting address of the memory block to be accessed by the DMA. During the start of the DMA process the content of the BA-register is loaded in the CA-register. If autoinitialization is enabled in mode register then the content of BA-register is reloaded in the CA-register at the end of DMA process.

#### **Base word count (BWC) register**

It is used to hold the count value for the number of bytes to be transferred by the DMA. During the start of DMA process the content of BWC register is loaded in CWC register. If autoinitialization is enabled in mode register then the content of BWC register is reloaded in CWC register at the end of the DMA process.

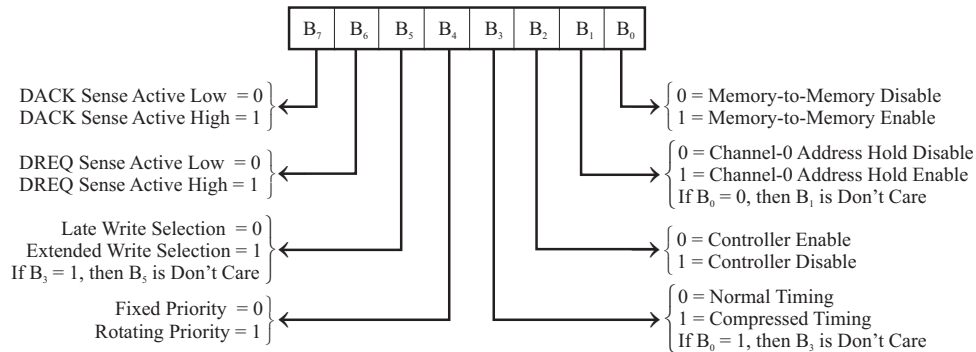
#### **Command register**

The command register is used to program the following features of 8237:

- Enable/Disable memory-to-memory transfer.
- Enable/Disable the DMA controller.
- Normal/Compressed timing.
- Fixed/Rotating priority.
- Type of (active **low/high**) DMA request and acknowledge signal.

The format of the control word to be loaded in the command register to program the above features is shown in Fig. 6.29. During memory-to-memory DMA transfer, the channel-0 registers are used to hold source address and the channel-1 registers are used to hold destination address. Data transfer takes place via the temporary register in 8237. The number of bytes transferred is determined by the channel-1 count register.

The bit  $B_2$  is used to turn ON/OFF the entire controller by software. The bit  $B_3$  is used to program the normal/compressed timing. In normal timing, the time taken to perform one DMA transfer will be four clock periods. In compressed timing, the time taken to perform one DMA transfer will be two clock periods.



**Fig. 6.29 :** Format of control word to be loaded in command register.

The bit B<sub>4</sub> is used to select fixed/rotating priority for DMA channels. In fixed priority channel-0 has the highest priority and channel-3 has the lowest priority. In rotating priority, after servicing a channel its priority is made as lowest. For example, if DMA request is made to channel-2 and there is no DMA request in other channels. Now after servicing channel-2 in rotating priority scheme the priorities of the channels from highest to lowest will be channel-0, channel-1, channel-3 and channel-2. Alternately if 8237 is programmed for fixed priority, then for the same situation after servicing the channel-2, the priorities of DMA channels from highest to lowest will be channel-0, channel-1, channel-2 and channel-3.

The bit B<sub>5</sub> is used to extend the timing of write pulse when the IO devices require wider write pulse. This is possible only in normal timing. The bit B<sub>6</sub> and B<sub>7</sub> are used to program the polarities (logic **low/high**) of the DMA request input and DMA acknowledge output.

### Mode register

Each channel has its own mode register and it is used to program the following features of each channel of 8237:

- Read/Write/Verify transfer.
- Demand/Single/Block transfer mode.
- Single/Cascaded operation of 8237.
- Enable/Disable autoinitialization.

The format of control word to be loaded in mode register is shown in Fig. 6.30. The control word of all the four mode registers are sent to same internal address, but the 8237 identifies the control word of a channel from the bits B<sub>0</sub> and B<sub>1</sub>. The bits B<sub>2</sub> and B<sub>3</sub> are used to program the read/write/verify transfer. In read transfer the data is transferred from memory to IO device. In write transfer the data is transferred from IO device to memory. Verification operations generate the DMA addresses without generating the DMA memory and IO control signals.

The bit B<sub>4</sub> is used to enable/disable autoinitialization of DMA channels. When it is enabled, the memory address and count value from base registers are loaded in current registers after completion of DMA process, which are used to repeat the DMA process between IO device and same block of memory.

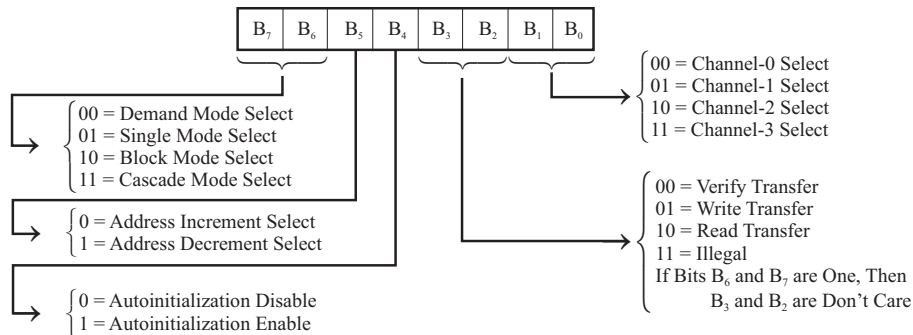


Fig. 6.30 : Format of control word to be loaded in the mode register.

The bits  $B_6$  and  $B_7$  are used to program various modes of operation like demand transfer mode, single transfer mode, block transfer mode and cascade mode. In demand transfer mode, the DMA transfer is performed until an external signal is applied to  $\overline{EOP}$  pin of 8237 or until the DREQ input becomes inactive.

In single transfer mode, the 8237 releases the bus to processor by deactivating the HOLD signal after transfer of each byte by DMA. If the DREQ pin is held active, then 8237 will make a request for DMA to the processor through HOLD pin again after a small delay. This will allow the processor to execute one instruction and the 8237 to perform one DMA transfer alternatively.

In block transfer mode, the 8237 will transfer an entire block of data specified by count register and then release the bus to processor by deactivating HOLD signal. In a cascaded operation, the hold request pin (HRQ) of one 8237 will be connected to HOLD pin of the processor and to each DREQ pin of this 8237, the HRQ pin of another 8237 can be connected. This connection can be extended until we get the required number of DMA channels.

When DMA request is made to a channel by another 8237, this channel cannot perform read/write/verify transfer.

### Request register

It is used to request a DMA transfer via software. The format of control word to be loaded in request register is shown in Fig. 6.31. The bit  $B_0$  and  $B_1$  select the channel in which DMA transfer is required and the bit  $B_2$  is used to set/reset DMA request.

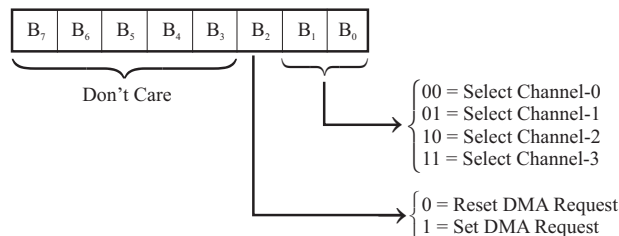
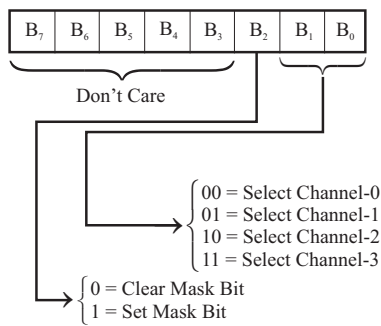


Fig. 6.31 : Format of control word to be loaded in a request register.

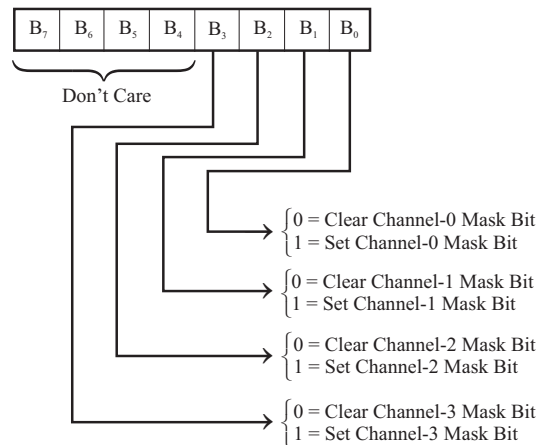
### Mask register

This register is used to mask (or disallow) the DMA request made through channels and to unmask (or enable) the DMA request made through channels. Please remember that, after a RESET all the channels are masked and so after a RESET the channels has to be unmasked by sending a control word to mask register.

The mask register has two internal address. One address is used to set/reset single mask bit (i.e., to mask/unmask one channel at a time) and another address is used to set/reset all the mask bits (i.e., to mask/unmask all the channels). The format of two control words for mask register are shown in Fig. 6.32.



**Fig. a :** Format of the control word to mask/unmask one channel.

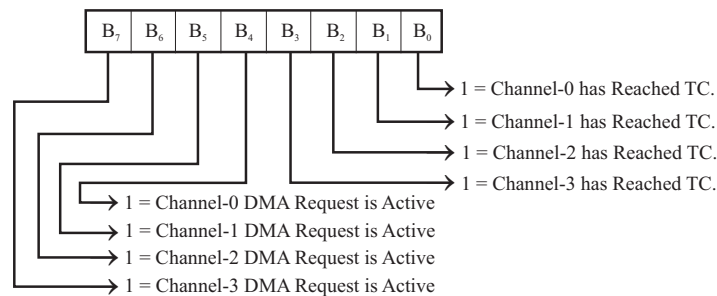


**Fig. b :** Format of the control word to mask/unmask all channel.

**Fig. 6.32 :** Format of the control word to be loaded in the mask register.

### Status register

The status register can be read to know whether the channels have reached their Terminal Count (TC) or not and also to know whether the DMA request on the DREQ pins are active or not. The format of status register is shown in Fig. 6.33.



**Fig. 6.33 :** Format of a status register.



### **Software Commands of 8237**

The 8237 has three software commands to control its operation and they are Clear first-last flip-flop, Master clear and Clear mask register. These software commands can be enabled by executing a write operation to the internal address allotted to these commands. (Please refer to Table-6.10 for internal addresses of these commands.) We need not worry about the data sent to these ports during the write operation because the 8237 will ignore the data. The functions of the software commands are given below.

#### **Clear first-last flip-flop**

This command resets the first-last flip-flop in 8237 to zero. The first-last flip-flop selects the low or high byte during read/write operation of address and count registers of the channels. If first-last flip-flop is zero (i.e., reset) then the low byte can be read/write. If it is one (i.e., set) then the high byte can be read/write. After every read/write operation the first-last flip-flop automatically toggles.

#### **Master clear**

This command is used as software RESET. The functions performed by this command is same as that of hardware RESET. During RESET all internal registers and first-last flip-flop are cleared and all the mask bits of the channels are set.

#### **Clear mask register**

This command is used to clear the mask bits of the DMA channels in order to enable all the four DMA channels.

### **Programming 8237**

The 8237 can work as a slave or as a temporary master in a microprocessor system. Normally the 8237 is interfaced to a system as a slave device. During the DMA operation it works as a temporary master. For proper DMA operation the 8237 has to be programmed, when it is working as a slave. The programming of 8237 refers to sending software commands and various control words to 8237, in order to inform the types of DMA, memory address, count value, etc., for each channel. At the start of programming all the DMA channels have to be disabled and then they are enabled at the end of programming. Also, the first-last flip-flop has to be cleared before sending 16-bit address/count value to 8237 in order to load low byte first and then high byte in address/count registers. The various steps in programming 8237 are given below:

1. First send a "master clear" software command to 8237, which mask/disable all DMA channels, clear first-last flip-flop and clear all internal register, except mask register.
2. Send a control word to command register to inform priority of DMA channels, normal/ compressed timings, polarity of DREQ and polarity of DACK signals.
3. Write a mode word to mode register of each channel to inform DMA mode and type of DMA transfer.
4. Send a "clear first-last flip-flop" software command to reset it to zero.

5. After ensuring that first-last flip-flop is zero, write the 16-bit address in the address register of each channel, by sending the low byte first and then the high byte.
6. Then write the 16-bit count value in the count register of each channel, by sending low byte first and then high byte. It is sufficient, if the first-last flip-flop is cleared at the beginning of sending a series of 16-bit address/count value, because after each write operation it automatically toggles to keep track of low byte and high byte.
7. Finally send "clear mask register" software command to enable all DMA channels. Now 8237 is ready to perform DMA process.

### Interfacing 8237 with 8085 Processor

A simple schematic for interfacing the 8237 with 8085 processor is shown in Fig. 6.34. The 8237 can be either memory-mapped or IO-mapped in the system. In the schematic shown in Fig. 6.34, the 8237 is IO-mapped in the system. The chip select signals for IO-mapped devices are generated by using a 3-to-8 decoder. The address lines  $A_4$ ,  $A_5$  and  $A_6$  are decoded to generate eight chip select signals (IOCS-0 to IOCS-7) and in this, the chip select signal IOCS-6 is used to select 8237. The address line  $A_7$  and the control signal  $\overline{IO/\overline{M}}$  are used as enable for decoder. The IO addresses of the internal register of 8237 are listed in Table-6.11.

The  $DB_0$ - $DB_7$  lines of 8237 are connected to data bus lines  $D_0$ - $D_7$  for data transfer with processor during programming mode. These lines ( $DB_0$ - $DB_7$ ) are also used by 8237 to supply the memory address  $A_8$ - $A_{15}$  during the DMA mode. The 8237 also supplies two control signals ADSTB and AEN to latch the address supplied by it during DMA mode on external latches. In the schematic shown in Fig. 6.34, two 8-bit latches are provided to hold the 16-bit memory address during DMA mode. During DMA mode, the AEN signal is also used to disable the buffers and latches used for address, data and control signals of the processor.

The 8237 provides separate read and write control signals for memory and IO devices during DMA. Therefore, the  $\overline{RD}$ ,  $\overline{WR}$  and  $\overline{IO/\overline{M}}$  of the 8085 processor are decoded by a suitable logic circuit to generate separate read and write control signals for memory and IO devices. (Please refer to Chapter-3, Fig. 3.18 for the logic circuit to generate separate read and write signals for memory and IO devices.)

The output clock of the 8085 processor should be inverted and supplied to the 8237 clock input for proper operation. The HRQ output of the 8237 is connected to HOLD input of the 8085 in order to make a HOLD request to the processor. The HLDA output of 8085 is connected to HLDA input of 8237, in order to receive the acknowledge signal from the processor once the HOLD request is accepted. The RESET OUT of the 8085 processor is connected to RESET of the 8237 processor.



**Fig. 6.34 : Interfacing of 8237 with 8085 processor.**

**TABLE - 6.11 : IO ADDRESSES OF 8237**

Name of the internal register of 8237	Binary address								Hexa address
	Decoder input and enable				Input to address pins of 8237				
	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	
Channel-0 Base and Current address register	0	1	1	0	0	0	0	0	60
Channel-0 Base and Current word count register	0	1	1	0	0	0	0	1	61
Channel-1 Base and Current address register	0	1	1	0	0	0	1	0	62
Channel-1 Base and Current word count register	0	1	1	0	0	0	1	1	63
Channel-2 Base and Current address register	0	1	1	0	0	1	0	0	64
Channel-2 Base and Current word count register	0	1	1	0	0	1	0	1	65
Channel-3 Base and Current address register	0	1	1	0	0	1	1	0	66
Channel-3 Base and Current word count register	0	1	1	0	0	1	1	1	67
Status/Command register	0	1	1	0	1	0	0	0	68
Request register	0	1	1	0	1	0	0	1	69
Write single mask register bit	0	1	1	0	1	0	1	0	6A
Mode register	0	1	1	0	1	0	1	1	6B
Clear first-last flip-flop	0	1	1	0	1	1	0	0	6C
Temporary register/Master clear	0	1	1	0	1	1	0	1	6D
Clear mask register	0	1	1	0	1	1	1	0	6E
Write all mask register bits	0	1	1	0	1	1	1	1	6F

**DMA Operation in 8085 using 8237**

After programming the 8237 in the slave mode, it will be ready to perform DMA. Once the 8237 is programmed it keeps on checking DMA request input from IO devices. When the 8237 detects a valid DMA request then it performs the following activity:

- 1) When the 8237 receives a DMA request from a peripheral it sends a hold request to the 8085 processor (provided the channel should be enabled and there should not be any pending higher priority DMA request).
- 2) When the 8085 processor receives a hold request, it will complete the current instruction execution and drive all its tristate (address, data and control) pins to **high impedance** state. Then, the 8085 sends an acknowledge signal to 8237.
- 3) On receiving an acknowledge from 8085, the 8237 will send an acknowledge to the peripheral device which requested DMA.
- 4) The 8237 asserts AEN **high**, which enables DMA memory address latches and disables the processor address latch.
- 5) Then the 8237 outputs the low byte address on A<sub>0</sub>-A<sub>7</sub> lines and high byte address on DB<sub>0</sub> to DB<sub>7</sub> lines. Also, the control signal ADSTB is asserted **high** to latch this address into external latches.

- 6) Also the DMA controller asserts appropriate read and write control signals to perform DMA transfer.
- 7) In block transfer mode, after performing one byte transfer the steps 4,5 and 6 are repeated again and again until the count is zero. In demand transfer mode the steps 4, 5 and 6 are repeated until an external end of process signal is applied or till the DMA request is deactivated. In single transfer mode the 8237 deactivate the hold request to the processor after one byte transfer by DMA.

### 6.2.9 DMA Controller - INTEL 8257

The DMA controller-8257 has been developed for 8085/8086/8088 microprocessor-based system. It is a device dedicated to perform a high speed data transfer between memory and IO device. The 8257 has four channels. So, it can be used to provide DMA to four IO devices. It cannot be connected in cascade like 8237 and it has less features than 8237.

For each DMA channel an address register and a count register has been dedicated to store the memory address and the count value for number of bytes to be read/written by DMA respectively. Apart from these dedicated registers, the 8257 has mode set and status registers.

The 8237 is a 40-pin IC and available in **Dual In-line Package (DIP)**. The pin configuration of 8257 is shown in Fig. 6.35. A brief description about the pins and signals of 8257 are listed in Table-6.12.

#### Features of 8257

- It has four independent DMA channels to service four IO devices.
- Each channel can be independently programmable to transfer upto 64 kb of data by DMA.
- Each channel can independently perform read transfer, write transfer and verify transfer.

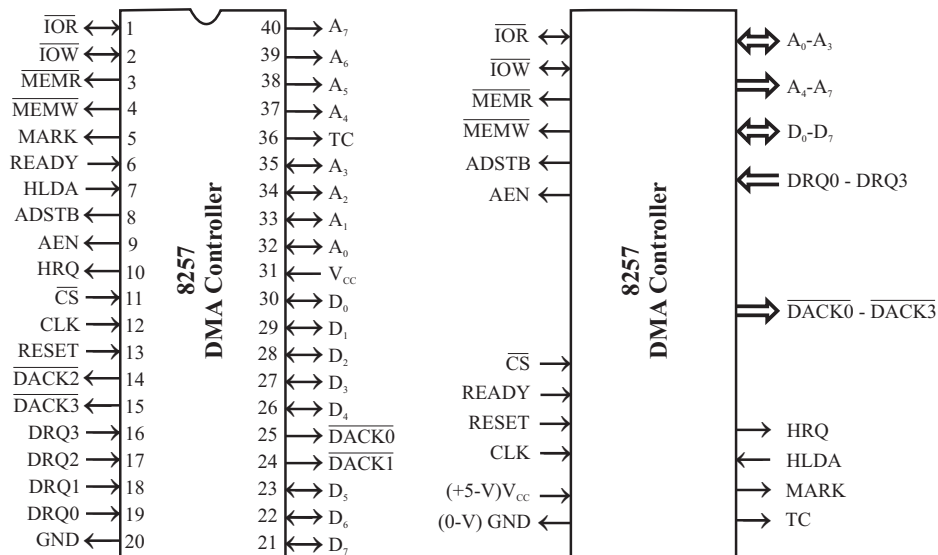


Fig. 6.35 : Pin configuration of an 8257.

**TABLE - 6.12 : PIN DESCRIPTION OF 8257**

Pin	Description
CLK	Clock input to 8257. Maximum clock frequency is 5 MHz. In 8085 system, the processor clock signal is inverted and applied to CLK of 8257.
$\overline{CS}$	Logic <b>low</b> chip select signal. It is input signal to select 8257 during programming mode.
RESET	Reset input to 8257. Connected to system reset, when the RESET signal goes <b>high</b> all the internal registers are cleared.
READY	Ready input signal and it is tied to $V_{cc}$ for normal timings. When READY input is tied <b>low</b> , the 8257 enter a wait state. This is used to get extra time in DMA machine cycles to transfer data between slow memory and IO devices.
HRQ	Hold request output signal. It is the hold request signal sent by 8257 to the processor HOLD pin, to make a request for bus to perform DMA transfer.
HLDA	Hold acknowledge input signal. It is the hold acknowledge signal to be send by the processor to inform the acceptance of hold request.
DREQ3 to DREQ0	DMA request inputs (Four channel inputs). Used by IO devices to request for DMA transfer.
DACK3 to DACK0	DMA acknowledge output signals. These are active <b>low</b> output signals from 8257 to the IO devices to inform the acceptance of DMA request.
$D_0 - D_7$	Data bus lines. These pins are used for data transfer between processor and DMA controller during programming mode. During DMA mode, these lines are used as multiplexed high order address and data lines.
$\overline{IOR}$	Bidirectional IO read control signal. It is input control signal for reading DMA controller during programming mode and output control signal for reading IO device during DMA (memory) write cycle.
$\overline{IOW}$	Bidirectional IO write control signal. It is input control signal for writing DMA controller during programming mode and output control signal for writing IO device during DMA (memory) read cycle.
TC	Terminal count.
MARK	Modulo-128 mark.
$A_3$ to $A_0$	Four bidirectional address lines. Used as input address during programming mode to select internal registers. During DMA mode the low order four bits of memory address are output by 8257 on these lines.
$A_7$ to $A_4$	Four unidirectional address lines. Used to output the memory address bits $A_7$ to $A_3$ during DMA mode.

Table - 6.12 : continued ....

Pin	Description
AEN	Address enable output signal. It is used to enable the address latch connected to $D_7 - D_0$ pins of 8257. It is also used to disable any buffers in the system connected to the processor.
ADSTB	Address strobe output signal. It is used to latch the high byte memory address issued through $D_7$ to $D_0$ lines by 8257 during DMA mode into an external latch.
MEMR	Memory read control signal. It is an output control signal issued during DMA read operation.
MEMW	Memory write control signal. It is an output control signal issued during DMA write operation.

### Functional Block Diagram of 8257

The functional block diagram of 8257 is shown in Fig. 6.36. The functional blocks of 8257 are data bus buffer, read/write logic, control logic and four numbers of DMA channels.

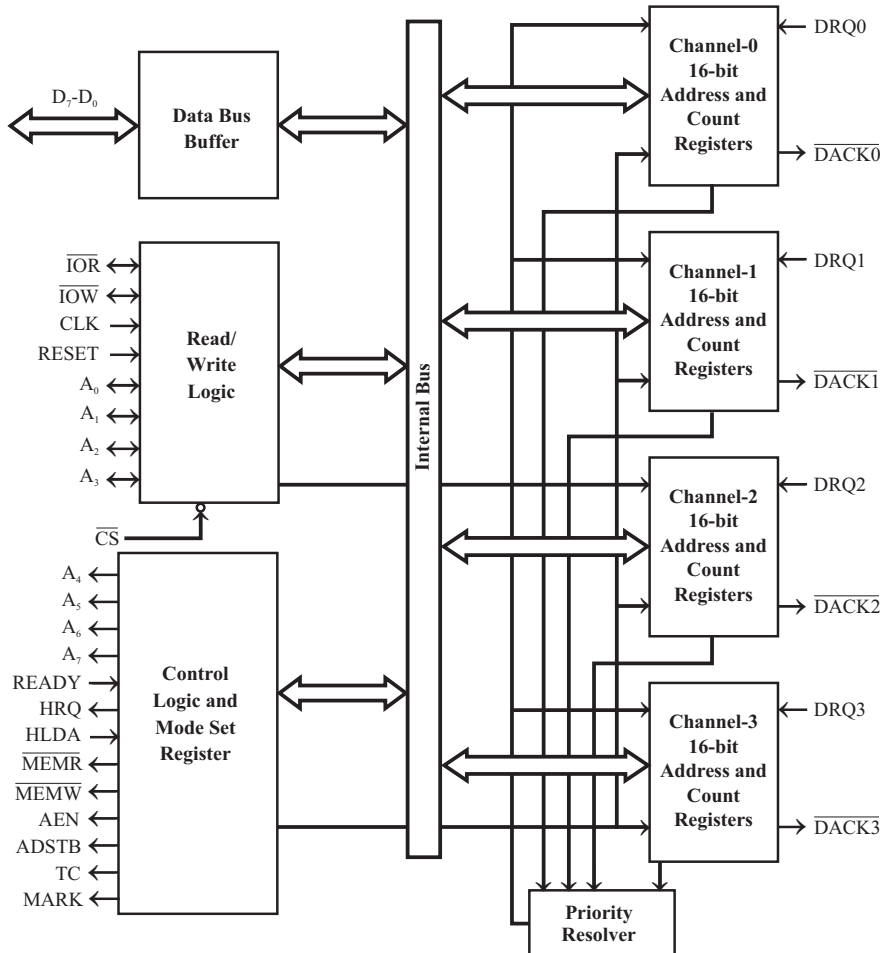
Each channel has two programmable 16-bit registers. One register is used to program the starting address of memory location for DMA data transfer and another register is used to program a 14-bit count value and a 2-bit code for type of DMA transfer (Read/Write/Verify transfer). The address in the address register is automatically incremented after every read/write/verify transfer. The format of count register is shown in Fig. 6.37(a).

In read transfer the data is transferred from memory to IO device. In write transfer the data is transferred from IO device to memory. Verification operations generate the DMA addresses without generating the DMA memory and IO control signals.

Apart from the address and count registers of each channel, the 8257 has a mode set register and status register. The mode set register is used to program various features of 8257 and the status register can be read to know the terminal count status of the channels. The registers of 8257 are selected for read/write operation during slave/programming mode by sending a 4-bit address to 8257 through  $A_0$  to  $A_3$  lines. The internal addresses of the registers of 8257 are listed in Table-6.13.

TABLE - 6.13 : INTERNAL ADDRESS OF 8257 REGISTERS

Register	Address			
	$A_3$	$A_2$	$A_1$	$A_0$
Channel-0 DMA address register	0	0	0	0
Channel-0 Count register	0	0	0	1
Channel-1 DMA address register	0	0	1	0
Channel-1 Count register	0	0	1	1
Channel-2 DMA address register	0	1	0	0
Channel-2 Count register	0	1	0	1
Channel-3 DMA address register	0	1	1	0
Channel-3 Count register	0	1	1	1
Mode set register (Write only)	1	0	0	0
Status register (Read only)	1	0	0	0



**Fig. 6.36 :** Functional block diagram of DMA controller 8257.

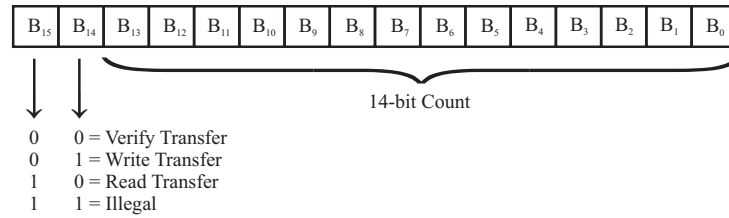
While programming 16-bit registers the low byte has to be sent first and then high byte. Internally, the loading of low byte and high byte into 16-bit registers are taken care of by a first/last flip-flop.

The mode set register is used to program the following features of 8257:

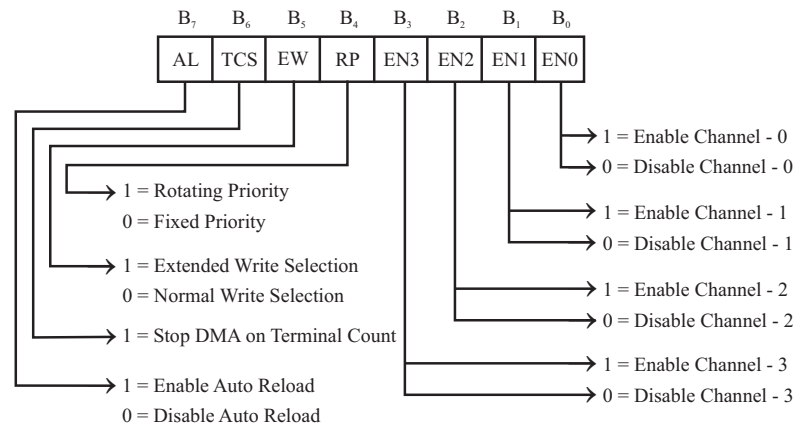
- Enable/disable a channel.
- Fixed/rotating priority
- Stop DMA on terminal count.
- Extended/normal write time.
- Auto reloading of channel-2.

The format of the control word to be loaded in mode set register of 8257 is shown in Fig. 6.37(b). The bits  $B_0$ ,  $B_1$ ,  $B_2$  and  $B_3$  of mode set register are used to enable/disable channel - 0, 1, 2 and 3 respectively. A one in this bit position will enable a particular channel and a zero will disable it.

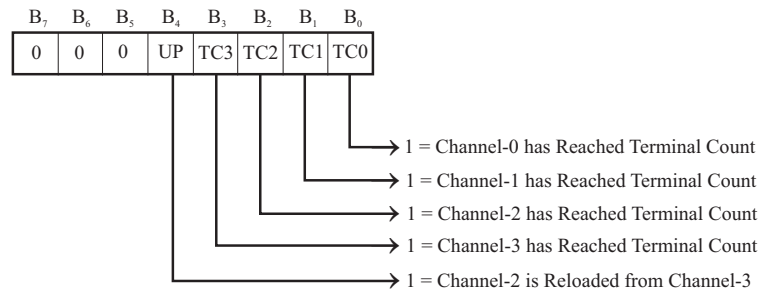




**Fig. a :** Format of count to be loaded in the count register of 8257.



**Fig. b :** Format of control word to be loaded in mode set register of 8257.



**Fig. c :** Status register of 8257.

**Fig. 6.37 :** Format of registers of 8257.

In the mode set register, if the bit  $B_4$  is set to one, then the channels will have rotating priority and if it zero then the channels will have fixed priority. In rotating priority, after servicing a channel its priority is made as lowest. In fixed priority the channel-0 has highest priority and channel-2 has lowest priority.

In mode set register, if the bit  $B_5$  is set to one, then the timing of write signals ( $\overline{MEMW}$  and  $\overline{IOW}$ ) will be extended and if the bit  $B_6$  is set to one then the DMA operation is stopped at the terminal count. The bit  $B_7$  is used to select the auto load feature for DMA channel-2. When bit  $B_7$  is set to one, then the content of channel-3 count and address registers are loaded to channel-2 count and address registers respectively whenever the channel-2 reaches terminal count. Therefore, when this mode is activated the number of channels available for DMA reduces from four to three.

The format of status register of 8257 is shown in Fig. 6.37(c). The processor can read the status of 8257 during slave mode to know the terminal count status of the channels. The bits  $B_0$ ,  $B_1$ ,  $B_2$  and  $B_3$  of status register indicates the terminal count status of channel-0, 1, 2 and 3 respectively. A one in these bit positions indicate that the particular channel has reached terminal count. These status bits are cleared after a read operation by microprocessor. The bit  $B_4$  of status register is called update flag and a one in this bit position indicates that the channel-2 registers has been reloaded from channel-3 registers in the auto load mode of operation.

### Interfacing of 8257 with 8085 Processor

A simple schematic for interfacing the 8257 with 8085 processor is shown in Fig. 6.38. The 8257 can be either memory-mapped or IO-mapped in the system. In the schematic shown in Fig. 6.38, the 8257 is IO-mapped in the system. The chip select signals for IO-mapped devices are generated by using a 3-to-8 decoder. The address lines  $A_4$ ,  $A_5$  and  $A_6$  are decoded to generate eight chip select signals (IOCS-0 to IOCS-7) and in this the chip select signal IOCS-6 is used to select 8257. The address line  $A_7$  and the control signal  $\overline{IO/\overline{M}}$  are used as enable for decoder. The IO addresses of the internal registers of 8257 are listed in Table-6.14.

**TABLE - 6.14 : IO ADDRESSES OF 8257 REGISTERS**

Register	Binary address								Hexa address
	Decoder input and enable				Input to address pins of 8257				
	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	
Channel-0 DMA address register	0	1	1	0	0	0	0	0	60
Channel-0 count register	0	1	1	0	0	0	0	1	61
Channel-1 DMA address register	0	1	1	0	0	0	1	0	62
Channel-1 count register	0	1	1	0	0	0	1	1	63
Channel-2 DMA address register	0	1	1	0	0	1	0	0	64
Channel-2 count register	0	1	1	0	0	1	0	1	65
Channel-3 DMA address register	0	1	1	0	0	1	1	0	66
Channel-3 count register	0	1	1	0	0	1	1	1	67
Mode set register (Write only)	0	1	1	0	1	0	0	0	68
Status register (Read only)	0	1	1	0	1	0	0	0	68

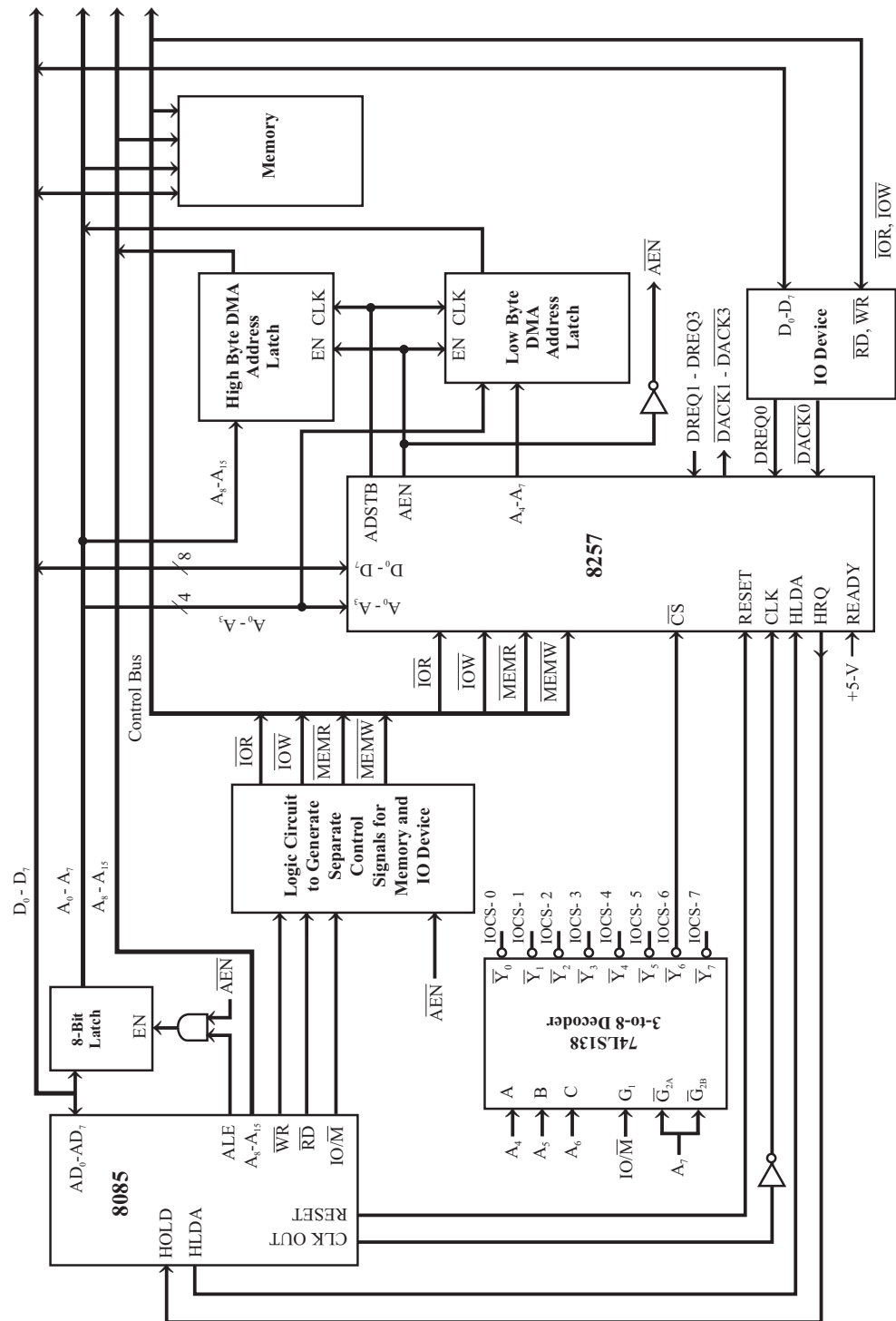


Fig. 6.38 : Interfacing DMA controller 8257 with 8085 microprocessor.

The  $D_0$ - $D_7$  lines of 8257 are connected to data bus lines  $D_0$ - $D_7$  for data transfer with processor during programming mode. These lines ( $D_0$ - $D_7$ ) are also used by 8257 to supply the memory address  $A_8$ - $A_{15}$  during the DMA mode. The 8257 also supply two control signal ADSTB and AEN to latch the address supplied by it during DMA mode on external latches. In the schematic shown in Fig. 6.38, two 8-bit latches are provided to hold the 16-bit memory address during DMA mode. During DMA mode, the AEN signal is also used to disable the buffers and latches used for address, data and control signals of the processor.

The 8257 provides separate read and write control signals for memory and IO devices during DMA. Therefore the  $\overline{RD}$ ,  $\overline{WR}$  and  $IO/\overline{M}$  of the 8085 processor are decoded by a suitable logic circuit to generate separate read and write control signals for memory and IO devices. (Please refer to Chapter-3, Fig. 3.18 for the logic circuit to generate separate read and write signals for memory and IO devices.)

The output clock of 8085 processor should be inverted and supplied to 8257 clock input for proper operation. The HRQ output of 8257 is connected to HOLD input of 8085 in order to make a HOLD request to the processor. The HLDA output of 8085 is connected to HLDA input of 8257, in order to receive the acknowledge signal from the processor once the HOLD request is accepted. The RESET OUT of 8085 processor is connected to RESET of 8257.

### **DMA Operation in 8085 using 8257**

In the slave mode the microprocessor sends control word to mode register, and programs the count and address registers of the required DMA channels. Once the 8257 is programmed it will keep on checking DMA request input from IO devices. Whenever a DMA request is made by an IO device the DMA operation is performed and the various steps of DMA operation are as follows :

- 1) When a peripheral device requires a DMA, it will assert DRQ signal **high**.
- 2) When the DRQ of a channel is asserted **high** and if the channel is enabled then the 8257 will assert HRQ (**HOLD Request**) as **high**.
- 3) When the 8085 processor receives a **high** signal on its HOLD pin, it will complete the current instruction execution and then drive all its tristate (address, data and control) pins to **high impedance** state and send an acknowledge signal to 8257 by asserting HLDA signal as **high**.
- 4) When the 8257 receive an acknowledge signal from 8085, the 8257 will send an acknowledge signal to the peripheral which requested DMA, by asserting  $\overline{DACK}$  signal as **low**.
- 5) The 8257 asserts AEN **high**, which enable the DMA memory address latches and disables the processor address latch. Then the 8257 outputs low byte DMA address on  $A_0$ - $A_7$  lines and high byte DMA address on  $D_0$ - $D_7$  lines. Also the ADSTB signal is asserted **high** to latch this address into external latches. Once the address is output on the address lines the content of address register is incremented by one and the count register is decremented by one.
- 6) Also the 8257 asserts appropriate read and write control signal to perform DMA transfer from peripheral to memory.
- 7) After performing one byte transfer the steps 5 and 6 are repeated again and again, until the terminal count (i.e., until the count reaches zero).

### 6.3 SERIAL DATA COMMUNICATION INTERFACE

#### 6.3.1 Serial Data Communication

The fastest way of transmitting data, within a microcomputer is parallel data transfer. For transferring data over long distances, however, parallel data transmission requires too many wires. Therefore, for long distance transmission, data is usually converted from parallel form to serial form so that it can be sent on a single wire or a pair of wires. Serial data received from a distant source is converted to parallel form so that it can be easily transferred on the microcomputer buses.

The three terms often encountered in literature on communication systems are simplex, half-duplex and full-duplex. A simplex data line can transmit data only in one direction. Data from sensors to processor and commercial radio stations are examples of simplex transmission.

Half-duplex transmission means that communication can take place in either direction between two systems, but can occur only in one direction at a time. An example of half duplex transmission is a two-way radio system, where one user always listens while the other talks because the receiver circuitry is turned off during transmit.

The term full-duplex means that each system can send and receive data at the same time. A normal phone conversation is an example of a full-duplex operation.

Serial data can be sent synchronously or asynchronously. In synchronous transmission, data are transmitted in block at a constant rate. The start and end of a block are identified with specific bytes or bit patterns. In asynchronous transmission, data is transmitted one by one. Each data has a bit which identifies its start and 1 or 2 bits which identify its end. Since each data is individually identified, data can be sent at any time. Figure 6.39 shows the bit format often used for transmitting asynchronous serial data.

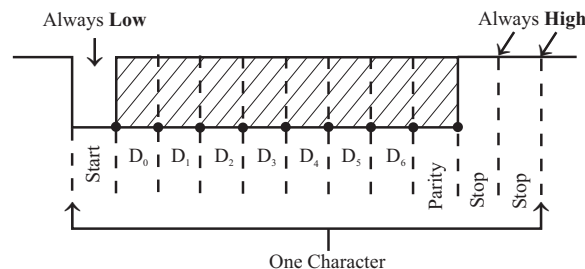


Fig. 6.39 : Bit format used for sending asynchronous serial data.

When no data is being sent, the signal line will be at constant **high** or marking state. The beginning of a data character is indicated by the line going **low** for 1 bit time. This bit is called a start bit. The data bits are then sent out on the line one after the other. Note that the least-significant bit is sent out first. Depending on the system, the data word may consist of 5,6,7 or 8 bits. Following the data bits is a parity bit, which is used to check for errors in received data. Some system do not insert or look for a parity bit. After the data bits and the parity bit, the signal line is returned **high** for at least 1-bit time to identify the end of the character. This **always-high** bit is referred to as a stop bit. Some systems may use 2 stop bits.

The term baud rate is used to indicate the rate at which serial data is being transferred. Baud rate is defined as  $\frac{1}{(\text{The time for a bit cell})}$ . In some systems one bit cell has one data bit, then the baud rate and bits/second are same. In other cases, 2 to 4 actual data bits are encoded within one transmitted bit time, so data bits per second and baud do not correspond. Commonly used baud rates are 110, 300, 1200, 2400, 4800, 9600 and 19,200 bauds.

In order to interface a microcomputer with serial data lines the data must be converted to and from serial form. A parallel-in-serial-out shift register and a serial-in-parallel-out shift register can be used to do this. In some cases of serial data transfer, handshake signals are needed to make sure that a transmitter does not send data faster than it can be read in by the receiving system. The programmable devices INTEL 8251A, National INS8250, etc., can be interfaced to microprocessors to perform such functions.

A device such as INTEL 8251A which can be programmed to do either asynchronous or synchronous communication is often called USART (**U**niversal **S**ynchronous **A**synchronous **R**eceiver **T**ransmitter). A device such as the National INS8250 which can only do asynchronous communication is often referred to as a **U**niversal **A**synchronous **R**eceiver **T**ransmitter (UART).

Once the data is converted to the serial form it must be in some way sent from the transmitting UART to the receiving UART. There are several ways in which serial data is commonly sent. One method is to use a current to represent a "1" in the signal line and no current to represent a "0". Another approach is to add line drivers at the output of the UART to produce a sturdy voltage signal. The range of each of these methods, however is limited to a few thousand feet.

For sending serial data over long distances the standard telephone system is a convenient path, because the wiring and connections are already in place. Standard phone lines, often referred to as switched lines because any two points can be connected together through a series of switches and have a bandwidth of about 300 to 3000 Hz. But, digital signals require very large bandwidth (typically 5 MHz). Therefore, for several reasons, digital signals cannot be sent directly over standard phone lines.

The solution to this problem is to convert the digital signals in to audio-frequency tones, which are in the frequency range that the phone lines can transmit. The device used to do this conversion and to convert transmitted tones back to digital information is called a MODEM. The term is a contraction of **m**odulator-**d**emodulator.

Modems and other equipment used to send serial data over long distances are known as data communication equipment or DCE. The terminals and computers that are sending or receiving the serial data are referred to as data terminal equipment or DTE.

### RS-232C Serial Data Standard

In serial IO, data can be transmitted as either current or voltage. Several standards have been developed for serial communication. When data is transmitted as voltage, the commonly used standard is known as RS-232C. It was developed by Electronics Industries Association(EIA), USA and adopted by IEEE. This standard, proposes a maximum of 25 signals for the bus used for serial data transfer. The 25 signals of RS-232C are listed in Table-6.15. In practice the first 9 signals are sufficient for most of the serial data transmission scheme and so the RS-232C bus signals are terminated on a D-type 9-pin connector. (When all the 25 signals are used, then RS232C serial bus is terminated on a 25-pin connector.)

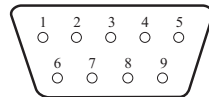
The voltage levels for all RS-232C signals are :

Logic **low** =  $-3\text{-V}$  to  $-15\text{-V}$  under load ( $-25\text{-V}$  on no load)

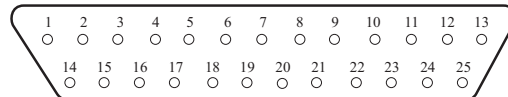
Logic **high** =  $+3\text{-V}$  to  $+15\text{-V}$  under load ( $+25\text{-V}$  on no load)

Commonly used voltage levels are :

$+12\text{-V}$  (logic **high**) and  $-12\text{-V}$ (logic **low**).



**Fig. a** : 9-pin D-type connector.



**Fig. b** : 25-pin D-type connector.

**Fig. 6.40** : Connectors used for terminating RS-232C bus.

The RS-232C signal levels are not compatible with TTL logic levels. Hence for interfacing TTL devices, level converters or RS-232C line drivers are employed. The popularly used level converters are,

MC1488 - TTL to RS-232C level converter.

MC1489 - RS-232C to TTL level converter.

MAX 232 - Bidirectional level converter.

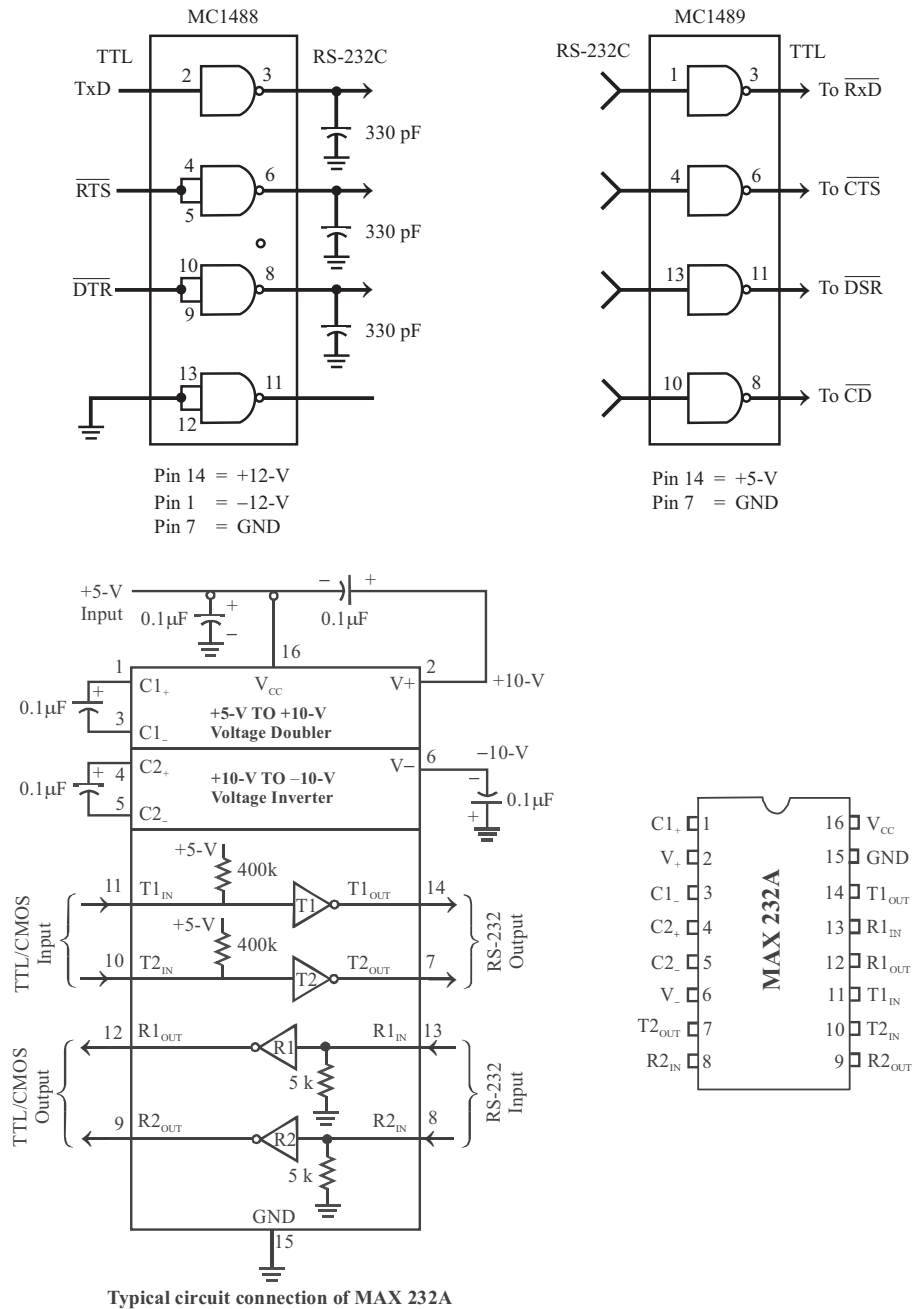
(Max 232 is equivalent to a combination of MC1488 and MC1489 in single IC.)

The signal level conversion using the above converters are shown in Fig. 6.41.

**TABLE - 6.15 : RS-232C PIN NAMES AND SIGNAL DESCRIPTION**

Pin number	Common name	RS-232 C name	Description	Signal direction on DCE
1	–	AA	Protective Ground	–
2	TxD	BA	Transmitted Data	IN
3	RxD	BB	Received Data	OUT
4	$\overline{\text{RTS}}$	CA	Request to send	IN
5	$\overline{\text{CTS}}$	CB	Clear to send	OUT
6	$\overline{\text{DSR}}$	CC	Data Set ready	OUT
7	GND	AB	Signal ground (Common return)	–
8	$\overline{\text{CD}}$	CF	Received line signal detector	OUT
9		–	Reserved for Data set testing	–
10		–	Reserved for Data set testing	–
11		–	Unassigned	–
12		SCF	Secondary Received Line signal Detector	OUT
13		SCB	Secondary clear to send	OUT
14		SBA	Secondary Transmitted Data	IN
15		DB	Transmission signal element timing (DCE source)	OUT
16		SBF	Secondary Received data	OUT
17		DD	Receiver signal element timing (DCE source)	OUT
18		–	Unassigned	–
19		SCA	Secondary request to send	IN
20	$\overline{\text{DTR}}$	CD	Data terminal ready	IN
21		CG	Signal quality detector	OUT
22		CE	Ring indicator	OUT
23		CH/CI	Data signal rate selector (DTE/DCE Source)	IN/OUT
24		DA	Transmit signal element timing (DTE source)	IN
25		–	Unassigned	–





**Note :** 1. For MAX 232 all capacitors should be 1  $\mu$ F.  
2. The voltage rating of all capacitors should be above 10-V.

Fig. 6.41 : TTL to RS-232C and RS-232C to TTL signal conversion.

### 6.3.2 USART - INTEL 8251A

The 8251A is a programmable serial communication interface chip designed for synchronous and asynchronous serial data communication. It is packed in a 28-pin DIP. The 8251A is the enhanced version of its predecessor, 8251 and it is compatible with 8251. The pin description of 8251A is shown in Fig. 6.42.

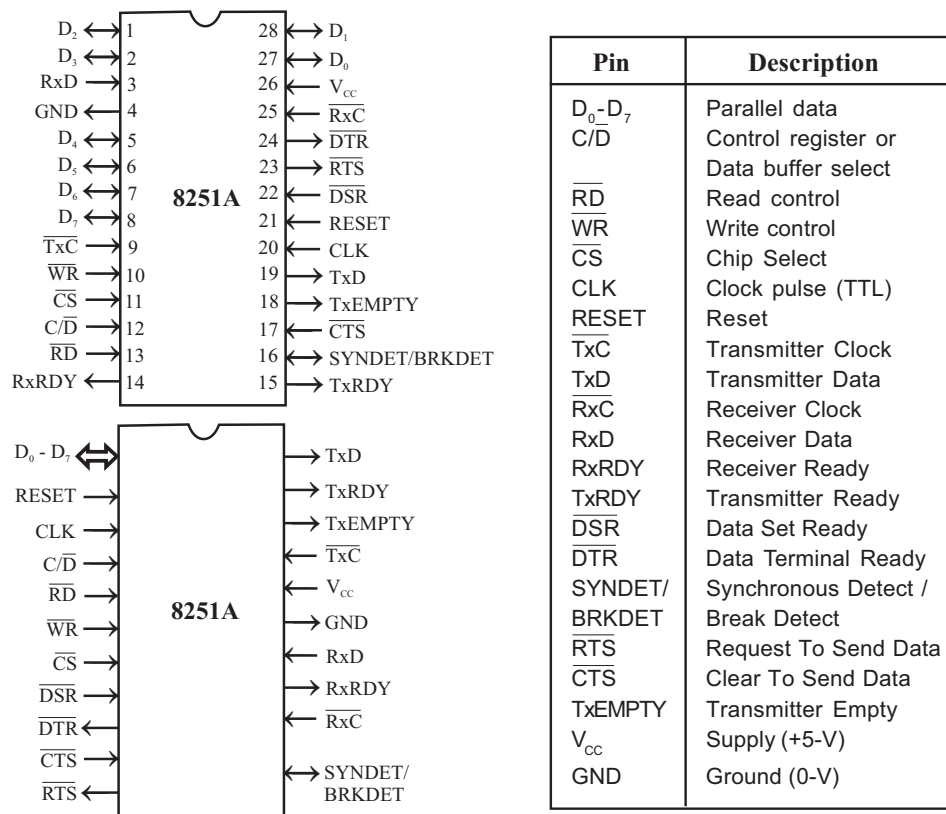
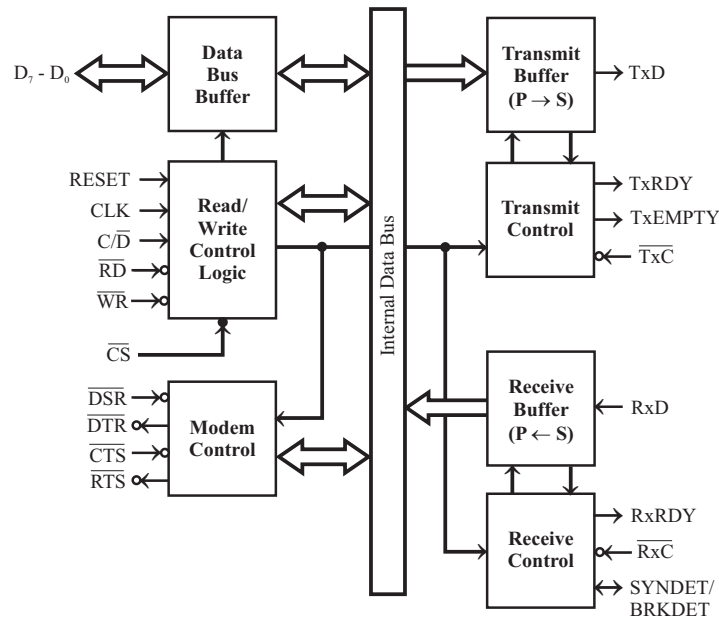


Fig. 6.42 : Pin description of 8251A.

The functional block diagram of 8251A is shown in Fig. 6.43. The block diagram shows five sections, they are Read/Write control logic, Transmitter, Receiver, Data bus buffer and Modem control.

#### Read/Write Control Logic

The Read/Write Control logic interfaces the 8251A with CPU, determines the functions of the 8251A according to the control word written into its control register and monitors the data flow. This section has three registers and they are control register, status register and data buffer.



**Fig. 6.43 :** The functional block diagram of 8251A - USART.

The signals  $\overline{RD}$ ,  $\overline{WR}$ ,  $C/\overline{D}$  and  $\overline{CS}$  are used for read/write operations with these registers. When  $C/\overline{D}$  is **high**, the control register is selected for writing control word or reading status word. When  $C/\overline{D}$  is **low**, the data buffer is selected for read/write operation.

A **high** on the reset input forces 8251A into the idle mode. The clock input is necessary for 8251A for communication with CPU and this clock does not control either the serial transmission or the reception rate.

### Transmitter Section

The transmitter section accepts parallel data from the CPU and converts them into serial data. The transmitter section is double buffered, i.e., it has a buffer register to hold an 8-bit parallel data and another register called output register to convert the previous data into a stream of serial bits.

The processor loads a data into buffer register. When output register is empty, the data is transferred from buffer to output register. Now the processor can again load another data in buffer register. If buffer register is empty, then TxRDY is asserted **high** and if output register is empty then TxEMPTY is asserted **high**. These signals can also be used as interrupt or status for data transmission.

The clock signal,  $\overline{Tx\overline{C}}$  controls the rate at which the bits are transmitted by the USART. The clock frequency can be 1, 16 or 64 times the baud rate.

### **Receiver Section**

The receiver section accepts serial data and converts them into parallel data. The receiver section is double buffered, i.e., it has an input register to receive serial data and convert to parallel and a buffer register to hold the previous converted data.

Normally RxD line is **high**, when the RxD line goes **low**, the control logic assumes it as a START bit, waits for half a bit time and samples the line again. If the line is still **low**, then the input register accepts the following bits, forms a character and loads it into the buffer register. The CPU reads the parallel data from the buffer register.

When the input register loads a parallel data to buffer register, the RxRDY line goes **high**. This signal can be used as an interrupt or status to indicate the readiness of the receiver section to CPU. The clock signal  $\overline{\text{RxC}}$  controls the rate at which bits are received by the USART. In the asynchronous mode, the clock frequency can be set to 1, 16 or 64 times the baud rate.

During asynchronous mode, the signal SYNDET/BRKDET will indicate the intentional break in the data transmission. If the RxD line remains **low** for more than 2 character time then this signal is asserted **high** to indicate the break in the transmission.

During synchronous mode, the signal SYNDET/BRKDET will indicate the reception of synchronous character. If the 8251A finds a synchronous character in the incoming string of data bits then it asserts SYNDET signal as **high**.

### **MODEM Control**

The MODEM control unit allows to interface a MODEM to 8251A and to establish data communication through MODEM over telephone lines. This unit takes care of handshake signals for MODEM interface.

### **Programming the 8251A**

The 8251A is programmed by sending mode word and command word. First reset the 8251A and then send a mode word to control register address. Next, the command word is sent to the same address. The CPU can check the readiness of the 8251A for data transfer by reading the status register. The format of control and status words are shown in Fig. 6.44.

The mode word informs 8251 about the baud rate, character length, parity and stop bits. The command word can be sent to enable the data transmission and/or reception. The information regarding the readiness of transmitter/receiver and the transmission errors can be obtained from status word.

If 8251A is programmed for a baud rate factor of 64x through mode word then the baud rate is clock frequency divided by 64. If baud rate factor is 16x, then baud rate is clock frequency divided by 16. If baud rate factor is 1x, then the baud rate is given by clock frequency.

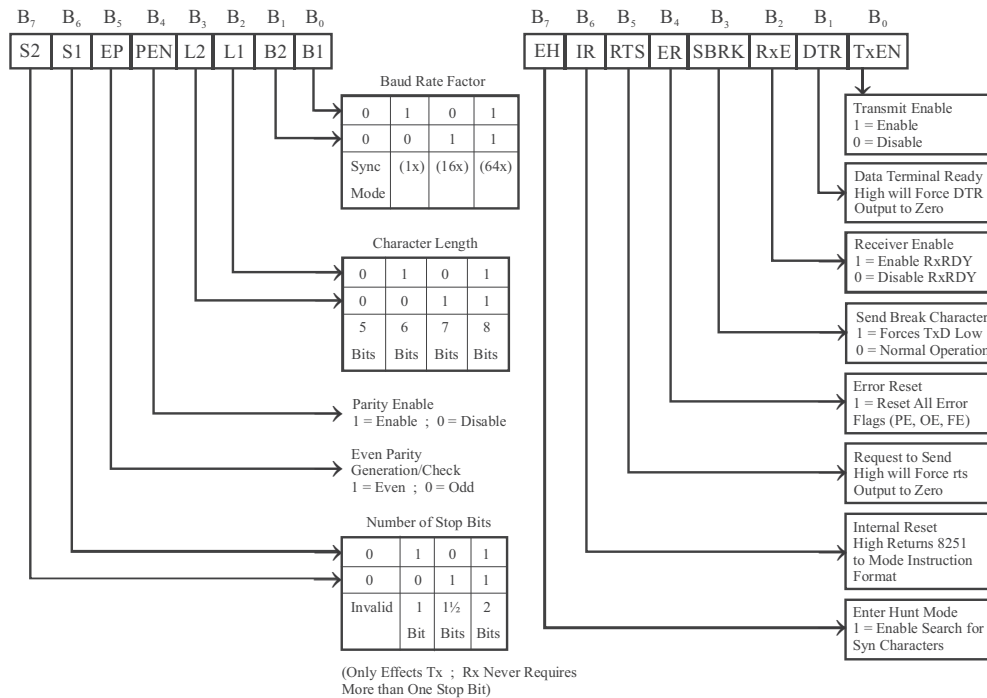


Fig. a : Mode word.

Fig. b : Command word.

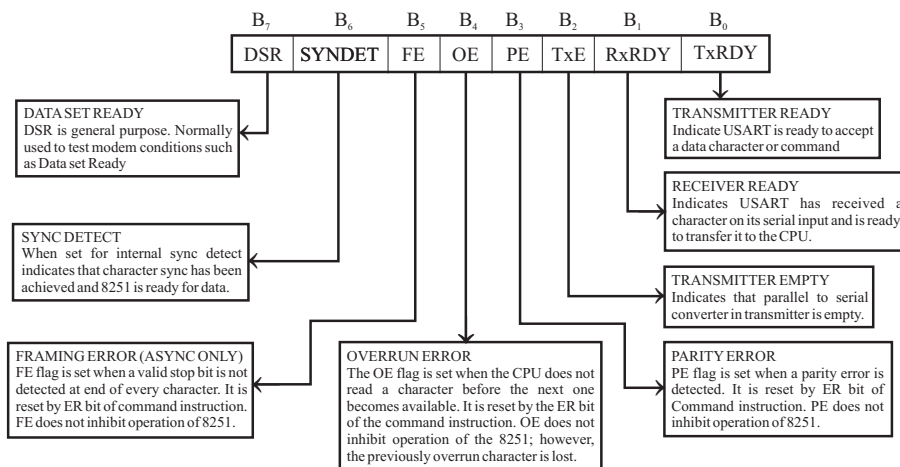


Fig. c : Status word.

Fig. 6.44 : Format of 8251A mode, command and status words.

### Interfacing 8251A with 8085

A simple schematic for interfacing the 8251A with 8085 processor is shown in Fig. 6.45. The 8251A can be either memory-mapped or IO-mapped in the system. In the schematic shown in Fig. 6.45, the 8251A is IO-mapped in the system. The chip select signals for IO-mapped devices are generated by using a 3-to-8 decoder. The address lines  $A_4$ ,  $A_5$  and  $A_6$  are decoded to generate eight chip select signals (IOCS-0 to IOCS-7) and in this, the chip select signal IOCS-2 is used to select 8251A. The address line  $A_7$  and the control signal  $\overline{IO/\overline{M}}$  are used as enable for decoder.

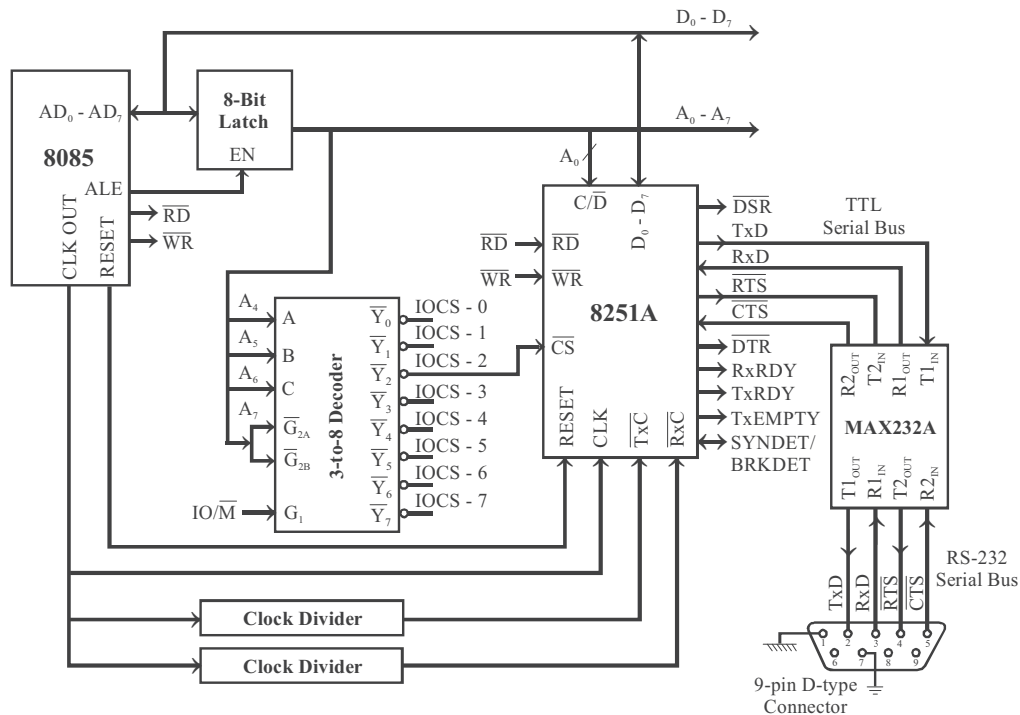


Fig. 6.45 : Interfacing of 8251A to 8085 microprocessor.

TABLE - 6.16 : IO ADDRESSES OF 8251A

Internal device of 8251A	Binary address								Hexa address
	Decoder input and enable				Input to address pin of 8251				
	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	
Data buffer	0	0	1	0	x	x	x	0	20
Control register	0	0	1	0	x	x	x	1	21

Note : The don't care "x" is considered as zero.

The address line  $A_0$  of 8085 is connected to  $C/\overline{D}$  of 8251A to provide the internal addresses. The IO addresses allotted to the internal devices of 8251A are listed in Table-6.16. The data lines  $D_0$ - $D_7$  are connected to  $D_0$ - $D_7$  of the processor to achieve parallel data transfer. The RESET and clock signals are supplied by the processor. Here the processor clock is directly connected to 8251A. This clock controls the parallel data transfer between the processor and 8251A.

The output clock signal of 8085, is divided by suitable clock dividers and then used as clock for serial transmission and reception ( $\overline{TxC}$  and  $\overline{RxC}$ ). In 8251A the transmission and reception baud rates can be different or same. Usually a programmable timer, 8254 (which is discussed in Section 6.5) is used to divide the processor output clock and supply to  $\overline{TxC}$  and  $\overline{RxC}$  at the required rate.

The TTL logic levels of the serial data lines (RxD and TxD) and the control signals necessary for serial transmission and reception are converted to RS232 logic levels using MAX232 and then terminated on a standard 9-pin D-type connector. The device which requires serial communication with processor can be connected to this 9-pin D-type connector using 9-core cable.

The signals TxEMPTY, TxRDY and RxRDY can be used as interrupt signals to initiate interrupt driven data transfer scheme between processor and 8251A.

## 6.4 KEYBOARD AND DISPLAY INTERFACE

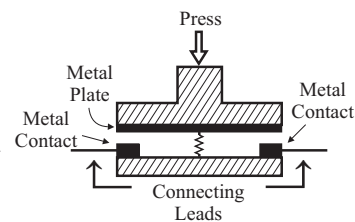
### 6.4.1 Keyboard Interface Using Ports

A common method of entering programs into a microcomputer is through a keyboard which consists of a set of switches. Basically each switch will have two normally open metal contacts. These two contacts can be shorted by a metal plate supported by a spring as shown in Fig. 6.46. On pressing the key, the metal plate will short the contacts and on releasing the key, again the contacts will be open. The processor has to perform the following three major task to get a meaningful data from a keyboard.

1. Sense a key actuation.
2. Debounce the key.
3. Decode the key.

The three major tasks mentioned above can be performed by the software, when a keyboard is connected through ports to 8085 processor. Consider a simple keyboard in which the keys are arranged in rows and columns as shown in Fig. 6.47. The rows are connected to port-A lines of 8255 and columns are connected to port-B lines, of the same chip. The rows and columns are normally tied **high**. At the intersection of a row and column, a key is placed such that pressing a key will short the row and the column.

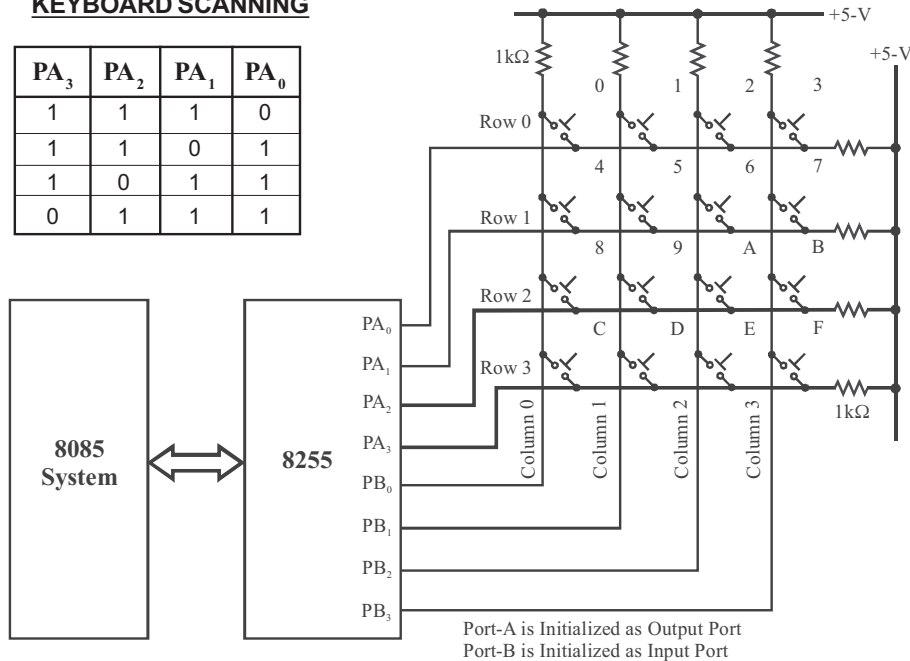
A key actuation is sensed by sending a **low** to all the rows through port-A. Pressing a key will short the row and column to which it is connected. So, the column to which the key is connected will be pulled **low**. Therefore, the columns are read through port-B to see whether any



**Fig. 6.46** : A representation of keyboard switch.

**TABLE - 6.17 : SCAN CODE FOR  
KEYBOARD SCANNING**

PA <sub>3</sub>	PA <sub>2</sub>	PA <sub>1</sub>	PA <sub>0</sub>
1	1	1	0
1	1	0	1
1	0	1	1
0	1	1	1



**Fig. 6.47 : Keyboard interfacing using ports.**

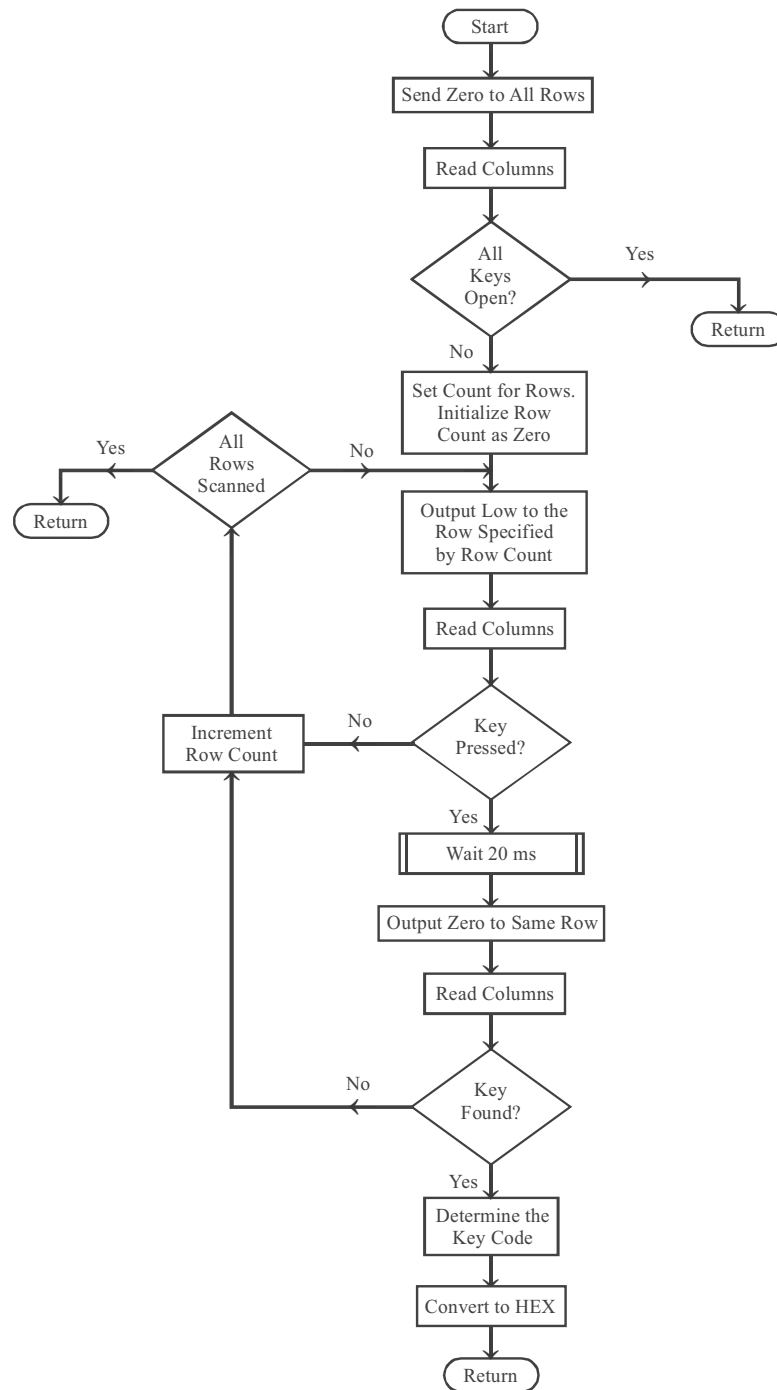
of the normally **high** columns are pulled **low** by a key actuation. If they are, then rows can be checked individually to determine the row in which the key is down. For checking each row, the scan code of the type shown in Table-6.17 are output to port-A one by one. This process of sensing a key actuation is called keyboard scanning.

A key press has to be accepted only after debouncing. Normally, the key bounces for 10 to 20 milliseconds when it is pressed and released. The bouncing time depends on the type of the key. When this bounce occurs, it may appear to the microcomputer that the same key has been actuated several times instead of just one time. This problem can be eliminated by scanning the row in which the key press is deducted after 10 to 20 millisecond and then verifying to see if the same key is still down. If it is, then the key actuation is valid. This process is called key debouncing.

After debouncing, the code for the key has to be generated. Each key can be individually identified by the port-A output value (row code) and port-B input value (column code). The next step is to translate the row and column code into a more popular code such as hexadecimal or ASCII. This can easily be accomplished by a program. The flowchart for the keyboard scanning when the keyboard is interfaced using ports is shown in Fig. 6.48.

In keyboard interfacing there are two methods of handling multiple key press and they are two-key lockout and N-key rollover. The two-key lockout takes into account only one key pressed. An additional key pressed and released does not generate any codes. The system is simple to



**Fig. 6.48 :** Flowchart for keyboard scanning subroutine.

implement and more often used. However, it might slow down the typing since each key must be fully released before the next one is pressed down. On the other hand, the N-key rollover will detect all the keys pressed in the order of entry and generates corresponding keycode.

The disadvantage in keyboard interfacing using ports is that, most of the processor time is utilized (or wasted) in keyboard scanning and debouncing.

#### 6.4.2 Display Interface Using Ports

The 7-segment LEDs are the most popular display devices used for single board microcomputers (microprocessor trainer kits). The 7-segment LEDs can be either common anode type or common cathode type.

Each 7-segment LED will have seven **L**ight **E**mitting **D**iodes (LEDs) arranged in the form of small rectangular segments and another LED as a dot point in a single package. In common cathode type, all the cathode terminals of LEDs are internally shorted and one/two pins are provided for external connection. The anode of the LEDs are terminated on separate pins for external connection. The pin configuration and the internal connection of common cathode 7-segment LED are shown in Fig. 6.49.

In common anode type all the anode terminals of LEDs are internally shorted and one/two pins are provided for external connection. The cathode of LEDs are terminated on separate pins for external connection. The pin configuration and the internal connection of common anode 7-segment LED are shown in Fig. 6.50.

In the 7-segment LED a segment will glow or emit light when it is forward biased. Therefore, a segment can be made to glow, by applying a **high** (logic-1/+5-V) to anode and a **low**(logic-0/0-V) to the cathode. The alphabetic/numeric characters can be displayed on the 7-segment LED by forward biasing the appropriate segments.

In common cathode 7-segment LED the common point is tied to logic-0. To display a character, logic-1 is applied to anode of segments which has to emit light and logic-0 is applied to anode of segments which should not emit light. The binary and hex codes for displaying the decimal digits 0 to 9 in common cathode 7-segment LED are listed in Table -6.18.

In common anode 7-segment LED the common point is tied to logic-1. To display a character, logic-0 is applied to cathode of segments which has to emit light and logic-1 is applied to cathode of segments which should not emit light. The binary and hex codes for displaying the decimal digits 0 to 9 in common anode 7-segment LED are listed in Table-6.19.

The display codes for LEDs can be generated by using the BCD to 7-segment decoder, IC 7447. When a BCD code is sent to the input of the 7447, it outputs **low** on the segments required to display the number represented by the BCD code. A simple schematic is shown in Fig. 6.51, to interface a common anode 7-segment LED to 8085 system using a port device. This circuit connection is referred to as static display, because current is being passed through the display at all times.

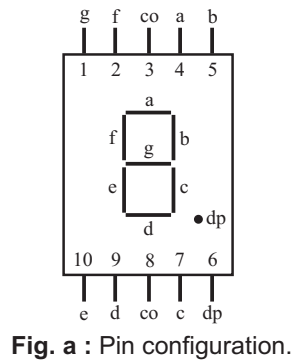


Fig. a : Pin configuration.

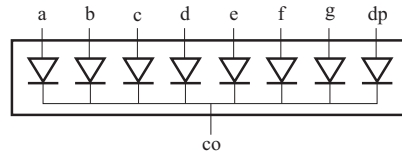


Fig. b : Internal connection.

co - common cathode  
dp - anode of dot point  
a, b, c, d, e, f, g - anodes of segments

Fig. 6.49 : Common cathode 7-segment LED.

TABLE - 6.18 : 7-SEGMENT DISPLAY CODE FOR COMMON CATHODE LED

BCD digit	Binary code								Hexa code
	dp	g	f	e	d	c	b	a	
0	0	0	1	1	1	1	1	1	3F
1	0	0	0	0	0	1	1	0	06
2	0	1	0	1	1	0	1	1	5B
3	0	1	0	0	1	1	1	1	4F
4	0	1	1	0	0	1	1	0	66
5	0	1	1	0	1	1	0	1	6D
6	0	1	1	1	1	1	0	1	7D
7	0	0	0	0	0	1	1	1	07
8	0	1	1	1	1	1	1	1	7F
9	0	1	1	0	1	1	1	1	6F

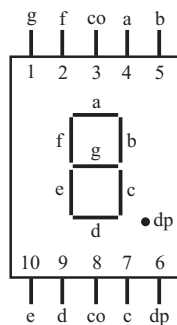


Fig. a : Pin configuration.

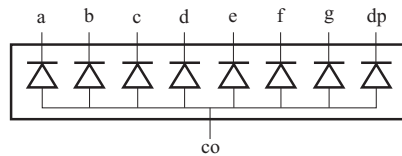


Fig. b : Internal connection.

co - common anode  
dp - cathode of dot point  
a, b, c, d, e, f, g - cathodes of segments

Fig. 6.50 : Common anode 7-segment LED.

TABLE - 6.19 : 7-SEGMENT DISPLAY CODE FOR COMMON ANODE LED

BCD digit	Binary code								Hexa code
	dp	g	f	e	d	c	b	a	
0	1	1	0	0	0	0	0	0	C0
1	1	1	1	1	1	0	0	1	F9
2	1	0	1	0	0	1	0	0	A4
3	1	0	1	1	0	0	0	0	B0
4	1	0	0	1	1	0	0	1	99
5	1	0	0	1	0	0	1	0	92
6	1	0	0	0	0	0	1	0	82
7	1	1	1	1	1	0	0	0	F8
8	1	0	0	0	0	0	0	0	80
9	1	0	0	1	0	0	0	0	90

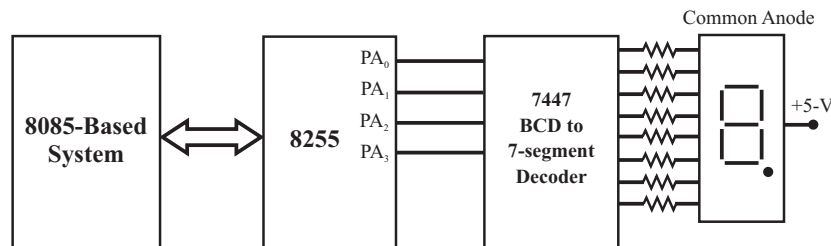


Fig. 6.51 : 7-Segment LED display using port.

A typical microprocessor system normally requires 6 to 8 numbers of 7-segment LEDs. The current requirement of each 7-segment LED is 140 mA to 200 mA. Hence, the total current requirement for 6 numbers of 7-segment LEDs will be 1200 mA. Also, each 7-segment LED requires a 7447 decoder and 4 lines of a port. The current required by the decoder and the LED displays might be several times the current required by the rest of the circuit in the microprocessor system.

The heavy current requirement in static display can be reduced drastically by using multiplexed display scheme. In multiplexed display only one 7-segment display is made to glow at a time. Each 7-segment LED is turned ON at definite intervals. Due to persistence of vision the display appears to be continuous to a human eye. (Actually LEDs are turned ON and OFF.)

*Note : A human eye can retain an image for 125 milliseconds.*

Following are the advantages of multiplexed display :

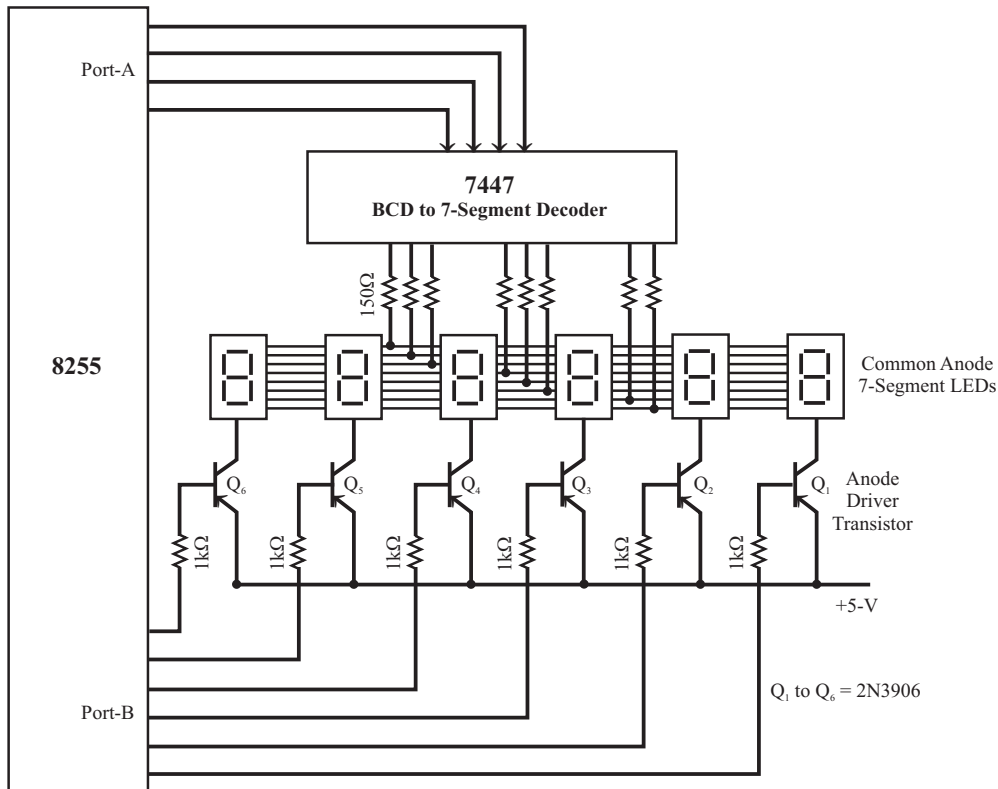
1. Only one 7447 is needed for all the 7-segment LEDs.
2. In a current requirement of one 7-segment LED, 6 to 8 LEDs can be interfaced.

Figure 6.52 shows a multiplexed display of 6 numbers of 7-segment common anode LEDs.

The segment pins (cathodes) of 7-segment LEDs are connected to a common bus. The output of the decoder (7447) is connected to this common bus. The BCD code for the character to be displayed is sent to 7447 through port-A lines. The common anode of each 7-segment LED has a driver transistor (PNP type). A driver transistor can be turned ON by sending **low** to the base of the transistor through port-B lines.

The trick of multiplexed display is that the segment information is sent out to all of the digits on the common bus, but only one display digit is turned on at a time. The PNP transistors in series with the common-anode of each digit acts as an ON and OFF switch for that digit.

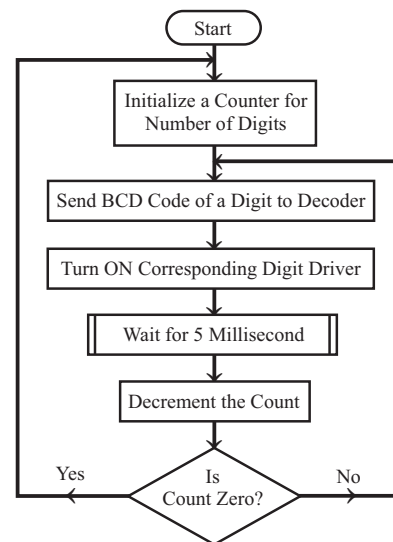
The BCD code for digit-1 is first output from port-A to the 7447. The 7447 outputs the corresponding seven-segment code on the segment bus lines. The transistor connected to digit-1 is then turned ON by outputting a **low** on the corresponding bit of port-B (Remember, a **low** turn ON a PNP transistor). All of the rest of the bits of port-B should be **high** to make sure no other digits are turned ON. After a few milliseconds, digit-1 is turned OFF by outputting all **high** to port-B.



**Fig. 6.52 :** A schematic diagram of a multiplexed display using ports.

Next the BCD code for digit-2 is output to the 7447 on port-A and a data to turn ON the driver transistor of digit-2 is output on port-B. After a few milliseconds, digit-2 is turned OFF and the process is repeated for digit-3. This process is continued until all of the digits have had a turn. Then digit-1 and the following digits are turned ON again in turn. This process is also called display refreshing.

With 6 digits and 5 ms per digit, we can get back to digit-1 every 25 ms or about 40 times a second. This refresh rate is fast enough so that, all digits appear to be turned ON all the time. Refresh rates of 40 to 200 times a second are acceptable. A flowchart for the operational flow in multiplexed display is shown in Fig. 6.53.



**Fig. 6.53 :** Flowchart for multiplexed display.

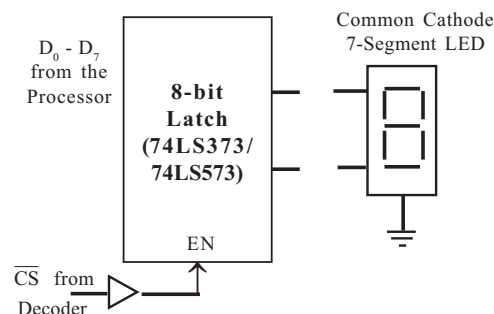
The great advantages of multiplexing the displays are that only one 7447 is required and only one digit is ON at a time. Hence it results in large saving in power and parts.

### 6.4.3 Latches and Buffers as IO Devices

The Latches and Buffers can be used as IO ports. Basically in programmable IO ports of 8155/8255/8355/8755 the ports are made of latches and buffers. Latches can be used as output ports and buffers can be used as input ports. Examples of 8-bit latches are 74LS373, 74LS273, 74LS573, 74LS574, etc. Examples of 8-bit buffers are 74LS245, 74LS244, 74LS240, INTEL 8286, etc.

The 8-bit latch can be used as an output device to interface LEDs or seven segment LEDs as shown in Fig. 6.54. A Latch is selected by a chip select signal. It can be mapped in the system either by IO mapping or memory mapping. The processor has to send an address to select the latch. A decoder in the system produces Chip Select signal ( $\overline{CS}$ ) which enables the latch. Then the processor loads the display code on the data bus. The latch will hold the display code, until it is loaded with another display code.

In Fig. 6.54, a segment is turned ON if a one is sent to anode through the latch. The cathode is permanently tied to ground.

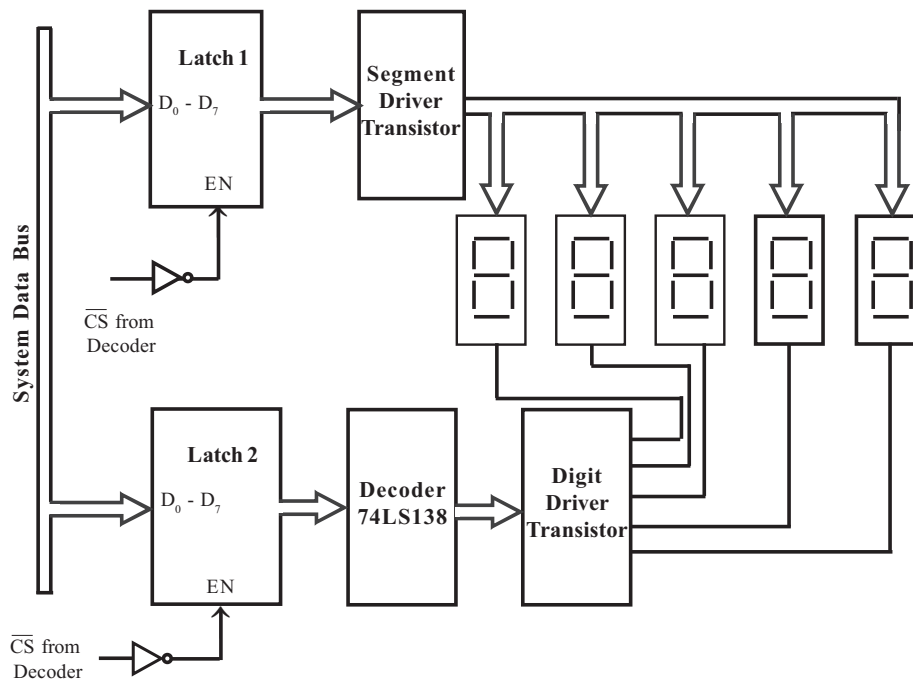


**Fig. 6.54** : An 8-bit latch as output port.

Using latches and decoders multiplexed display is also possible as shown in Fig. 6.55. In this scheme two latches are used. One for turning ON the segment driver transistor and another for turning ON the digit driver transistor. The segments (Anodes in case of common cathode LEDs) are connected to a common bus. Driver transistors are provided to satisfy the current requirement of LEDs. The segment transistors can be turned ON by sending appropriate code through Latch-1.

The common point (Cathode in case of common cathode LEDs) of each 7-segment LED is connected to a driver transistor (digit driver). The digit driver transistors can be turned ON, one at a time by the decoder.

The processor selects Latch-1 and sends the display code (7-segment code) to it. Then the processor selects Latch-2 and send an appropriate binary count to turn on a particular digit. The input to the decoder 74LS138 is a binary count and the output of the decoder will turn ON only one digit transistor. After a delay time (typically 3 to 10 ms) all the segments are turned OFF by sending appropriate code to segment drivers. Then the display code for next digit is sent to segment drivers and the corresponding digit driver is turned ON, by sending an appropriate count to decoder.



**Fig. 6.55 :** Multiplexed 7-segment LED display using latches.

The disadvantage in using ports and latches as output devices is that a considerable processor time is consumed for display refreshing.

Similarly an input device such as a keyboard can also be interfaced using buffers. The interfacing of keyboard using buffers will be similar to that of keyboard interface using ports of 8255 discussed in Section 6.4.1. The disadvantage in using buffers for keyboard interfacing is that most of the processor time is consumed for keyboard scanning and debouncing.



### 6.4.4 Keyboard /Display Controller - INTEL 8279

The INTEL 8279 is a dedicated controller specially developed for interfacing keyboard and display devices to 8085/8086/8088 microprocessor-based system. It relieves the processor from the time consuming task like keyboard scanning and display refreshing.

The important features of 8279 are :

- Simultaneous keyboard and display operations.
- 2-key lockout or N-key rollover with contact debounce.
- Scanned keyboard mode.
- Scanned sensor mode.
- Strobed input entry mode.
- 8-character keyboard FIFO.
- 16-character display.
- Right or left entry 16-byte display RAM.
- Mode programmable from CPU.
- Programmable scan timing.
- Interrupt output on key entry.

The 8279 provides an interface for a maximum of 64-contact key matrix (arranged as  $8 \times 8$  matrix array of key switches). Keyboard entries are debounced and stored in the internal FIFO RAM. It generates an interrupt signal for each key entry, to inform the processor to read the keycode from FIFO.

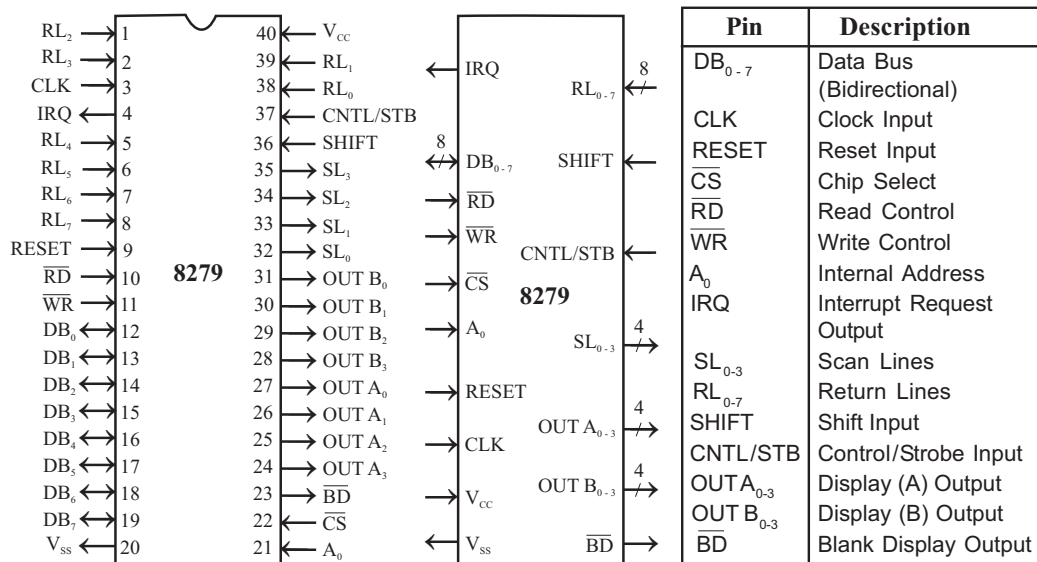


Fig. 6.56 : Pin description of 8279.

The 8279 provides a multiplexed interface for 7-segment LEDs and other popular display devices. It consists of  $16 \times 8$  display RAM which can also be organized into dual  $16 \times 4$  RAM. The CPU has to load the display codes in this RAM. Once the data is loaded, the 8279 takes care of display and refreshing. A maximum of 16 numbers of 7-segment LEDs can be interfaced using 8279.

The 8279 is a 40-pin IC available in DIP (Dual In-Line Package). The pin configuration of 8279 is shown in Fig. 6.56. The 8279 has two internal addresses decided by the logic level of  $A_0$ . If  $A_0$  is **low** then the processor can read or write to data register of 8279. If  $A_0$  is **high** then the processor can write to control register or read status register. The 8279 can be either IO-mapped or memory-mapped in the system.

### Block Diagram of 8279

The functional block diagram of 8279 is shown in Fig. 6.57. The four major sections of 8279 are keyboard, scan, display and CPU interface.

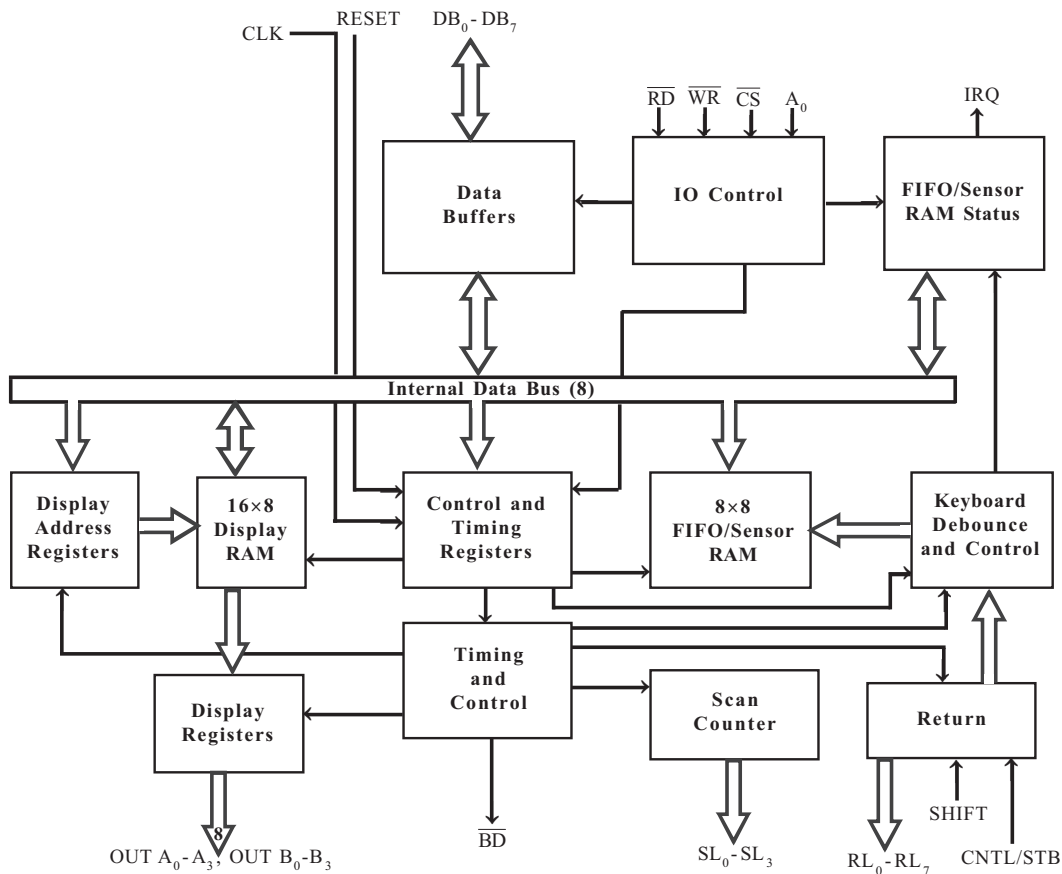


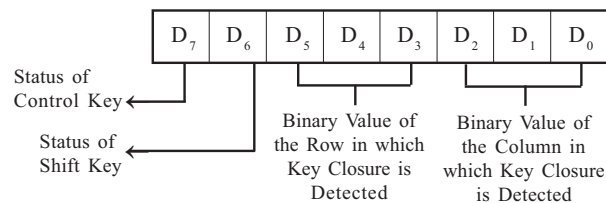
Fig. 6.57 : Block diagram of 8279.

### Keyboard section

The keyboard section consists of eight return lines  $RL_0 - RL_7$  that can be used to form the columns of a keyboard matrix. It has two additional input : shift and control/strobe. The keys are automatically debounced. The two operating modes of keyboard section are 2-key lockout and

N-key rollover. In the 2-key lockout mode, if two keys are pressed simultaneously, only the first key is recognized. In the N-key rollover mode simultaneous keys are recognized and their codes are stored in FIFO.

The keyboard section also has a  $8 \times 8$  FIFO (**F**irst-**I**n-**F**irst-**O**ut) RAM. The FIFO can store eight keycodes in the scan keyboard mode. The status of the shift key and control key are also stored along with keycode. The 8279 generates an interrupt signal when there is an entry in FIFO. The format of keycode entry in FIFO for scan keyboard mode is shown in Fig. 6.58.



**Fig. 6.58 :** Keycode entry in FIFO for scan keyboard mode.

In sensor matrix mode the condition (i.e., open/close status) of 64 switches is stored in FIFO RAM. If the condition of any of the switches change then the 8279 asserts IRQ as **high** to interrupt the processor.

### Display section

The display section has eight output lines divided into two groups  $A_0$ - $A_3$  and  $B_0$ - $B_3$ . The output lines can be used either as a single group of eight lines or as two groups of four lines, in conjunction with the scan lines for a multiplexed display. The output lines are connected to the anodes through driver transistor in case of common cathode 7-segment LEDs. The cathodes are connected to scan lines through driver transistors. The display can be blanked by  $\overline{BD}$  line. The display section consists of  $16 \times 8$  display RAM. The CPU can read from or write into any location of the display RAM.

### Scan section

The scan section has a scan counter and four scan lines,  $SL_0$  to  $SL_3$ . In decoded scan mode, the output of scan lines will be similar to a 2-to-4 decoder. In encoded scan mode, the output of scan lines will be binary count, and so an external decoder should be used to convert the binary count to decoded output. The scan lines are common for keyboard and display. The scan lines are used to form the rows of a matrix keyboard and are also connected to digit drivers of a multiplexed display to turn ON/OFF.

### CPU interface section

The CPU interface section takes care of data transfer between 8279 and the processor. This section has eight bidirectional data lines  $DB_0$ - $DB_7$  for data transfer between 8279 and CPU. It requires two internal address  $A_0 = 0$  or 1, for selecting either data buffer or control register of 8279. The control signals  $\overline{WR}$ ,  $\overline{RD}$ ,  $\overline{CS}$  and  $A_0$  are used for read/write to 8279. It has an interrupt request line IRQ, for interrupt driven data transfer with processor.

**Write Display RAM**

**Code :**

1	0	0	AI	A	A	A	A
---	---	---	----	---	---	---	---

The CPU sets up the 8279 for a write to the Display RAM by first writing this command. After writing the command with  $A_0 = 1$ , all subsequent writes with  $A_0 = 0$  will be to the Display RAM. The addressing and Auto increment functions are identical to those for the Read Display RAM.

**Display Write Inhibit/Blanking**

**Code :**

				A	B	A	B
1	0	1	X	IW	IW	BL	BL

The IW Bits can be used to mask nibble A and nibble B in application requiring separate 4-bit display ports. By setting the IW flag (IW=1) for one of the ports, the port becomes masked.

The BL flags are available for each nibble. The last Clear command issued determined the code to be used as a **blank**.

**Clear**

**Code :**

1	1	0	$C_D$	$C_D$	$C_D$	$C_F$	$C_A$
---	---	---	-------	-------	-------	-------	-------

The  $C_D$  bits are available in this command to clear all rows of the Display RAM to a selectable blanking code as follows.

<b>CD</b>	<b>CD</b>	<b>CD</b>	
0	X	All Zeros (X = Don't Care)	
1	0	AB = Hex 20 (0010 0000)	
1	1	All Ones	

↑  
Enable clear display if this bit is 1

If the  $C_F$  bit is asserted ( $C_F = 1$ ), the FIFO status is cleared and the interrupt output line is reset.

$C_A$ , the clear all bit, has the combined effect of  $C_D$  and  $C_F$ ; it uses the  $C_D$  clearing code on the Display RAM and also clears FIFO status. Furthermore, it resynchronizes the internal timing chain.

**Read FIFO/Sensor RAM**

**Code :**

0	1	0	AI	X	A	A	A
---	---	---	----	---	---	---	---

  
X = Don't care

The CPU sets up the 8279 for a read of the FIFO/Sensor RAM by first writing this command. In the Scan keyboard Mode, the Auto-Increment flag (AI) and the Ram address bits (AAA) are irrelevant.

In the Sensor Matrix Mode, the RAM address bits AAA select one of the 8 rows of the Sensor RAM. If the AI flag is set (AI = 1), each successive read will be from the subsequent row of the sensor RAM.

**End Interrupt/Error Mode Set**

**Code :**

1	1	1	E	X	X	X	X
---	---	---	---	---	---	---	---

  
X = Don't care

For the sensor matrix modes this command lowers the IRQ line and enables further writing into RAM. For the N-Key rollover mode, if the E bit is programmed to 1 the chip will operate in the special Error mode.

**Keyboard/Display Mode Set**

**Code :**

0	0	0	D	D	K	K	K
---	---	---	---	---	---	---	---

DD

0	0	Eight No. of 8-bit character display	-Left entry
0	1	Sixteen No. of 8-bit character display	-Left entry
1	0	Eight No. of 8-bit character display	-Right entry
1	1	Sixteen No. of 8-bit character display	-Right entry

KKK

0	0	0	Encoded Scan Keyboard - 2-Key Lockout
0	0	1	Decoded Scan Keyboard - 2-Key Lockout
0	1	0	Encoded Scan Keyboard - N-Key Rollover
0	1	1	Decoded Scan Keyboard - N-Key Rollover
1	0	0	Encoded Scan Sensor Matrix
1	0	1	Decoded Scan Sensor Matrix
1	1	0	Strobed Input, Encoded Display Scan
1	1	1	Strobed Input, Decoded Display Scan.

**Program Clock**

**Code :**

0	0	1	P	P	P	P	P
---	---	---	---	---	---	---	---

All timing and multiplexing signals for the 8279 are generated by an internal prescaler. This prescaler divides the external clock (pin 3) by a programmable integer. Bits P P P P P determine the value of this integer which ranges from 2 to 31. Choosing a divisor that yields 100 kHz will give the specified scan and debounce times.

**Read Display RAM**

**Code :**

0	1	1	AI	A	A	A	A
---	---	---	----	---	---	---	---

The CPU sets up the 8279 for a read of the Display RAM by first writing this command. The address bits AAAA select one of the 16 rows of the Display RAM. If the AI flag is set (AI=1), this row address will be incremented after each read or write to the Display RAM.

**Fig. 6.59 : 8279 Command word formats.**

The 8279 requires an internal clock frequency of 100 kHz. This can be obtained by dividing the input clock by an internal prescaler. The prescaler can take a value from 2 to 31, which is programmable. The RESET signal sets the 8279 in 16-character display with two-key lockout keyboard mode. Also the reset will set the clock prescaler to 31.

### Programming the 8279

The 8279 can be programmed to perform various functions through eight command words. The formats of the command words and a brief explanation are presented in Fig. 6.64.

#### 6.4.5 Keyboard and Display Interface Using 8279

In a microprocessor-based system, when keyboard and 7-segment LED display are interfaced using ports or latches then the processor has to carry out the following tasks :

- Keyboard scanning
- Key debouncing
- Keycode generation
- Sending display code to LED
- Display refreshing

The above functions have to be performed continuously in specified time intervals. Hence most of the processor time will be utilized for the above task. To overcome this problem, the dedicated Keyboard/Display controller such as INTEL 8279 can be employed in the system. The 8279 provides a hardware solution for keyboard and display interfacing in microprocessor-based system.

When 8279 is employed, the task of the processor is to program the 8279 by sending control words and load the display code in display RAM of 8279. Once 8279 is programmed it takes care of keyboard scanning, debouncing, keycode generation and display refreshing. Whenever 8279 detects a key press, it informs the processor through interrupt so that the processor can read the keycode from FIFO of 8279.

### Interfacing 8279 with 8085 Processor

A typical Hexa keyboard and 7-segment LED display interfacing circuit using 8279 is shown in Fig. 6.60. The circuit can be used in 8085 microprocessor system and consists of 16 numbers of hexa-keys and 6 numbers of 7-segment LEDs. The 7-segment LEDs can be used to display six digit alphanumeric character.

The 8279 can be either memory-mapped or IO-mapped in the system. In the circuit of Fig. 6.60, the 8279 is IO-mapped. The address line  $A_0$  of the system is used as  $A_0$  of 8279. The clock signal for 8279 is obtained by dividing the output clock signal of 8085 by a clock divider circuit.

The chip select signal  $\overline{CS}$ , is obtained from the IO address decoder of the 8085 system. The chip select signals for IO-mapped devices are generated by using a 3-to-8 decoder. Address lines  $A_4$ ,  $A_5$  and  $A_6$  are used as input to decoder. The address line  $A_7$  and the control signal  $IO/\overline{M}$  are used as enable for decoder. The chip select signal  $IOCS-3$  is used to select 8279. The IO address of the internal devices of 8279 are shown in Table-6.24.

The circuit has 6 numbers of 7-segment LEDs and so the 8279 has to be programmed in encoded scan. (Because in decoded scan, only 4 numbers of 7-segment LEDs can be interfaced.) In encoded scan the output of scan lines will be binary count. Therefore an external, 3-to-8 decoder is used to decode the scan lines  $SL_0$ ,  $SL_1$  and  $SL_2$  of 8279 to produce eight scan lines  $S_0$  to  $S_7$ . The decoded scan lines  $S_0$  and  $S_1$  are common for keyboard and display. The decoded scan lines  $S_2$  to  $S_5$  are used only for display and the decoded scan lines  $S_6$  and  $S_7$  are not used in the system.



*The **McGraw-Hill** Companies*

**TABLE - 6.20 : IO ADDRESSES OF 8279**

Internal device	Binary address								Hexa address
	Decoder input and enable				Input to address line of 8279				
	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	
Data register	0	0	1	1	x	x	x	0	30
Control register	0	0	1	1	x	x	x	1	31

*Note : Don't care "x" is considered as zero.*

The common cathode LEDs, LT543 are used in circuit shown in Fig. 6.60. The corresponding segments of anodes are connected to a common line to form a bus and this bus can be called as segment bus (i.e., segment "a" of all 7-segment LEDs are connected to a common line, similarly segment "b" and so on).

Anode and Cathode drivers are provided to take care of the current requirement of LEDs. The pnp transistors, BC158 are used as driver transistors. The anode drivers are called segment drivers and cathode drivers are called digit drivers.

The 8279 outputs the display code for one digit through its output lines (OUT A<sub>0</sub> to OUT A<sub>3</sub> and OUT B<sub>0</sub> to OUT B<sub>3</sub>) and sends a scan code through, SL<sub>0</sub>-SL<sub>3</sub>. The display code is inverted by segment drivers and sent to segment bus. The scan code is decoded by the decoder and turns ON the corresponding digit driver. Now one digit of the display character is displayed. After a small interval (10 millisecond, typical), the display is turned OFF (i.e., display is blanked) and the above process is repeated for the next digit. Thus, multiplexed display is performed by 8279.

*Note : Since anode drivers invert the display code, the complement of the data required to turn ON a common cathode LED should be loaded in display RAM of 8279.*

The keyboard matrix is formed using the return lines, RL<sub>0</sub> to RL<sub>7</sub> of 8279 as columns and decoded scan lines S<sub>0</sub> and S<sub>1</sub> as rows. A hexa key is placed at the crossing point of each row and column. A key press will short the row and column. Normally the column and row line will be **high** (i.e., the 8279 will tie the return line as **high** and decoder will tie the scan line as **high**). During scanning the 8279 will output binary count on SL<sub>0</sub> to SL<sub>3</sub>, which is decoded by decoder to make a row as zero. When a row is zero the 8279 reads the columns. If there is a key press then the corresponding column will be zero.

If 8279 detects a key press then it waits for debounce time and again read the columns to generate keycode. In encoded scan keyboard mode, the 8279 stores an 8-bit code for each valid key press. The keycode consist of the binary value of the column and row in which the key is found and the status of shift and control key. The format of the code entered in FIFO RAM is shown in Fig. 6.58. After a scan time, the next row is made zero and the above process is repeated and so on. Thus 8279 continuously scans the keyboard.

### 6.5 PROGRAMMABLE TIMER - INTEL 8254

When a processor has to perform time-based activities, there are two methods to maintain the timings of operations. In the first method the processor can execute a delay subroutine. In this method, the delay subroutine will load a count value in one of the register of the processor and starts decrementing the count value. After every decrement operation, the zero flag is checked to verify whether the count has reached zero or not. If the count has reached zero the delay subroutine is terminated. Now the desired time will be elapsed and the processor can perform the desired time-based task. In this method the time is estimated in terms of processor clock periods needed to execute the delay subroutine.

In the second method, an external timer can maintain the timings and interrupt the processor at periodical intervals. In the first method, the processor time is wasted by simply decrementing a register. But in the second method, the processor time can be efficiently utilized, because the processor can perform other tasks in between timer interrupts. One of the programmable external timer device is 8254 developed by INTEL. The INTEL 8254 timer has three independent counters. In each counter a count value can be loaded and the count value can be decremented by applying a clock signal. At the end of count, each counter will generate an output which can be used as interrupt to processor to initiate the time-based activity. Some of the applications of programmable timer are given below :

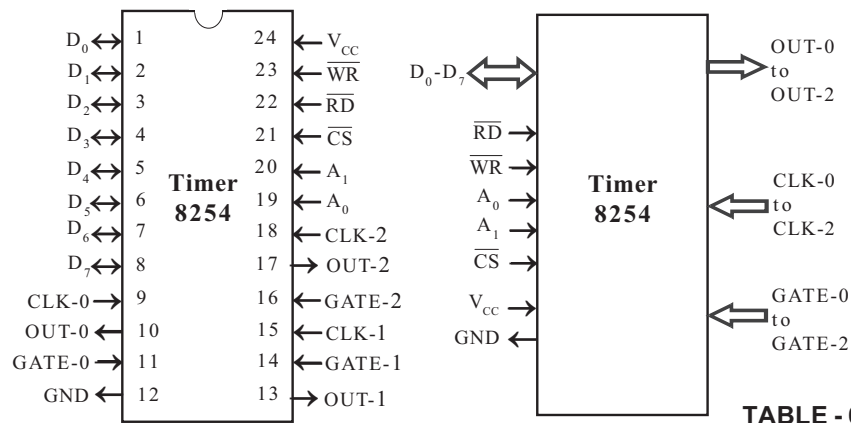
1. The timer can interrupt a time-sharing operating system at specified intervals so that it can switch programs.
2. The timer can send timing signals at periodic intervals to IO devices. (For eg., start of conversion signal to ADC)
3. The timer can be used as baud rate generator. (For example, the timer can be used as a clock divider to divide the processor clock to desired frequency for TxC and RxC of USART-8251A.)
4. The timer can be used to measure the time between external events.
5. The timer can be used as an external event counter to count repetitive external operations and inform the count value to the processor.
6. The timer can be used to initiate an activity through interrupt after a programmed number of external events have occurred.

The 8254 is a 24-pin IC packed in DIP and requires a single +5-V supply. The pin configuration of 8254 is shown in Fig. 6.61. The functional block diagram of 8254 is shown in Fig. 6.62.

The 8254 has three independent 16-bit counters, which can be programmed to work in any one of the possible six modes. Each counter has a clock input, gate input and counter output. To operate a counter, a count value has to be loaded in count register, gate should be tied **high** and a clock signal should be applied through clock input. The counter counts by decrementing the count value by one in each cycle of clock signal and generates an output depending on the mode of operation. The maximum input clock frequency for 8254 is 10 MHz.

*Note : Another timer released by INTEL is 8253 which is the low clock version of 8254. The maximum input clock frequency to 8253 is 2.6 MHz. The 8253 and 8254 are pin to pin compatible and functionally same except the clock frequency.*





Pin	Description
$D_0 - D_7$	Bidirectional data lines
$\overline{CS}$	Chip select
$\overline{RD}$	Read control
$\overline{WR}$	Write control
$A_0, A_1$	Internal address
CLK-0 to CLK-2	Clock input to counters
GATE-0 to GATE-2	Gate control input to counters
OUT-0 to OUT-2	Output of counters

Fig. 6.61 : Pin configuration of an 8254 timer.

TABLE - 6.21 : INTERNAL ADDRESSES OF 8254

Internal address		Device selected
$A_1$	$A_0$	
0	0	Counter-0
0	1	Counter-1
1	0	Counter-2
1	1	Control Register

The 8254 has eight data lines which can be used to communicate with the processor. The control words and count values are written into 8254 registers through data bus buffer. The  $\overline{CS}$  is used to select the chip. The address lines  $A_0$  and  $A_1$  are used to select any one of the four internal devices as shown in Table-6.21. The control signals  $\overline{RD}$  and  $\overline{WR}$  are used by the processor to perform read/write operation. The processor can read the count value in the count register with/without stopping the counter any time.

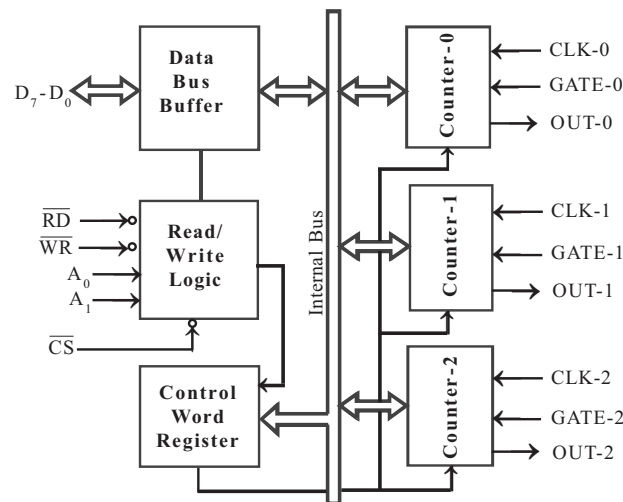


Fig. 6.62 : Functional block diagram of an 8254 timer.

### Interfacing 8254 with 8085 Processor

A simple schematic for interfacing the 8254 with 8085 processor is shown in Fig. 6.63. The 8254 can be either memory-mapped or IO-mapped in the system. In the schematic shown in Fig. 6.63, the 8254 is IO-mapped in the system. The chip select signals for IO-mapped devices are generated by using a 3-to-8 decoder. The address lines  $A_4$ ,  $A_5$  and  $A_6$  are decoded to generate eight chip select signals (IOCS-0 to IOCS-7) and in this, the chip select IOCS-5 is used to select 8254. The address line  $A_7$  and the control signal  $\text{IO}/\overline{\text{M}}$  are used to enable the decoder.

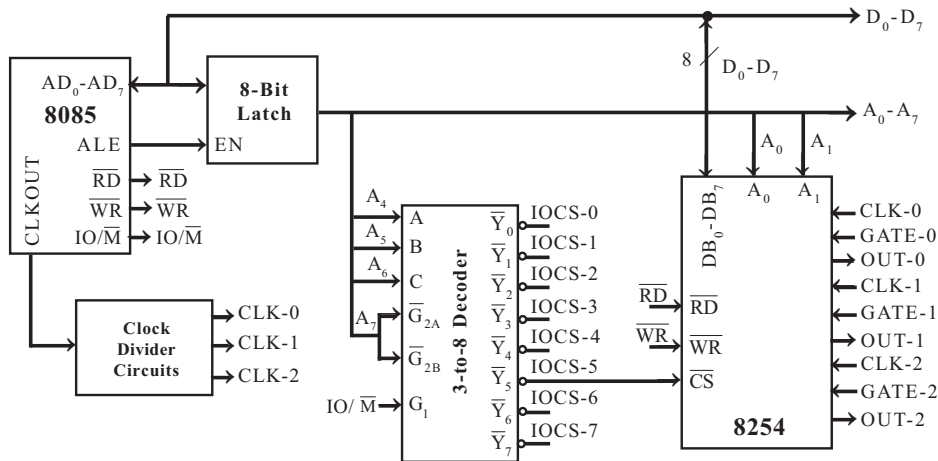


Fig. 6.63 : Interfacing of 8254 with 8085 processor.

The address lines  $A_0$  and  $A_1$  of 8085 are connected to  $A_0$  and  $A_1$  of 8254 to provide the internal addresses. The IO addresses allotted to the internal devices of 8254 are listed in Table-6.22. The data lines  $D_0$ - $D_7$ ,  $\overline{\text{RD}}$  and  $\overline{\text{WR}}$  signals of 8254 are connected to  $D_0$ - $D_7$ ,  $\overline{\text{RD}}$  and  $\overline{\text{WR}}$  of the processor respectively to achieve parallel data transfer.

TABLE- 6.22 : IO ADDRESSES OF 8254

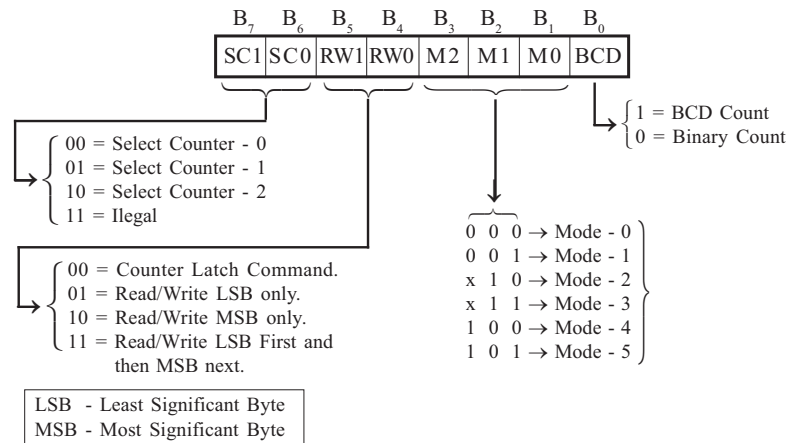
Internal device	Binary address								Hexa address
	Decoder input and enable				Input to address pins of 8254				
	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	
Counter - 0	0	1	0	1	x	x	0	0	50
Counter - 1	0	1	0	1	x	x	0	1	51
Counter - 2	0	1	0	1	x	x	1	0	52
Control Register	0	1	0	1	x	x	1	1	53

Note : Don't care "x" is considered as zero.

The clock signals required for the counters can be obtained either from processor clock output or from an external clock source. The clock signal from 8085 can also be divided to lower values by using clock divider circuits and then applied to clock input of counters.

### Programming 8254

Each counter of 8254 can be individually programmed by writing a control word followed by the count value. The format of control word is shown in Fig. 6.64.



**Fig. 6.64 :** Format of control word for timer 8254.

The bit  $B_0$  (BCD) of control word is used to select BCD or binary count and the bits  $B_1$  to  $B_3$  ( $M_0$ ,  $M_1$  and  $M_2$ ) are used to select the mode of operation for the counter specified by bits  $B_6$  and  $B_7$  of control word. Please remember that for each counter a separate control word has to be sent to same control register address. The 8254 identifies the control word for a particular counter from bits  $B_6$  and  $B_7$  of control word.

The bits  $B_4$  and  $B_5$  are used for read/write command. These bits are programmed for reading/writing the 16-bit count value in proper order. If the count value is read without stopping the counter, then the count value may change between reading the LSB and MSB. To avoid this, the counter latch command can be used to latch the count value to an internal latch available at the output of each counter before the read operation.

Alternatively, a separate read-back control word is available for latching the count value in 8254. (This control word is not available in 8253.) The format of read-back control word of 8254 is shown in Fig. 6.65. This control word has to be sent to the same control register address before read operation to latch the count value. The control register identifies this control word from the value of bits  $B_6$  and  $B_7$ .

The read-back control word can be used to latch one or all the counters by sending a single control word. This control word is also used to latch the status register to the output latch of the counters, so that the status registers can be read by using the respective counter address. At any one time we can latch either the count value by programming the bit  $B_5$  as zero or latch the status register by programming the bit  $B_4$  as zero.

The format of status register of each counter is shown in Fig. 6.66. The status word of a counter can be read to check the programmed status of the counter and also to verify whether the count value has reached terminal count i.e., zero or not.

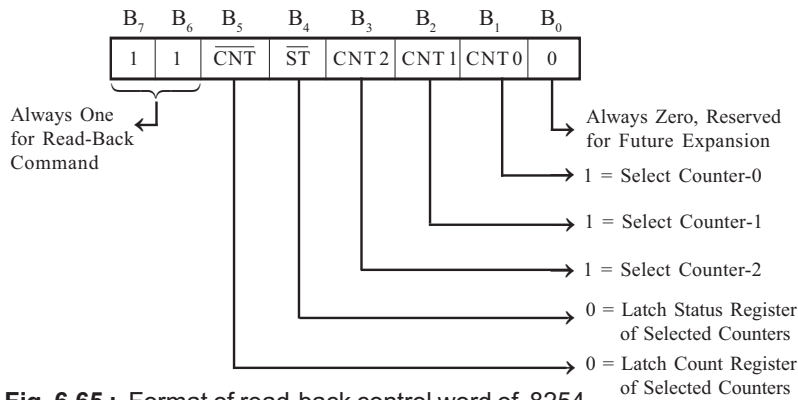


Fig. 6.65 : Format of read-back control word of 8254.

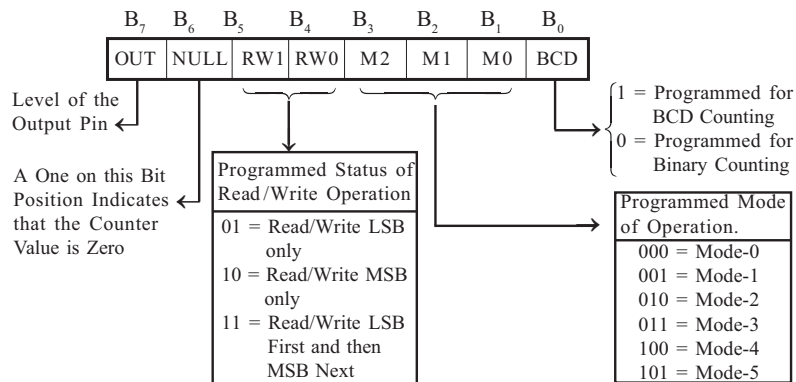


Fig. 6.66 : Format of status word of each counter of 8254.

### Operating Modes of 8254

The 8254 has six modes of operation. Each counter of 8254 can be independently programmed to work in one of the possible six operating modes. The six modes are as follows :

- Mode -0 → Interrupt on terminal count.
- Mode -1 → Hardware retriggerable one shot.
- Mode -2 → Rate generator or Timed interrupt generator.
- Mode -3 → Square wave mode.
- Mode -4 → Software triggered strobe.
- Mode -5 → Hardware triggered strobe.

The initialization procedure for each mode is almost same, but the output of each mode will be different. To initialize a counter, the following steps are necessary :

1. Write a control word into control register.
2. Write a count value in count register.

The writing of count value depends on the control word. There may be three possible choices.

They are as follows :

1. If the control word is framed for writing LSB only then write LSB alone.
2. If the control word is framed for writing MSB only then write MSB alone.
3. If the control word is framed for writing LSB first and MSB next, then write LSB first and write MSB next.

*Note : LSB - Least Significant Byte (Low order byte).  
MSB - Most Significant Byte (High order byte).*

In all the modes the GATE signal acts as a control signal to start, stop or maintain the counting process. In modes 0, 2, 3 and 4 once the count value is loaded in the counter, the timer starts decrementing the count value if the GATE is **high**. Whenever the GATE signal goes **low**, the counter stops counting and will resume counting only when the GATE is made **high** again.

In modes 1 and 5 the GATE acts as a triggering pulse. In these modes, the count value is loaded in the counter and it starts the decrementing process only when the GATE signal makes a low-to-high transition (i.e. the count process is initiated only on the rising edge of the GATE signal). In modes 1 and 5 the GATE signal need not remain **high** (after initiation), to maintain the counting process.

A brief description about each mode of operation is presented here. In the following discussions it is assumed that the counter is initialized for binary count, by writing only LSB of the count.

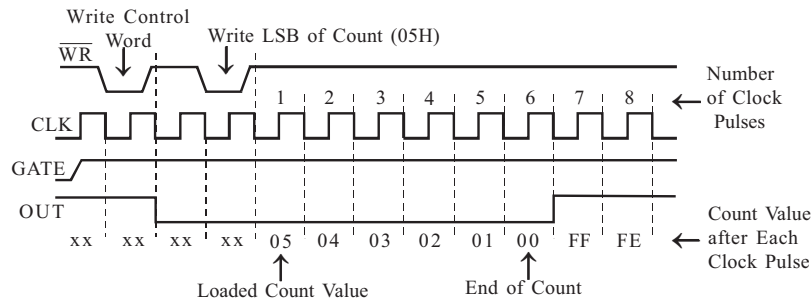
#### **Mode-0 : Interrupt on terminal count**

In Mode-0 operation when a count value is loaded in a counter it starts decrementing the count value by one for each input clock pulse (provided the GATE is **high**) and asserts the output as **high** when the count value is zero. (i.e. on terminal count). This low-to-high transition of the counter output can be used as an interrupt to the processor to initiate any activity. In mode-0 the 8254 will count as long as GATE is **high**. Whenever the GATE signal goes **low** the counter stops counting and will resume counting only when the GATE is made **high** again.

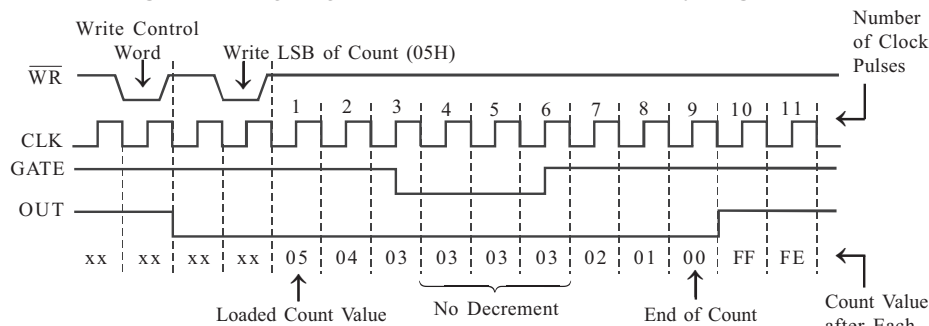
The timing diagram for mode-0 operation is shown in Fig. 6.67. In the timing diagram of Fig. 6.67(a) initially the counter output remains **high** and it is assumed that the GATE is always **high**. The processor writes the control word and count value using write control signal ( $\overline{WR}$ ). Once the control word is written into control register the output goes **low**. After the write operation of count value by the processor, the 8254 requires one clock pulse to load the count value in the respective count register. Therefore, in the first clock pulse after  $\overline{WR}$  goes **high**, the 8254 loads the count value in the count register and in each subsequent clock pulse, the count value is decremented by one. When the count value becomes zero, the output of the counter is asserted **high**.

Figure 6.67 shows the timing diagram for a count value of 05H initially loaded in the count register. Here, the output goes **high** after 6 ( $5+1 = 6$ ) clock pulse. In general, if a count value of N is loaded in the count register then the output goes **high** after N+1 clock pulses. Please remember that the counter continues to decrement the count value even after zero ( $00 \rightarrow FF$  ;  $FF \rightarrow FE$  and so on) as long as GATE is **high** and clock signal is supplied. The output of the counter remains **high** until a new count or command is send to the counter.

In the timing diagram shown in Fig. 6.67(b), the GATE is made **low** for a small period before the terminal count value. It is observed that, in this period, the count value is not decremented and previous value is maintained as such. The counter resumes operation only when the GATE is made **high** again.



**Fig. a :** Timing diagram of Mode-0 with GATE always high.



**Fig. b :** Example diagram of mode-0 when the GATE is made low for small duration before the terminal count.

**Note :** "xx" represents undefined count value.

**Fig. 6.67 :** Timing diagram of mode-0 of 8254.

### Mode-1 : Hardware retriggerable one shot

In mode-1 the counter functions as a retriggerable monostable multivibrator (one shot). In this mode the output will be **high** once the control word is sent to control register. The GATE acts as a trigger pulse to start the count process. When a low-to-high transition of GATE signal occurs, the count value is loaded in the counter and the count is decremented by one for each clock pulse. When the count value is loaded in the counter the OUTPUT goes **low** and it becomes **high** when the count value is zero. Therefore the mode-1 produces a logic **low** pulse output whose width is equal to the duration of the count.

The timing diagram of mode-1 operation is shown in Fig. 6.68. The processor writes the control word and count value using  $\overline{WR}$  control signal. Initially the output is assumed to be **high**. Even if it is **low**, it is asserted **high**, once the control word is written into control register. Initially the GATE can be **high** or **low**. If the GATE is **low** then it is made **high** to initiate the count process. If it is **high** then it is made **low** and after a small delay it is made again **high**, because the count process is initiated only after a low-to-high transition of GATE. After the trigger pulse (i.e., low-to-high transition) the gate can remain either in **high** state or in **low** state.

The first clock pulse after a low-to-high transition of gate is used to load the count value in the counter and for each subsequent clock the count value is decremented by one. Once the count value is loaded to the counter the output is asserted **low** and at the end of count, when the count

value is zero, the output is asserted **high**. In the timing diagram shown in Fig. 6.68(a), a count value of 05H is loaded and so the output remains **low** for 5 clock periods. In general if a count value of N is loaded in the counter then the output will remain **low** for N clock periods. Therefore the output low pulse width will be N times the clock period.

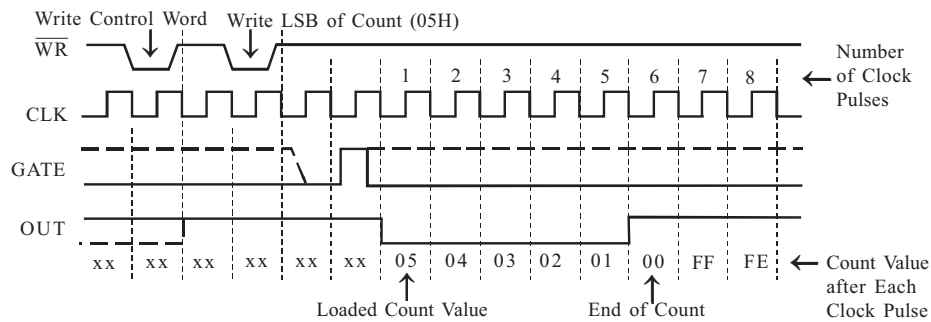


Fig. a : Timing diagram of mode-1.

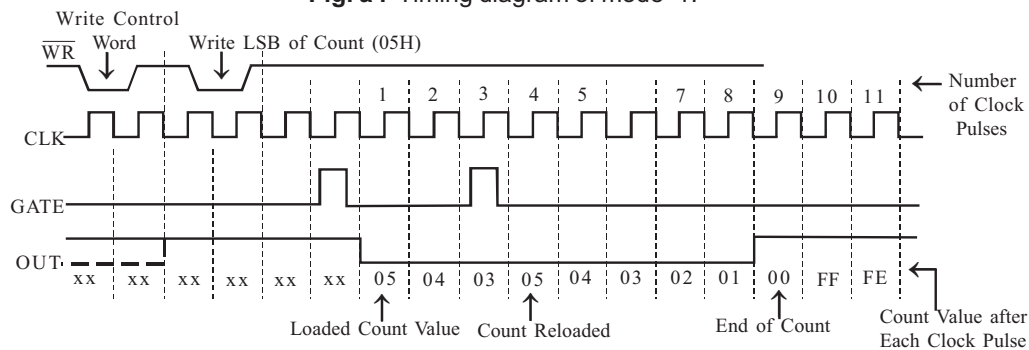


Fig. b : Timing diagram of mode-1 with GATE retriggering before end of count.

Note : "xx" represents undefined count value.

Fig. 6.68 : Timing diagram of mode-1 of 8254.

In the timing diagram of Fig. 6.68(b), the GATE is retriggered before the end of count. In this case, the original count value is reloaded again in the clock pulse after gate retriggering and the count value is decremented by one in each subsequent clock pulse.

### Mode-2 : Rate generator or timed interrupt generator

Mode-2 is used to generate a periodic low pulse of width equal to one clock period. If a count value of N is loaded in the counter then the output will go **low** once in N clock periods. Therefore, the frequency of low pulse generated will be equal to the input clock frequency divided by N. For Mode-2 operation GATE should be always **high**.

The timing diagram of mode-2 operation is shown in Fig. 6.69. The processor writes the control word and count value using  $\overline{WR}$  control signal. Initially the output is assumed to be **high**. Even if it **low**, it is asserted **high**, once the control word is written into control register. The GATE input is permanently tied to logic **high**. In the first clock pulse, after the  $\overline{WR}$  signal goes **high**, the count value is loaded to the counter and the count value is decremented by one for each subsequent clock pulse.

Initially the output is **high**. When the count reaches one, the output is asserted **low**. In the next clock pulse the output is asserted **high** and the original count value is reloaded. In the subsequent clock pulses the count value is decremented. The above process is repeated again and again until a next command by the processor. In the timing diagram shown in Fig. 6.69, a count value of 03H is loaded in the counter. In a total period of 3 clock periods, the output goes **low** for one clock period. If the gate is made **low** at any time during the count process, the counter will stop the operation and resumes the counting only when the gate is made **high** again.

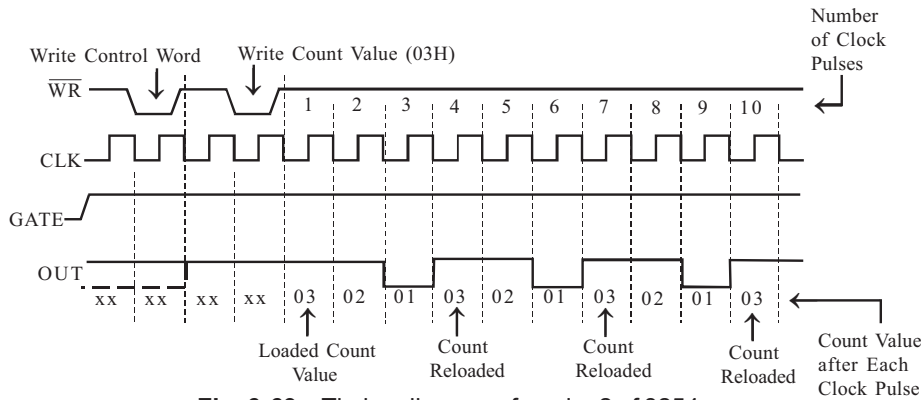


Fig. 6.69 : Timing diagram of mode-2 of 8254.

### Mode-3 : Square wave mode

In mode-3, the counter generates a square wave at the output pin. The frequency of the square wave will be given by the frequency of input clock signal divided by the count value loaded in the count register. If the count value  $N$  is even number then the output will be alternatively **high** for  $\frac{N}{2}$  clock periods and **low** for  $\frac{N}{2}$  clock periods. If the count value is odd number then the output will be alternatively **high** for  $\frac{N+1}{2}$  clock periods and **low** for  $\frac{N-1}{2}$  clock periods (i.e., when the count value is odd, then the output **high** period will be more than low period by one clock period). The timing diagram of Mode-3 is shown in Fig. 6.70.

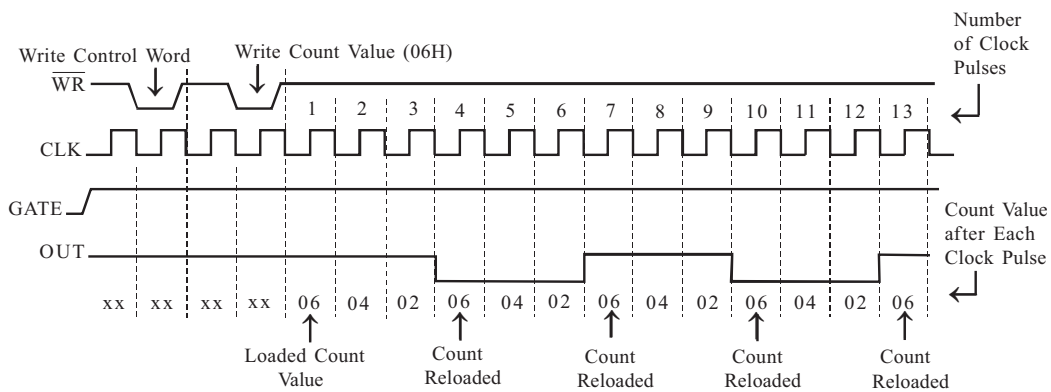


Fig. 6.70 : Timing diagram of mode-3 of 8254.



In the timing diagram shown in Fig. 6.70, a count value of 06H is loaded in the counter. The count value is loaded in the counter in the first clock pulse after  $\overline{WR}$  signal goes **high**. Then for each subsequent clock pulse the count is decremented by two. When the count value reaches two, then in the next clock pulse, the output is asserted **low** and original/initial count is reloaded in the counter and for each subsequent clock pulse the count is decremented by two. When the count value reaches two then in the next clock pulse the output is asserted **high** and the original/initial count is reloaded and the above process is repeated again and again.

In the output waveform generated on the output pin of the counter, the high period and low period are equal to three clock periods. The frequency of waveform generated is given by the clock signal divided by six, because six clock periods are required to generate one cycle of output wave. Throughout the mode-3 operation the GATE input signal should be maintained as **high**. If it is made **low** during count process then the counter stops counting and resumes the operation only after the GATE is made **high**.

#### Mode-4 : Software triggered strobe

Mode-4 is used to generate a single logic **low** pulse after a delay. In this mode when a count value, N is loaded in the counter, a logic **low** pulse of width equal to one clock period is generated in the (N+1)<sup>th</sup> clock pulse. Here, the delay time is N clock periods. This signal is often used as strobe signal in parallel data transfer scheme. The mode-4 is called software triggered strobe because the counter starts its operation once the count value is written into count register by a software instruction. However, the GATE input signal should remain **high** throughout the mode-4 operation.

The timing diagram of mode-4 operation is shown in Fig. 6.71. The GATE is permanently tied to **high**. The processor writes the control word and count value using write control signal. In the first clock pulse after  $\overline{WR}$  signal goes **high**, the count value is loaded in the counter and in each subsequent clock pulse the count value is decremented by one. When the count value reaches zero the output is asserted **low** for one clock period and then it is made **high**.

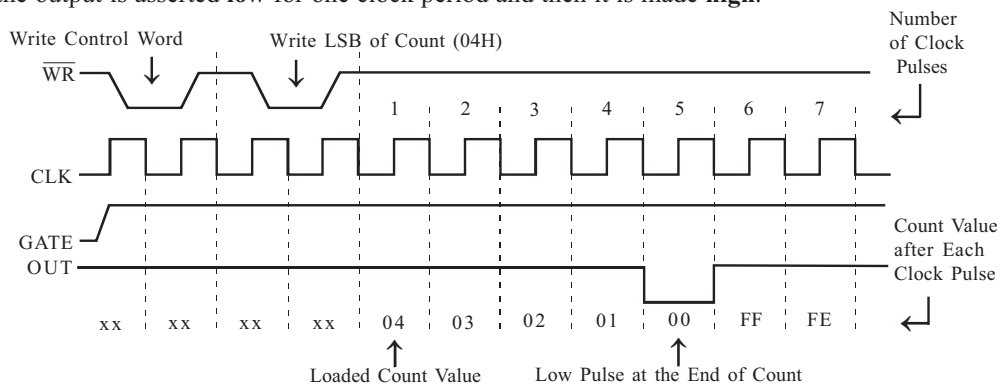
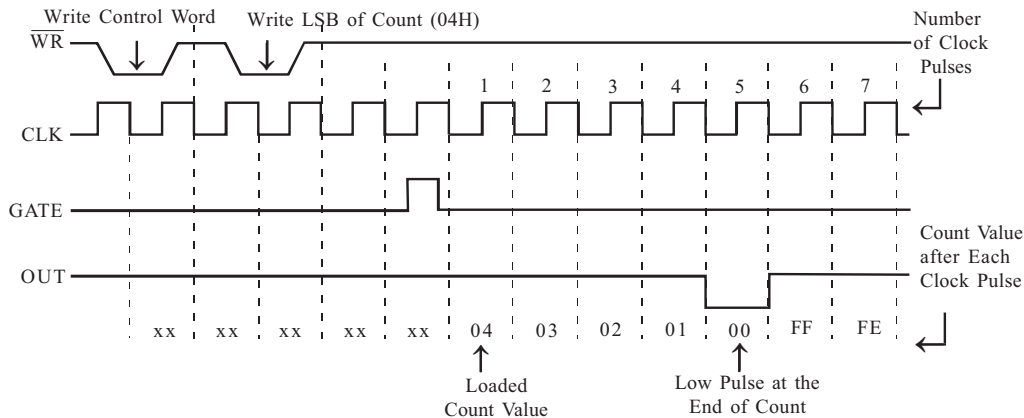


Fig. 6.71 : Timing diagram of mode-4 operation of 8254.

Here a count value of 04H is loaded in the counter. Initially the output remains **high** and in the fifth clock pulse the output goes **low** for one clock period. In mode-4 operation if the GATE is made **low** during count process then the counter stops counting and resumes the operation only when the GATE is made **high**.

**Mode-5 : Hardware triggered strobe**

Mode-5 is the same as that of mode-4, except that the counter is initiated by a low-to-high transition of the GATE signal. In mode-4 the counter will start decrementing the count value immediately after the write operation of count value by the processor. But in mode-5, the counter will wait for a low-to-high transition of GATE signal after the write operation of count value by the processor.



**Fig. 6.72 :** Timing diagram of mode-5 operation of 8254.

The timing diagram of mode-5 operation is shown in Fig. 6.72. In the first clock pulse after a low-to-high transition of GATE, the count value is loaded in the counter and for each subsequent clock pulse the count value is decremented by one. When the count value reaches zero the output is asserted **low** for one clock period and then it is made **high**. Here, a count value of 04H is loaded in the counter. Initially the output remains **high** and the counter waits for a low-to-high transition of the GATE signal. In the fifth clock pulse after a low-to-high transition of GATE signal, the output goes **low** for one clock period.

In mode-5 operation if the gate signal makes another low-to-high transition (i.e., retrigged) before the end of count. Then the original count value is reloaded in the clock pulse after gate retrigging and count value is decremented by one in each subsequent clock pulse.

## 6.6 DAC INTERFACE

In many applications, the microprocessor has to produce analog signals for controlling certain analog devices. Basically the microprocessor system can produce only digital signals. In order to convert the digital signal to analog signal a **Digital-to-Analog Converter (DAC)** has to be employed.

The DAC will accept a digital (binary) input and convert to analog voltage or current. Every DAC will have "n" input lines and an analog output. The DAC requires a reference analog voltage ( $V_{ref}$ ) or current ( $I_{ref}$ ) source. The smallest possible analog value that can be represented by the n-bit binary code is called resolution. The resolution of DAC with n-bit binary input is  $\frac{1}{2^n}$  of reference analog value. Every analog output will be a multiple of the resolution. In some converters the input reference analog signal will be multiplied or divided by a constant to get full scale value. In this case the resolution will be  $\frac{1}{2^n}$  of full scale value.

For example, consider an 8-bit DAC with reference analog voltage of 5 volts. Now the resolution of the DAC is  $(1/2^8) \times 5$  volts. The 8-bit digital input can take,  $2^8 = 256$  different values. The analog values for all possible digital input are as shown in Table-6.23.

**TABLE - 6.23**

Digital input	Analog output
0000 0000	$\frac{0}{2^8} \times 5$ Volts
0000 0001	$\frac{1}{2^8} \times 5$ Volts
0000 0010	$\frac{2}{2^8} \times 5$ Volts
0000 0011	$\frac{3}{2^8} \times 5$ Volts
1111 1111	$\frac{255}{2^8} \times 5$ Volts

The maximum input digital signal will have an analog value which is equal to reference analog value minus resolution. The digital-to-analog converters can be broadly classified into three categories, and they are current output, voltage output and multiplying type DAC. The current output DAC provides an analog current as output signal. In voltage output DAC, the analog current signal is internally converted to voltage signal.

In multiplying type DAC the output is given by the product of the input signal and the reference source and the product is linear over a broad range. Basically, there is not much difference between these three types and any DAC can be viewed as multiplying DAC.

The basic components of a DAC are resistive network with appropriate values, switches, a reference source and a current to voltage converter as shown in Fig. 6.73.

The switches in the circuit of Fig. 6.73 can be transistors which connect the resistance either to ground or  $V_{ref}$ . The resistors are connected in such a way that for any possible binary input, the total current  $I_T$  is in binary proportion. The operational amplifier converts the current  $I_T$  to a voltage signal  $V_0$ , which can be calculated from the following equation :

$$V_0 = V_{ref} \frac{R_f}{R} \left( \frac{D_2}{2^1} + \frac{D_1}{2^2} + \frac{D_0}{2^3} \right)$$

The circuit of Fig. 6.73 can be modified as 8-bit DAC by increasing the number of R/2R ladder. For an 8-bit DAC the output voltage is given by,

$$V_0 = V_{ref} \frac{R_f}{R} \left( \frac{D_7}{2^1} + \frac{D_6}{2^2} + \frac{D_5}{2^3} + \frac{D_4}{2^4} + \frac{D_3}{2^5} + \frac{D_2}{2^6} + \frac{D_1}{2^7} + \frac{D_0}{2^8} \right)$$

The time required for converting the digital signal to analog signal is called conversion time. It depends on the response time of the switching transistors and the output amplifier. If the DAC is interfaced to microprocessor then the digital data (signal) should remain at the input of DAC, until the conversion is complete. Hence, to hold the data a latch is provided at the input of DAC.

The Digital-to-Analog converters compatible to microprocessors are available with or without internal latch and I to V converting amplifier. The AD558 of Analog Device is an example of 8-bit DAC with an internal latch and I to V converting amplifier. The output of AD558 is an analog voltage signal.

The AD558 can be directly interfaced to 8085 microprocessor bus and it requires only two control signals : Chip Select ( $\overline{CS}$ ) and Chip Enable ( $\overline{CE}$ ). [No handshake signals are necessary for interfacing a DAC. The time between loading two digital data to DAC is controlled by software time delay.]

The DAC0800 of National Semiconductor Corporation is an example of 8-bit DAC without internal latch and I to V converting amplifier. The DAC0800 can be interfaced to microprocessor using either a port device or a latch.

### 6.6.1 DAC0800

The DAC0800 is an 8-bit, high speed, current output DAC with a typical settling time (conversion time) of 100 ns. It produces complementary current output which can be converted to voltage by using simple resistor load.

The DAC0800 is available as a 16-pin IC in DIP. The pin configuration of DAC0800 is shown in Fig. 6.74 and the internal block diagram of DAC0800 is shown in Fig. 6.75.

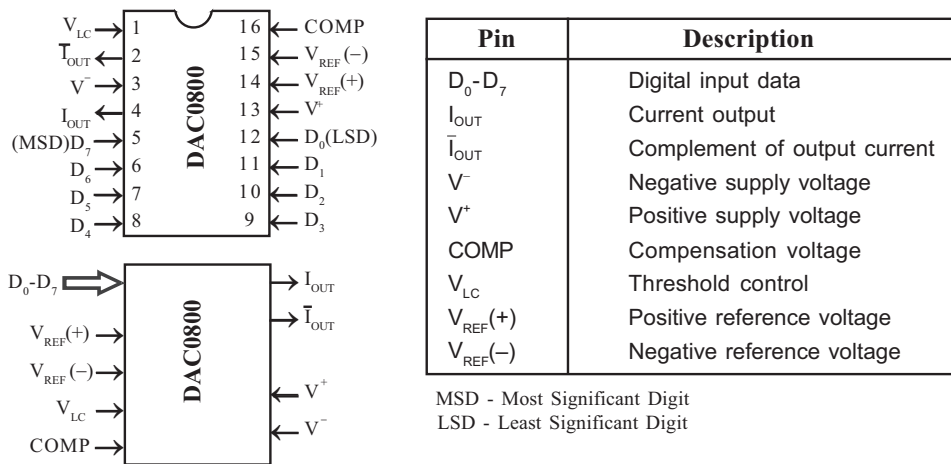


Fig. 6.74 : Pin description of DAC0800.

The DAC0800 requires a positive and a negative supply voltage in the range of  $\pm 5$ -V to  $\pm 18$ -V. It can be directly interfaced with TTL, CMOS, PMOS and other logic families. For TTL input, the threshold pin should be tied to ground ( $V_{LC} = 0$ -V). The reference voltage and the digital input will decide the analog output current, which can be converted to a voltage by simply connecting a resistor to output terminal or by using an op-amp I to V converter. A typical example of generating a positive voltage output using DAC0800 is shown in Fig. 6.76.

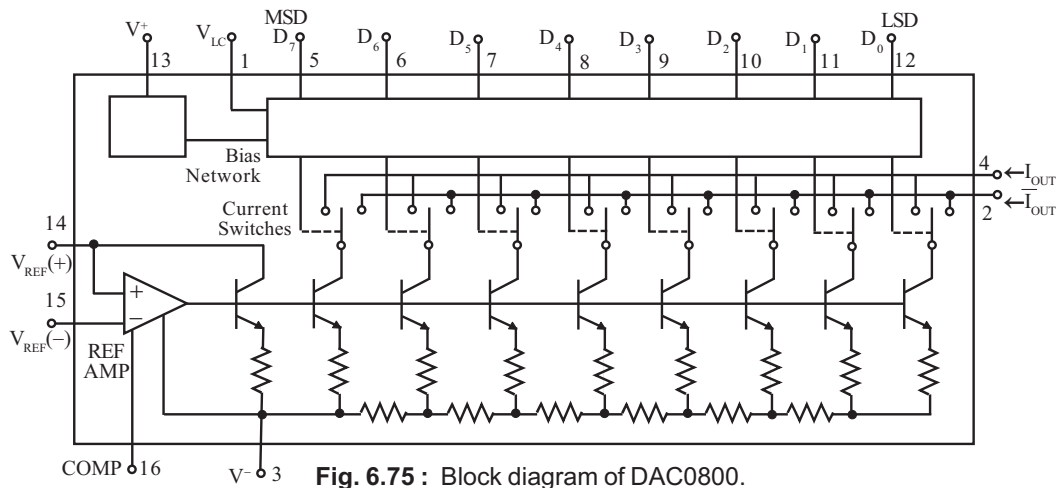
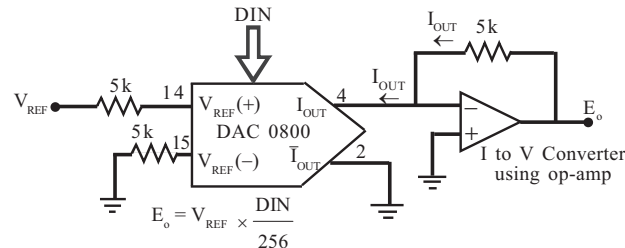


Fig. 6.75 : Block diagram of DAC0800.



where, DIN = Decimal Equivalent of Binary Input

Fig. 6.76 : DAC 0800 with V to I converter to produce positive output voltage.

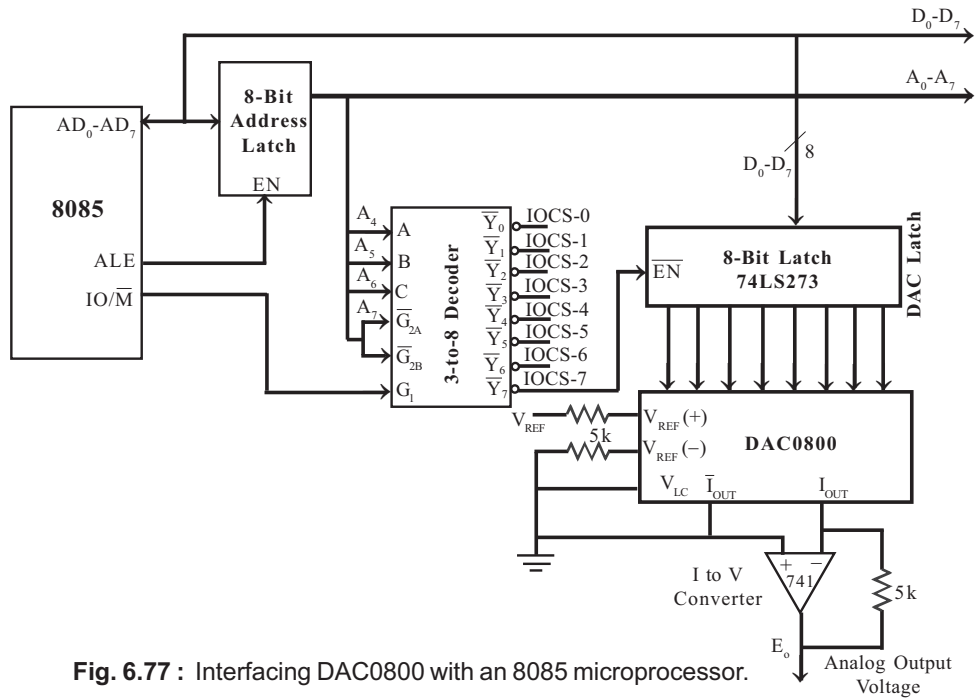
### Interfacing DAC0800 with 8085

The DAC0800 can be interfaced to 8085 system bus by using an 8-bit latch and the latch can be enabled by using one of the chip select signal generated for IO devices. A simple schematic for interfacing DAC0800 with 8085 is shown in Fig. 6.77. In this schematic the DAC0800 is interfaced using an 8-bit latch 74LS273 to the system bus. The 3-to-8 decoder 74LS138 is used to generate chip select signals for IO devices. The address lines  $A_4$ ,  $A_5$  and  $A_6$  are used as input to decoder. The address line  $A_7$  and the control signal  $IO/\bar{M}$  are used as enable for decoder. The decoder will generate eight chip select signals and in this the signal  $IOCS-7$  is used as enable for latch of DAC. The IO address of the DAC is shown in Table-6.24.

In order to convert a digital data to analog value, the processor has to load the data to a latch. The latch will hold the previous data until the next data is loaded. The DAC will take definite time to convert the data. The software should take care of loading successive data only after the conversion time. The DAC 0800 produces a current output, which is converted to voltage output using I to V converter.

TABLE - 6.24 : IO ADDRESS OF DAC LATCH

Device	Binary address								Hexa address
	Decoder input and enable				Unused address lines				
	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	
DAC Latch 74LS273	0	1	1	1	x	x	x	x	70



**Fig. 6.77 :** Interfacing DAC0800 with an 8085 microprocessor.

## 6.7 ADC INTERFACE

In many applications, an analog device has to be interfaced to digital system. But, the digital devices cannot accept the analog signals directly. So, the analog signals are converted to equivalent digital signal (data) using **Analog-to-Digital Converter (ADC)**.

The Analog to Digital (A/D) conversion is the reverse process of Digital to Analog (D/A) conversion. The A/D conversion is also called quantization, in which the analog signal is represented by an equivalent binary data. The analog signals vary continuously and defined for any interval of time. The digital signals (or data) can take only finite values and are defined only for discrete instant of time. If the digital data is represented by n-bit binary then it can have  $2^n$  different values. In A/D conversion the given analog signal has to be divided into steps of  $2^n$  values, and each step is represented by one of the  $2^n$  values.

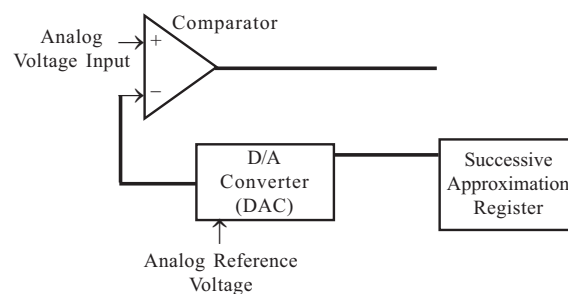
The Analog to Digital Converters can be classified into two groups based on the technique involved for conversion. The first group includes successive-approximation, counter and flash-type converters. The technique involved in these devices is that the given analog signal is compared with internally generated analog signal. The second group includes integrator converters and voltage to frequency converters. In the devices of second group, the given analog signal is converted to time or frequency and the new parameters (time or frequency) is compared with known values to produce digital signal.

The trade-off between the two techniques is based on accuracy vs speed. The successive-approximation and the flash type are fast but generally less accurate than the integrator and the voltage-to-frequency type converters. Also, the flash type is costlier. The successive-approximation type converters are used for high speed conversion and the integrating type converters are used for high accuracy.

The resolution of the converter is the minimum analog value that can be represented by the digital data. If the ADC gives n-bit digital output and the full scale analog input is X volts, then the resolution is  $\frac{1}{2^n} \times X$  volts. In ADC, another critical parameter is conversion time. The conversion time is defined as the total time required to convert an analog signal into its digital equivalent. It depends on the conversion technique and the propagation delay in various circuits.

### Successive-Approximation ADC

A successive approximation ADC consists of D/A converter, successive-approximation register and comparator. Figure 6.78 shows the functional blocks of a typical successive-approximation A/D converter.



The conversion process is initiated by a Start Of Conversion (SOC) signal from the processor to ADC. On receiving the SOC, the control unit of ADC will give a start command to successive-approximation register and it starts generating digital signal by successive-approximation method. The generated digital data is converted to analog signal by D/A converter and then compared with the given analog signal. When the analog signals are equal the comparator output informs the control unit to stop generation of the digital signal. The digital data available at this instant is given as output through output register. Also, the control unit generates a signal to indicate the End Of Conversion (EOC) process to the processor.

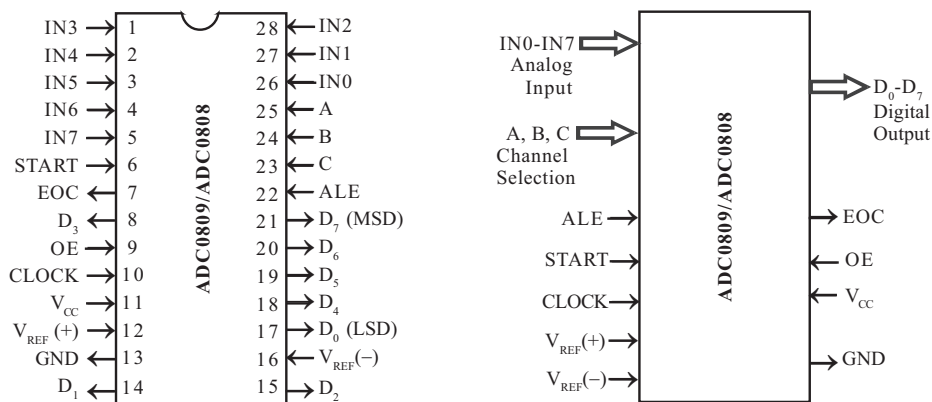
### Successive-approximation method of conversion

In this method the MSD (**M**ost **S**ignificant **D**igit) is first set to "1" and all other digits are reset to "0". The analog signal generated for this digital data is compared with given analog signal. (Initially the comparator output will be **high**. After comparison the output of comparator remains in **high** state if the given analog signal is higher than the generated analog signal. Otherwise, if the

given signal is less than the generated signal, then the output of comparator changes from **high** to **low** state.) If the output state of comparator changes, then the MSD is reset to "0" otherwise it is retained as "1". Then the above process is repeated by setting the next higher order bit to "1". The process is continued for each bit starting from MSD to LSD. (During a process, the higher order bits are the bits determined in earlier steps and the lower order bits are reset to "0".) After one complete cycle through MSD to LSD, the data available on the successive-approximation register will be the digital equivalent of the given analog signal.

### 6.7.1 ADC0809

The ADC0809 is an 8-bit successive-approximation type ADC with an inbuilt 8-channel multiplexer. The ADC0809 is suitable for interface with the 8086 microprocessor. The ADC0809 is available as a 28-pin IC in DIP (Dual In-line Package). The ADC0809 has a total unadjusted error of  $\pm 1$  LSD (Least Significant Digit). The ADC0808 is also same as ADC0809 except the error. The total unadjusted error in ADC0808 is  $\pm \frac{1}{2}$  LSD. The pin configuration of ADC0809/ADC0808 is shown in Fig. 6.79.



LSD = Least Significant Digit, MSD = Most Significant Digit

**Fig. 6.79 : Pin configuration of ADC0809/ADC0808.**

**TABLE - 6.25 : SIGNAL DESCRIPTION OF ADC0809/ADC0808**

Signals	Description
IN0-IN7	Eight single ended analog input to ADC.
A, B, C	3-bit binary input to select one of the eight analog signals for conversion at any one time.
ALE	Address latch enable. Used to latch the 3-bit address input to an internal latch.
START	Start of conversion pulse input. To start ADC process this signal should be asserted <b>high</b> and then <b>low</b> . This signal should remain <b>high</b> for atleast 100 ns.
CLOCK	Clock input and the frequency of clock can be in the range of 10 kHz to 1280 kHz. Typical clock input is 640 kHz.



Table - 6.25 : continued ....

Signals	Description
$V_{REF}(+), V_{REF}(-)$	Reference voltage input. The positive reference voltage can be less than or equal to $V_{cc}$ and the negative reference voltage can be greater than or equal to ground.
$D_0-D_7$	The 8-bit digital output. The reference voltages will decide the mapping of analog input to digital data.
EOC	End of conversion. This signal is asserted <b>high</b> by the ADC to indicate the end of conversion process and it can be used as interrupt signal to processor.
OE	Output buffer Enable. This signal is used to read the digital data from output buffer after a valid EOC.
$V_{cc}$	Power supply, +5-V
GND	Power supply ground, 0-V

The internal block diagram of ADC0809/ADC0808 is shown in Fig. 6.80. The various functional blocks of ADC are 8-channel multiplexer, comparator, 256R resistor ladder, switch tree, successive-approximation register, output buffer, address latch and decoder.

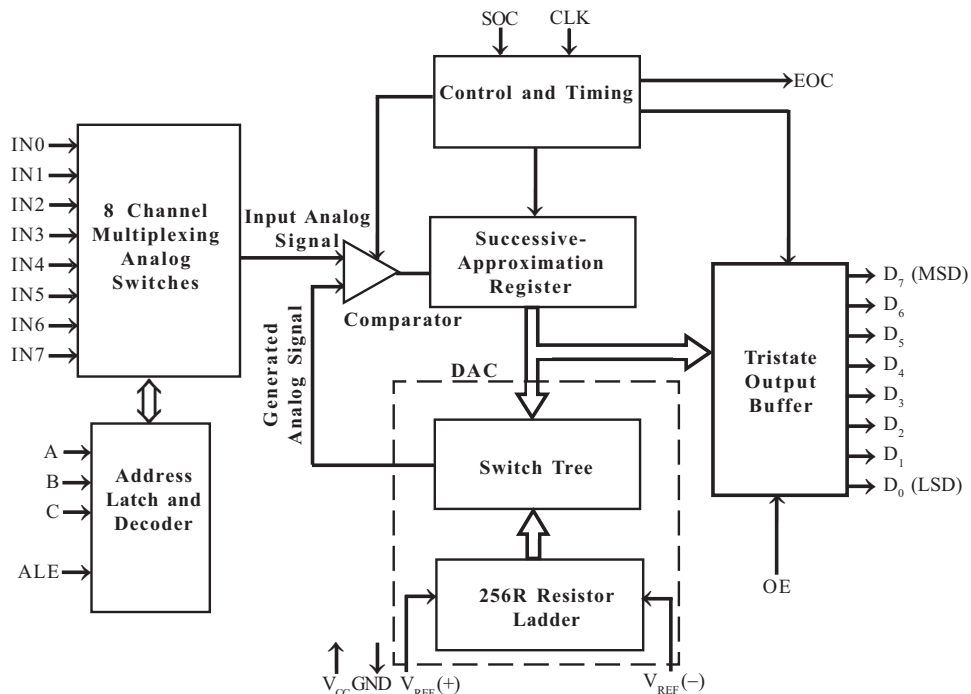


Fig. 6.80 : Functional block diagram of ADC0809/ADC0808.

The 8-channel multiplexer can accept eight analog inputs in the range of 0 to 5-V and allow one by one for conversion depending on the 3-bit address input. The channel selection logic is shown in Table-6.26.

The Successive-Approximation Register (SAR) performs eight iterations to determine the digital code for input value. The SAR is reset on the positive edge of START pulse and start the conversion process on the falling edge of START pulse. A conversion process will be interrupted on receipt of new START pulse. The End-Of-Conversion (EOC) will go **low** between 0 and 8 clock pulses after the positive edge of START pulse. The ADC can be used in continuous conversion mode by tying the EOC output to START input. In this mode an external START pulse should be applied whenever power is switched ON.

The 256R resistor network and the switch tree is shown in Fig. 6.81. The 256R ladder network has been provided instead of conventional R/2R ladder because of its inherent monotonicity, which guarantees no missing digital codes. Also, the 256R resistor network does not cause load variations on the reference voltage.

The comparator in ADC0809/ADC0808 is a chopper-stabilized comparator. It converts the DC input signal into an AC signal and amplifies the AC signal using high gain AC amplifier. Then it converts the AC signal to DC signal. This technique limits the drift component of the amplifier, because the drift is a DC component and it is not amplified/ passed by the AC amplifier. This makes the ADC extremely insensitive to temperature, long term drift and input offset errors.

In ADC conversion process the input analog value is quantized and each quantized analog value will have a unique binary equivalent. The quantization step in ADC0809/ADC0808 is given by,

$$Q_{\text{step}} = \frac{V_{\text{REF}}}{2^8} = \frac{V_{\text{REF}}(+)-V_{\text{REF}}(-)}{256_{10}}$$

The digital data corresponding to an analog input ( $V_{\text{in}}$ ) is given by,

$$\text{Digital data} = \left( \frac{V_{\text{in}}}{Q_{\text{step}}} - 1 \right)_{10}$$

TABLE - 6.26

Address input			Selected channel
C	B	A	
0	0	0	IN0
0	0	1	IN1
0	1	0	IN2
0	1	1	IN3
1	0	0	IN4
1	0	1	IN5
1	1	0	IN6
1	1	1	IN7

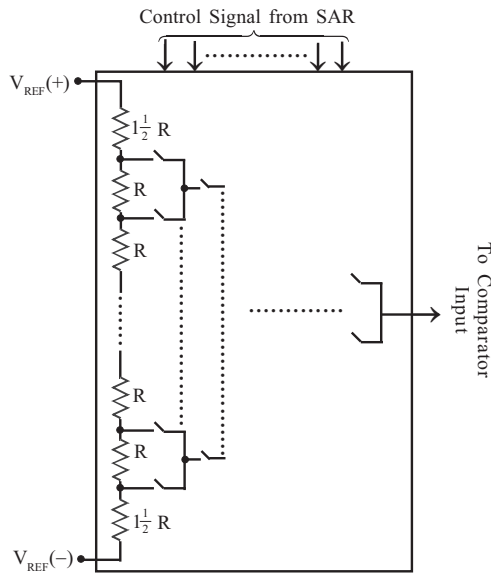


Fig. 6.81 : 256R resistor network and switch tree.

### EXAMPLE 1

Let,  $V_{REF}(+) = 3.84\text{-V}$ ,  $V_{REF}(-) = 0\text{-V}$

$$\therefore Q_{\text{step}} = \frac{V_{\text{REF}}(+)-V_{\text{REF}}(-)}{256_{10}} = \frac{3.84}{256} = 0.015\text{-V} = 15\text{-mV}$$

Let the input analog voltage be 2.56-V. Now the digital data corresponding to 2.56-V is given by,

$$\text{Digital data} = \frac{V_{in}}{Q_{step}} - 1 = \frac{2.56}{0.015} - 1 = 169_{10} = A9_H = 1010\ 1001_2$$

### EXAMPLE 2

Let  $V_{REF}(+) = 5-V$ ,  $V_{REF}(-) = 0-V$

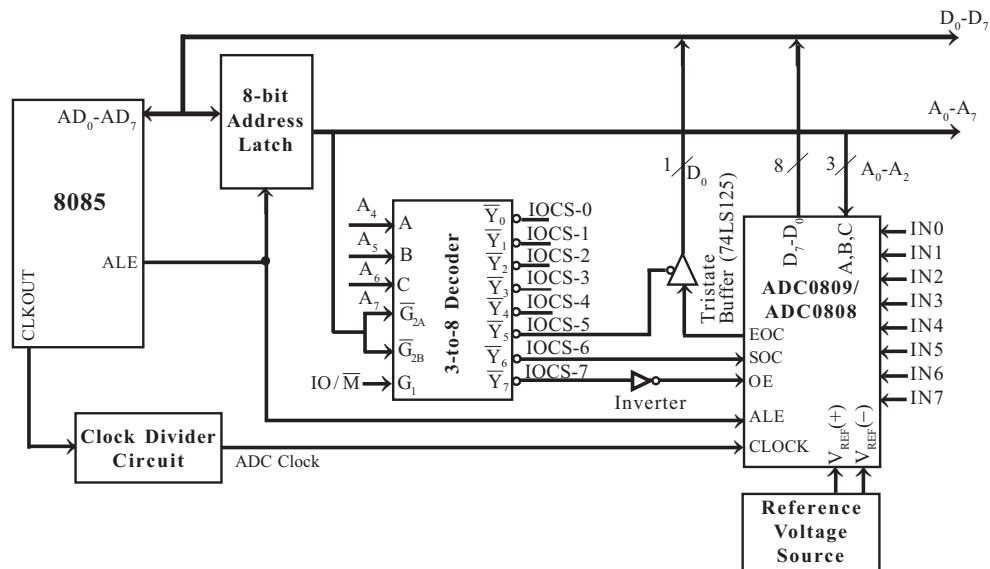
$$\therefore Q_{\text{step}} = \frac{V_{\text{REF}}(+)-V_{\text{REF}}(-)}{256_{10}} = \frac{5}{256} = 0.01953125$$

Let the input analog voltage be 1.25-V. Now the digital data corresponding to 1.25-V is given by,

$$\text{Digital data} = \frac{V_{in}}{Q_{step}} - 1 = \frac{1.25}{0.01953125} - 1 = 63_{10} = 3F_H = 0011\ 1111_2$$

## Interfacing ADC0809 with 8085

A simple schematic for interfacing ADC0809/ADC0808 with 8085 microprocessor is shown in Fig. 6.82. The ADC can be either memory-mapped or IO-mapped in the system. Here, the ADC is IO-mapped in the system. The chip select signals for IO-mapped devices are generated by using a 3-to-8 decoder. The address lines  $A_4$ ,  $A_5$  and  $A_6$  are used as input to decoder. The address line  $A_7$  and the control signal  $\overline{IO/\overline{M}}$  are used as enable for decoder. The decoder generates eight chip select signals (IOCS-0 to IOCS-7) and in this, three chip select signals are used for ADC interface.



**Fig. 6.82 :** Interfacing ADC0809/ADC0808 with 8085 microprocessor.

TABLE - 6.27 : IO ADDRESS OF ADC0809/ADC0808

Operation performed	Binary address								Hexa address
	Decoder input/enable				Address input to ADC				
	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	
SOC channel-0	0	1	1	0	x	0	0	0	60
SOC channel-1	0	1	1	0	x	0	0	1	61
SOC channel-2	0	1	1	0	x	0	1	0	62
SOC channel-3	0	1	1	0	x	0	1	1	63
SOC channel-4	0	1	1	0	x	1	0	0	64
SOC channel-5	0	1	1	0	x	1	0	1	65
SOC channel-6	0	1	1	0	x	1	1	0	66
SOC channel-7	0	1	1	0	x	1	1	1	67
Read EOC	0	1	0	1	x	x	x	x	50
Read ADC output	0	1	1	1	x	x	x	x	70

The chip select signal IOCS-6 is used to give **Start Of Conversion (SOC)** signal to ADC along with a channel address. The chip select IOCS-5 is used to enable the tristate buffer provided for interfacing EOC with data bus. The chip select signal IOCS-7 is inverted and used to enable the output buffer of ADC whenever the digital data has to read from ADC.

The output clock signal of 8085 microprocessor is divided by suitable clock divider circuit and used as clock signal for ADC. A separate voltage source has to be provided to give an accurate reference voltage levels. The **End Of Conversion (EOC)** signal of ADC is connected to the bus line D<sub>0</sub> of the system through a tristate buffer, so that the processor can check for a valid EOC before reading the output buffer of ADC.

The working of ADC 0809 with 8085 will be as follows :

- First the processor selects a channel by sending an address and SOC pulse is asserted **high** and **low**.
- Once address of channel and SOC pulse are applied, the ADC will start converting the signal at the selected channel.
- Then the processor keeps on polling the status of EOC to verify whether it is set to one. (when the conversion is completed by ADC0809 the EOC is set to one.)
- When the processor finds a valid EOC, it will read the digital value from output buffer of ADC.

## 6.8 SUMMARY

- The programmable devices can be set up to perform specific functions by writing control words into the control registers.
- The data transfer schemes refer to the method of data transfer between the processor and peripheral devices.
- The data transfer schemes have been broadly classified into programmed data transfer scheme and DMA data transfer scheme.
- In programmed data transfer, a subroutine requests the device for data transfer to or from one of the processor register.
- The different types of programmed data transfer scheme are Synchronous, Asynchronous and Interrupt driven data transfer scheme.
- The direct data transfer between IO device and memory is called DMA.
- The DMA data transfer can be performed in a microcomputer system using DMA controller.

- The DMA controller works as a slave during the programming mode and works as a master during the DMA mode.
- In DMA data transfer, the processor is forced to hold state by an IO device through DMA controller until the data transfer between the device and the memory is complete.
- The different types of DMA are Cycle stealing, Block mode and Demand transfer mode.
- In cycle stealing DMA, one DMA data transfer is performed in between instruction cycle.
- The INTEL 8212 is an 8-bit IO port device. It is hardwired in the system as either an input or output port.
- The IO function of 8212 is determined by mode (MD) pin. If MD = 1, then 8212 can be used as output port and if MD = 0, then it can be used as input port.
- The 8155 consists of 256 bytes of R/W memory, three programmable IO ports and a programmable timer.
- In 8155, the ports A and B can be programmed to work either as simple or handshake IO port.
- In 8155, the timer has a 14-bit counter which can be programmed to work in 4 modes.
- In 8155, the signals used for handshake data transfer are **Strobe (STB)**, **Buffer Full (BF)** and **Interrupt (INTR)**.
- The INTEL 8255 consists of three programmable ports A, B and C.
- The port-A of 8255 can be programmed to work in three operating modes. They are Simple (Mode-0), Handshake (Mode-1) or Bidirectional (Mode-2) IO port.
- The port-B of 8255 can be programmed to work either as Simple (Mode-0) or Handshake (Mode-1) IO port.
- The port-C pins of 8255 can be individually set/reset to generate control signals.
- The two control words of 8255 are Mode set and Bit set/reset control words.
- In 8255, the handshake signals used for input operation are **Input Buffer Full (IBF)**, **Strobe (STB)** and **Interrupt (INTR)**.
- In 8255, the handshake signals used for output operation are **Output Buffer Full (OBF)**, **Acknowledge (ACK)** and **Interrupt (INTR)**.
- The INTEL 8355 consists of 2 kb ROM and two numbers of 8-bit programmable IO ports.
- In 8355, each IO port line is individually programmable as input or output.
- The INTEL 8755 consists of 2 kb EPROM and two numbers of 8-bit programmable IO ports.
- The individual pins of the ports of 8755 can be programmed as input or output by loading an 8-bit data in DDR of the concerned port.
- The INTEL 8279 is a dedicated controller specially developed for keyboard/display interfacing in 8085/8086 microprocessor-based system.
- The 8279 can generate 256 keycodes with 64 keys arranged as 8×8 matrix, shift and control.
- The 8279 can provide a multiplexed interface for a maximum of 16 numbers of 7-segment LED's.
- The 8279 has a 8 x 8 RAM for storing key codes and 16 x 8 RAM for storing display codes.
- The 8279 requires an internal clock frequency of 100 kHz.
- The various tasks involved in keyboard interfacing are sensing a key actuation, debouncing the key and generating the keycode.
- The process of eliminating multiple keycode generation due to bouncing of key is called debouncing.
- The disadvantage in keyboard interfacing using ports is that most of the processor time is utilized in keyboard scanning and debouncing.
- The current requirement of a 7-segment LED is 140 mA to 200 mA.
- In multiplexed display, only one 7-segment LED is turned ON at a time. Due to persistence of vision the display appears to be continuous to a human eye.
- The disadvantage in using the ports and latches for LED display interfacing is that a considerable processor time is utilized for display refreshing.
- The 8279 provides a hardware solution for keyboard and display interfacing in microprocesso- based system.
- The serial data can be sent synchronously or asynchronously.
- In synchronous serial transmission, the data is sent in block at a constant rate. The start and end of a block are identified with specific bytes or bit patterns.

- In asynchronous transmission, data is transmitted one by one and each data has a bit which identifies its start and 1 or 2 bits which identify its end.
- The term baud rate is used to indicate the rate at which serial data is being transferred. Baud rate is defined as  $\frac{1}{\text{The time for a bit cell}}$ .
- The device which can be programmed to perform synchronous or Asynchronous serial communication is called USART (Universal Synchronous Asynchronous Receiver Transmitter). The INTEL 8251A is an example of USART.
- The RS-232C is a serial bus whose signals are standardized by Electronics Industries Association (EIA), USA and adapted by IEEE.
- The RS-232C proposes a maximum of 25 signals for serial data transfer, but the first 9 signals are sufficient for most of the serial data transmission scheme.
- In RS-232C, the commonly used voltage levels are +12-V (logic **high**) and -12-V (logic **low**).
- The RS-232C signal levels are not compatible with TTL logic levels. Hence for interfacing TTL devices to RS-232C serial bus, level converters are employed.
- The INTEL 8251A is a programmable serial communication interface chip designed for synchronous and asynchronous serial data communication.
- The transmitter and receiver sections of 8251A are double buffered.
- The 8251A provides control signals necessary for MODEM interface.
- The control words of 8251A are Mode word and Command word.
- The resolution of n-bit DAC or ADC is  $1/2^n$  of full scale value (or  $1/2^n$  of reference value)
- The conversion or settling time is the time taken to convert a digital data to analog signal in DAC and vice versa in ADC.

## 6.9 SHORT QUESTIONS AND ANSWERS

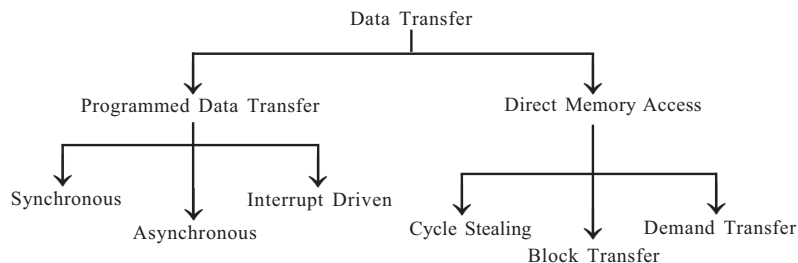
6.1 What is a programmable peripheral device?

If the functions performed by a peripheral device can be altered or changed by a program instruction then the peripheral device is called programmable device. Usually the programmable devices will have control registers. The device can be programmed by sending control word in the prescribed format to the control register.

6.2 What is data transfer scheme and what are its types ?

The data transfer scheme refers to the method of data transfer between the processor and peripheral devices.

The different types of data transfer schemes are shown below :



6.3 What is synchronous data transfer scheme?

In synchronous data transfer scheme, the processor does not check the readiness of the device after a command has been issued for read/write operation. In this scheme the processor will request the device to get ready and then read/write to the device immediately after the request. In some synchronous schemes a small delay is allowed after the request.

6.4 *What is asynchronous data transfer scheme ?*

In asynchronous data transfer scheme, first the processor sends a request to the device for read/write operation. Then the processor keeps on polling the status of the device. Once the device is ready, the processor executes a data transfer instruction to complete the process.

6.5 *How is DMA initiated?*

When the IO device needs a DMA transfer, it will send a DMA request signal to the DMA controller. The DMA controller in turn sends a HOLD request to the processor. When the processor receives a HOLD request, it will drive its tristate pins to **high impedance** state at the end of current instruction execution and sends an acknowledge signal to the DMA controller. Now the DMA controller will perform DMA transfer.

6.6 *What are the different types of DMA?*

The different types of DMA data transfer are cycle stealing (or single transfer) DMA, Block transfer (or Burst mode) DMA and Demand transfer DMA.

6.7 *What is cycle stealing DMA?*

In cycle stealing DMA (or single transfer mode) the DMA controller will perform one DMA transfer in between the instruction cycles (i.e., In this mode the execution of one processor instruction and one DMA data transfer will take place alternatively).

6.8 *What are block and demand transfer modes DMA?*

In block transfer mode, the DMA controller will transfer a block of data and relieve the bus to the processor. After sometime another block of data is transferred by the DMA and so on.

In demand transfer mode the DMA controller will complete the entire data transfer at a stretch and then relieve the bus to the processor.

6.9 *What are the operating modes of 8212 ?*

8212 can be hardwired to work either as a latch or tristate buffer. If mode (MD) pin is tied **high** then it will work as a latch and it can be used as the output port. If mode (MD) pin is tied **low** then it works as a tristate buffer and it can be used as the input port.

6.10 *What are the various internal devices of INTEL 8155?*

The INTEL 8155 is an IC consisting of static RAM, IO ports and timer. The internal devices of 8155 are 256 bytes of static RAM, three numbers of programmable IO ports and a 14-bit programmable timer.

6.11 *What is handshake port ?*

In handshake port, signals are exchanged between IO device and port or between port and processor for checking/informing various conditions of the device.

6.12 *Explain the working of a handshake input port.*

In handshake input operation, the input device will check whether the port is empty or not. If the port is empty then it will load data to port. When the port receives the data, it will inform the processor for read operation. Once the data has been read by the processor, the port will signal the input device that it is empty. Now, the input device can load another data to the port and the above process will be repeated.

6.13 *Explain the working of a handshake output port.*

In handshake output operation, the processor will load a data to port. When the port receives the data, it will inform the output device to collect the data. Once the output device accepts the data, the port will inform the processor that it is empty. Now the processor can load another data to the port and the above process will be repeated.

6.14 *What are the internal devices of 8255 ?*

The internal devices of 8255 are Port-A, Port-B and Port-C. The ports can be programmed for either input or output function in different operating modes.

6.15 What are the operating modes of port-A of 8255?

The port-A of 8255 can be programmed to work in any one of the following operating modes as input or output port.

- Mode-0 : Simple IO port.
- Mode-1 : Handshake IO port.
- Mode-2 : Bidirectional IO port.

6.16 What are the functions performed by port-C of 8255?

1. Port-C pins are used for handshake signals.
2. Port-C can be used as an 8-bit parallel IO port in Mode-0.
3. It can be used as two numbers of 4-bit parallel ports in Mode-0.
4. The individual pins of port-C can be set or reset for various control applications.

6.17 What is the bit format used for sending asynchronous serial data?

In asynchronous transmission, each data character has a bit which identifies its start and 1 or 2 bits which identify its end. A typical bit format is shown in Fig. Q6.17.

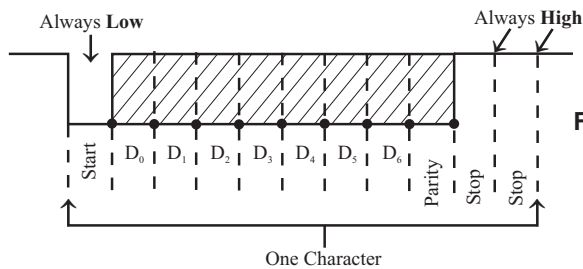


Fig. Q6.17 : Bit format used for sending asynchronous serial data.

6.18 What is baud rate ?

The baud rate is the rate at which the serial data is transmitted. Baud rate is defined as  $\frac{1}{\text{(The time for a bit cell)}}$ . In some systems one bit cell has one data bit, then the baud rate and bits per second are same.

6.19 What is RS-232C standard ?

RS-232C is a serial bus consisting of a maximum of 25 signals. These bus signals are standardized by EIA (Electronics Industries Association), USA and adopted by IEEE. Usually the first 9 signals are sufficient for most of the serial data transmission. The RS-232C serial bus is usually terminated using either a 9-pin connector or a 25-pin connector.

6.20 What voltage levels are used in RS-232C serial communication standard?

The voltage levels for all RS-232C signals are as follows.

- Logic low = -3-V to -15-V under load (-25-V on no load)
- Logic high = +3-V to +15-V under load (+25-V on no load)

Commonly used voltage levels are +12-V (logic **high**) and -12-V (logic **low**).

6.21 How is the RS-232C serial bus interfaced to TTL logic device ?

The RS-232C signal voltage levels are not compatible with TTL logic levels. Hence, for interfacing TTL devices to RS-232C serial bus, level converters are used. The popularly used level converters are MC 1488 and MC 1489 or MAX 232.

6.22 What is USART ?

The device which can be programmed to perform synchronous or asynchronous serial communication is called USART (Universal Synchronous Asynchronous Receiver Transmitter). INTEL 8251A is an example of USART.



6.23 *What are the functions performed by INTEL 8251A?*

The INTEL 8251A is used for converting parallel data to serial or vice versa. The data transmission or reception can be either asynchronous or synchronous. The 8251A can be used to interface MODEM and establish serial communication through MODEM over telephone lines.

6.24 *What are the control words of 8251A and what are its functions?*

The control words of 8251A are Mode word and Command word. The mode word informs 8251 about the baud rate, character length, parity and stop bits. The command word can be sent to enable the data transmission and/or reception.

6.25 *What is the information that can be obtained from the status word of 8251?*

The status word can be read by the CPU to check the readiness of the transmitter or receiver and to check the character synchronization in synchronous reception. It also provides information regarding various errors in the data received. The various error conditions that can be checked from the status word are parity error, overrun error and framing error.

6.26 *List the functions performed by 8279.*

The functions performed by 8279 are :

- Keyboard scanning
- Key debouncing
- Keycode generation
- Informing the key entry to CPU
- Storing display codes
- Output display codes to LEDs
- Display refreshing

6.27 *What is the maximum number of keycodes that can be generated by 8279?*

In scanned keyboard mode the maximum size of keyboard matrix array that can be interfaced to 8279 is  $8 \times 8$ , which consists of 64 keys. In addition, the 8279 has two control keys called shift and control. For each key press, an 8-bit code is generated and stored in FIFO (keyboard RAM of 8279). The keycode consists of row and column number of the key in binary along with the status of shift and control key. Hence with 64 contact keys, shift and control key, a maximum of 256 keycodes can be generated by 8279.

6.28 *What are the programmable display features of 8279 ?*

The 8279 can be used for interfacing LEDs or 7 segment LEDs. In decoded scan, 4 numbers of 7-segment LEDs can be interfaced and in encoded scan, a maximum of 16 numbers of 7-segment LEDs can be interfaced. The 8279 can be programmed for left entry or right entry.

6.29 *What are the different scan modes of 8279?*

The different scan modes of 8279 are decoded scan and encoded scan. In decoded scan mode, the output of scan lines will be similar to a 2-to-4 decoder. In encoded scan mode, the output of scan lines will be binary count. So, an external decoder should be used to convert the binary count to decoded output.

6.30 *What are the tasks involved in keyboard interface ?*

The tasks involved in keyboard interfacing are sensing a key actuation, debouncing the key and generating keycodes (Decoding the key). These tasks are performed by software if the keyboard is interfaced through ports and they are performed by hardware if the keyboard is interfaced through 8279.

6.31 *What is debouncing ?*

When a key is pressed it bounces for a short time. If a key code is generated immediately after sensing a key actuation, then the processor will generate the same keycode a number of times. (A key typically bounces for 10 to 20 millisecond.) Hence, the processor has to wait for the key bounces to settle before reading the keycode. This process is called keyboard debouncing.

6.32 *What is the disadvantage in keyboard interfacing using ports?*

The disadvantage in keyboard interfacing using ports is that most of the processor time is utilized in keyboard scanning and debouncing. As a result the computational speed/efficiency of the processor will be reduced.

6.33 *What is multiplexed display? What is its advantage?*

The process of switching ON the display devices one by one for a specified time interval is called multiplexed display. In microprocessor-based systems, six to eight 7-segment LEDs are interfaced to provide multiplexed display. At any one time only one 7-segment LED is made to glow at a time. After a few milliseconds, the next 7-segment LED is made to glow and so on. Due to persistence of vision, it will appear as if the LEDs are glowing continuously. The advantage in multiplexed display is that the power requirement of the display devices are reduced to a very large extent.

6.34 *What is the disadvantage in 7-segment LED interfacing using ports?*

The disadvantage in using ports for 7-segment LED interfacing is that most of the processor time is utilized for display refreshing.

6.35 *What is the advantage in using INTEL 8279 for keyboard and display interfacing?*

When 8279 is used for keyboard and display interfacing, it takes care of all the tasks involved in keyboard scanning and display refreshing. Hence the processor is relieved from the task of keyboard scanning, debouncing, keycode generation and display refreshing. So, the processor time can be more efficiently used for computing.

6.36 *What is the difference in programming the 8279 for encoded scan and decoded scan?*

If the 8279 is programmed for decoded scan then the output of scan lines will be decoded output and if it is programmed for encoded scan then the output of scan lines will be binary count. In encoded mode, an external decoder should be used to decode the scan lines.

6.37 *How is a keyboard matrix is formed in a keyboard interface using 8279?*

The return lines,  $RL_0$  to  $RL_7$  of 8279 are used to form the columns of keyboard matrix. In decoded scan the scan lines  $SL_0$  to  $SL_3$  of 8279 are used to form the rows of keyboard matrix. In encoded scan mode, the scan line  $SL_0$  to  $SL_3$  are connected to input of a decoder and the output lines of decoder are used as rows of keyboard matrix.

6.38 *What is scanning in keyboard and what is scan time?*

The process of sending a zero to each row of a keyboard matrix and reading the columns for key actuation is called scanning. The scan time is the time taken by the device/processor to scan all the rows one by one starting from first row and coming back to the first row again.

6.39 *What is scanning in display and what is the scan time?*

In display devices, the process of sending display codes to 7-segment LEDs to display the LEDs one by one is called scanning (or multiplexed display). The scan time is the time taken to display all the 7-segment LEDs one by one, starting from first LED and coming back to the first LED again.

6.40 *What is resolution in DAC?*

The resolution in DAC is the smallest possible analog value that can be generated by the n-bit binary input. If the reference voltage in n-bit DAC is  $V_{REF}$ , then the resolution is  $(1/2^n) \times V_{REF}$  volts.

6.41 *What are the internal devices of a typical DAC?*

The internal devices of a DAC are R/2R resistive network, an internal latch and current to voltage converting amplifier.

6.42 *What is settling or conversion time in DAC?*

The time taken by the DAC to convert a given digital data to corresponding analog signal is called conversion time.

6.43 *What are the different types of ADC?*

The different types of ADC are successive-approximation ADC, counter type ADC, flash type ADC, integrator converters and voltage-to-frequency converters.

6.44 *What is resolution and conversion time in ADC?*

The resolution in ADC is the minimum analog value that can be represented by the digital data. If the ADC gives n-bit digital output and the analog reference voltage is  $V_{REF}$ , then the resolution is  $(1/2^n) \times V_{REF}$  Volts. The conversion time in ADC is defined as the total time required to convert an analog signal into its digital equivalent.

# MICROCOMPUTER SYSTEM DESIGN AND APPLICATIONS

## 7.1 DESIGNING A MICROPROCESSOR-BASED SYSTEM

Design of microcomputer system starts with specifications. The specification of the system includes the following :

1. Input device
2. Output device
3. Memory requirement
4. System clock frequency
5. Peripheral devices required
6. Type of CPU (Microprocessor)
7. Applications or Nature of work.

### Input Devices

The popular input device in single board microcomputer system (microprocessor trainer kit) is the Hex-keyboard. Other forms of input devices are DIP switches, ADC interfaced through port and floppy disk interfaced through floppy disk controller - INTEL 8272. The Hex-keyboard is normally interfaced to 8085/8086 system using INTEL 8279 keyboard and display controller. A maximum of 64 keys can be interfaced using 8279. Along with shift and control, 256 key-codes can be generated using 8279.

### Output Devices

The popular output device used in single board microcomputer (microprocessor trainer kit) is the 7-segment LED. The seven segment LEDs are interfaced to 8086 processor using INTEL 8279 keyboard and display controller. The 8279 is a dedicated controller which takes care of key-board scanning and display refreshing. A maximum of 16 number of 7-segment LEDs can be interfaced using one 8279 in 8085/8086-based system as multiplexed display.

Other output devices are LCD (Liquid Crystal Display), printer, floppy disk and CRT terminal. The LCD and printer can be interfaced using ports. Special dedicated controllers are required for interfacing floppy disk and CRT terminal. The INTEL 8272 or INTEL 82072 floppy disk controller and INTEL 8275 CRT controller are popularly used with 8085/8086/8088 systems.

### **Memory Requirement**

The memory requirement of the system is splitted between EPROM and RAM. The memory capacity of EPROM and RAM are estimated based on the applications and work to be performed by the processor. Most of the microprocessors use memory with word size of 1-byte. Hence the memory capacity of the system is specified in kilo bytes.

The popular EPROM used in the 8085-based system are 2708 ( $1k \times 8$ ), 2716 ( $2k \times 8$ ), 2732 ( $4k \times 8$ ), 2764 ( $8k \times 8$ ) and 27256 ( $32k \times 8$ ). The popular static RAM used in the 8085-based system are 6208 ( $1k \times 8$ ), 6216 ( $2k \times 8$ ), 6232 ( $4k \times 8$ ), 6264 ( $8k \times 8$ ), and 62256 ( $32k \times 8$ ). The memories are chosen with compatible access time, i.e., the access time of memories should be less than the read time and write time of the processor.

The total memory requirement of the system is implemented by using more than one memory IC. But the processor, at any one time can communicate with (or access) only one memory IC. To select a memory IC, chip select signals has to be generated using decoders. The input to the decoders are the unused address lines. Also, to each memory location, specific addresses should be allotted. [These techniques are discussed in memory interfacing.]

The EPROMs are mapped at the beginning of memory space in order to store the monitor program in EPROM and to execute the monitor program upon power-on-reset. [Every system will be resetted when power supply is switched ON.]

In 8085-based system, the interrupt vector addresses belong to EPROM locations. Normally, a jump instruction with an address of RAM location is stored in the vector locations. Hence, the user can store the interrupt subroutines in these jump addresses.

Apart from allocating addresses to memory devices, the peripherals and IO devices should also be allotted specific addresses. The peripherals and IO devices can be either memory-mapped or IO-mapped in the system. If the memory requirement of the system is very large and in future if memory expansion is required, then the peripherals and IO devices are IO-mapped in the system. If memory requirement of the system is less, then the peripherals and IO devices are memory mapped in the system.

### **System Clock Frequency**

The microprocessor and the peripheral devices require a clock signal for synchronizing various internal operations or devices. An oscillator is needed for generating the clock signal. The oscillator consists of an amplifier and a feedback network. The feedback network has R, L, C or Quartz crystal.

In 8085 processor, the oscillator circuit (except the Quartz crystal and L-C component) is fabricated in the processor itself. Hence, it is necessary to connect a quartz crystal external to the processor. The oscillator generates a clock whose frequency is double to that of internal clock. The processor divides the generated clock by **two** for internal operations.

For each system a maximum clock speed is specified. Driving a system at the maximum clock is advantageous, because the execution time will be minimum if the clock is maximum. When the system is driven at maximum clock, then the peripherals chosen should have speed compatibility with the processor.

### **Peripheral Devices**

The peripheral devices required for a system depends on its applications. Some of the peripheral devices that can be interfaced to 8086-based system are:

- Programmable Interval Timer – INTEL 8253/8254
- USART – INTEL 8251
- Programmable Peripheral Interface – INTEL 8255
- Keyboard /Display Controller – INTEL 8279
- Programmable Interrupt Controller – INTEL 8259
- DMA Controller – INTEL 8237/8257
- ADC
- DAC, etc.

When the system has to monitor an analog signal from a sensor, then an ADC can be interfaced using 8255 ports. If the processor has to control an analog device, then it has to convert the digital signal to analog signal using DAC.

When the system requires large number of interrupt inputs, the interrupt structure of the system has to be expanded by using interrupt controller 8259. One 8259 supports 8-interrupt request.

The USART-8251 can be used for serial data communication and the programmable timer-8253/8254 can be employed for various timing operations.

### **Type of CPU**

The CPU of the system is a microprocessor. The microprocessor is chosen based on clock speed, instruction execution time, memory capacity, size of data and address, addressing modes, the operations it can perform and the number of additional devices required to form a system.

### **Application or Nature of Work**

The specifications of the microprocessor itself depends on the applications for the proposed system and the nature of work it is going to perform. The input device, output device, memory requirement, peripheral requirement, and the choice of CPU, all depends on the nature of work to be performed by the system.

## **7.2 8085-BASED MINIMUM SYSTEM**

A minimum system is one which is formed using minimum number of IC chips. The minimum system in 8085 is formed using 8155, 8355, and 8755. In this, the 8085 is the CPU and the 8155, 8355, 8755 are memory and port devices. The 8155, 8355 and 8755 are called **Programmable Peripheral Interface (PPI)**. The 8155 has ports and  $256 \times 8$  static RAM, the 8355 has ports and

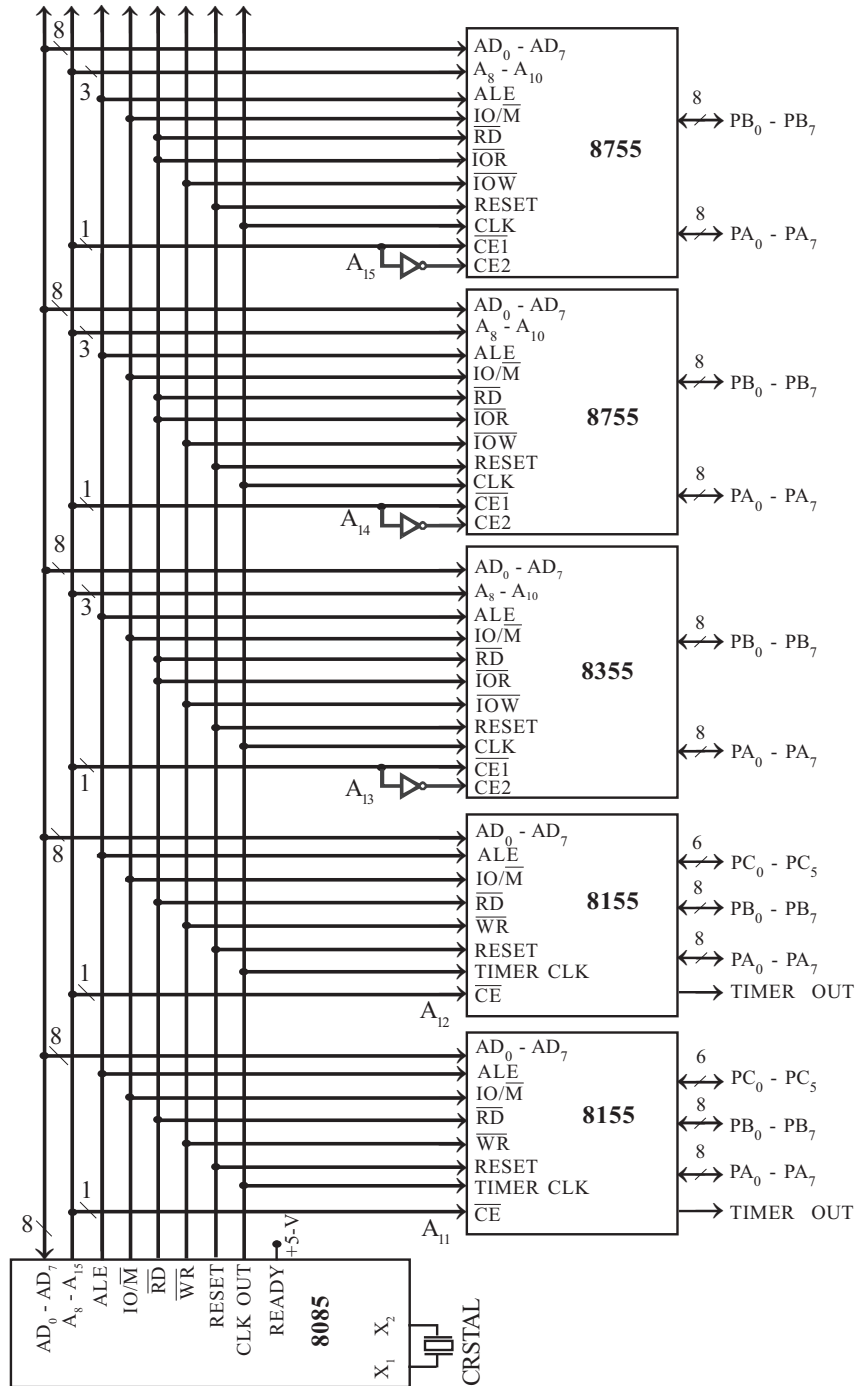


Fig. 7.1 : 8085-based minimum system.

2 k × 8 ROM and the 8755 has ports and 2 k × 8 EPROM. The port and memory requirement of the system are provided by PPI. The input and output devices like key-board and display LEDs are interfaced via ports of PPI to the system.

In minimum system, the monitor program and system program are usually stored in permanent memories like ROM and EPROM of 8355 and 8755. For stack operations we require RAM memory. The RAM memory requirement is provided by 8155 .

The PPIs (8155, 8355 and 8755) have internal address latches, hence there is no need to demultiplex the  $AD_0$ - $AD_7$  lines of the processor. The demultiplexing is done by these devices, internally. These devices are mapped in the system by linear address selection method. In this method, the unused address lines are directly connected to chip select pins of the peripheral ICs. The memories of the PPI are interfaced by memory mapping and the ports of the PPIs are mapped by IO mapping. The PPIs accept  $IO/\overline{M}$  signals and differentiate between memory and port addresses. A 16-bit address is allotted to memory locations and 8-bit IO address is allotted to ports. An example of minimum system is shown in Fig. 7.1.

### 7.3 TEMPERATURE CONTROL SYSTEM

The microprocessor-based temperature control system can be used for automatic control of the temperature of a plant. A simplified block diagram of 8085 microprocessor-based temperature control system is shown in Fig. 7.2.

The system consist of 8085 microprocessor as CPU, EPROM memory for program storage, RAM memory for stack and data storage, INTEL 8279 for keyboard and display interface, ADC, DAC, INTEL 8255 for IO ports, amplifiers, signal conditioning circuit, temperature sensor and supply control circuit. In this system, the temperature is controlled by controlling the power input to the heating element.

The EPROM memory is provided for storing system program, and RAM memory for temporary data storage and stack operation. Using INTEL 8279, a keyboard and six numbers of 7-segment LEDs are interfaced to the system. The system has been designed to accept the desired temperature and various control commands through keyboard. The 7-segment display has been provided to display the temperature of the plant at any time instant.

The temperature of the plant is measured using a temperature sensor. The different types of temperature sensors that can be used for temperature measurement are thermo-couples, Thermistors, PN-junctions, IC sensors like AD590, etc. These sensors will convert the input temperature to proportional analog voltage or current. The output signal of the sensor will be a weak signal and so it has to be amplified using high input impedance operational amplifier. Then the analog signal is scaled to suitable level by the signal conditioning circuit.

The microprocessor can process only digital signals and so the analog signal from signal conditioning circuit cannot be read by the processor directly. The system has an Analog-to-Digital Converter (ADC) to convert the analog signal to proportional digital data. In this system, the ADC is interfaced to 8085 processor through port-A and port-C of 8255. The 8085 processor send signal to ADC through port-C to start conversion and at the end of conversion it reads the digital data from the port-A of 8255.



*The McGraw-Hill Companies*



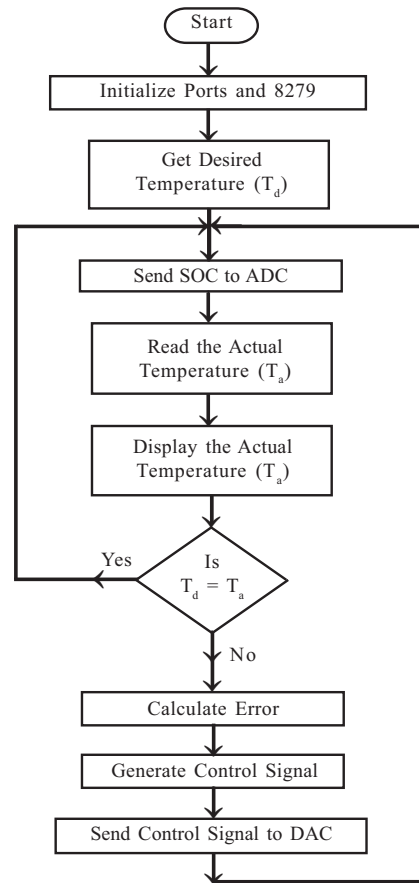
The 8085 processor calculates the actual temperature using the input data and displays it on the 7-segment LED. Also, the processor compares the desired temperature with actual temperature (The operator can enter the desired temperature through keyboard) and calculate the error (the difference between the actual temperature and the desired temperature).

The error is used to compute a digital control signal, which is converted to analog control signal by DAC. The DAC is interfaced to the system through port-B of 8255. The analog control signal produced by DAC is used to control the power supply of the heating element of the plant.

The digital control signal can be computed by the 8085 processor using different digital control algorithms (P/PI/PID/FUZZY logic control algorithms).

The control circuit for power supply can be either thyristor-based circuit or relay. In case of thyristor control circuits, the firing angle can be varied by the control signal to control the power input to the heater. In case of relay, the control signal can switch ON/OFF the relay to control the power input to the heater.

The sequence of operations performed by the microprocessor-based system are shown in the flowchart of Fig. 7.3.



**Fig. 7.3 :** Flowchart for temperature control system.

#### 7.4 MOTOR SPEED CONTROL SYSTEM

The microprocessor-based speed control system can be used to automatically control the speed of a motor. A typical 8085 microprocessor-based dc motor speed control system is shown in Fig. 7.4. In this system, the speed of the dc motor is varied by varying the armature voltage and the field voltage is kept constant. A controlled rectifier using SCR develops the required armature voltage and the uncontrolled rectifier generates the required field voltage. The microprocessor controls the speed of the motor by varying the firing angle of SCRs in the controlled rectifier.

The speed control system has been developed using 8085 microprocessor as CPU. The system has EPROM for system program storage, and RAM for temporary data storage and stack.

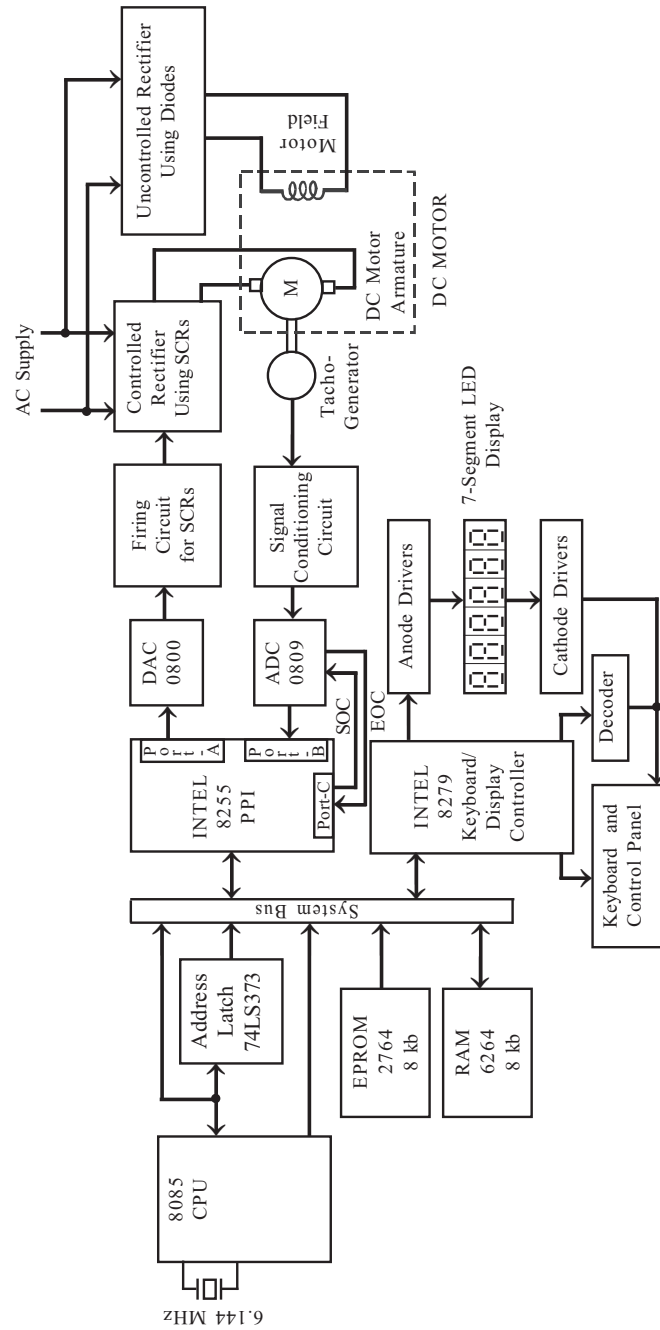


Fig. 7.4 : An 8085 microprocessor-based dc motor speed control system.

A keyboard has been provided to input the desired speed and other commands to operate the system. In order to display the speed of the motor, 7-segment LED display has been provided. The keyboard and 7-segment LED display have been interfaced to 8085-based system using Keyboard/Display controller INTEL 8279.

The speed of the dc motor is measured using a tachogenerator. It produces an analog voltage proportional to the speed of the motor. Then the analog signal is scaled to desired level by the signal conditioning circuit and digitized using ADC. (The processor cannot process the analog signal directly, hence the analog signal is digitized using ADC.)

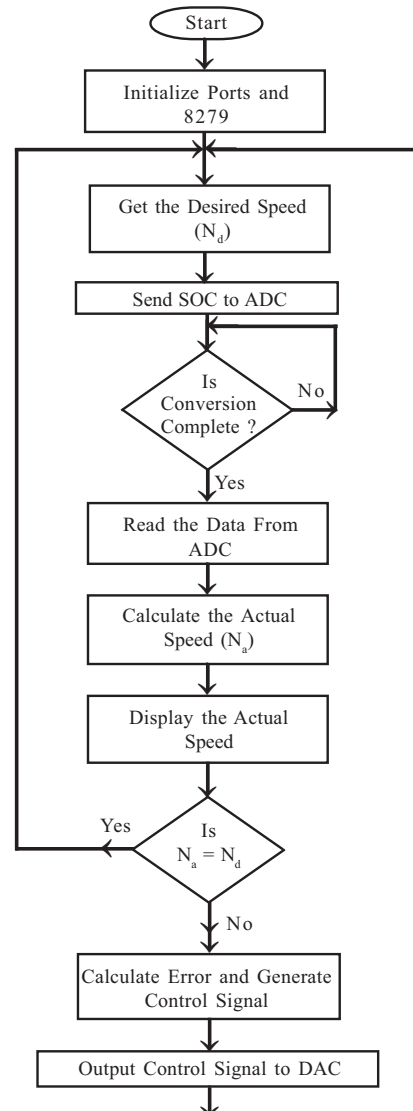
The ADC is interfaced to 8085 processor through the port-B and port-C of 8255. The processor can send a start of conversion to ADC through port-C pin and at the end of conversion it can read the digital data from port-B of 8255. This digital data is proportional to actual speed.

The processor calculates the actual speed and displays it on LEDs. Also, the processor compares the actual speed with the desired speed entered by the operator through the keyboard. If there is a difference then an error is estimated. The error can be modified by a digital control algorithm, (P/PI/PID/FUZZY logic control algorithm) to produce a digital control signal.

The digital control signal is converted to analog signal by the DAC. The analog control signal is used to alter the firing angle of SCRs in the controlled rectifiers. The operational flow of the speed control system is shown in the flowchart of Fig. 7.5.

## 7.5 TRAFFIC LIGHT CONTROL SYSTEM

The traffic lights placed at the road crossings can be automatically switched ON/OFF in the desired sequence using the microprocessor system. The system can also have a manual control option, so that during heavy traffic (or during traffic jam) the duration of ON/OFF time can be varied by the operator.



**Fig. 7.5 :** Flowchart for a dc motor speed control system.

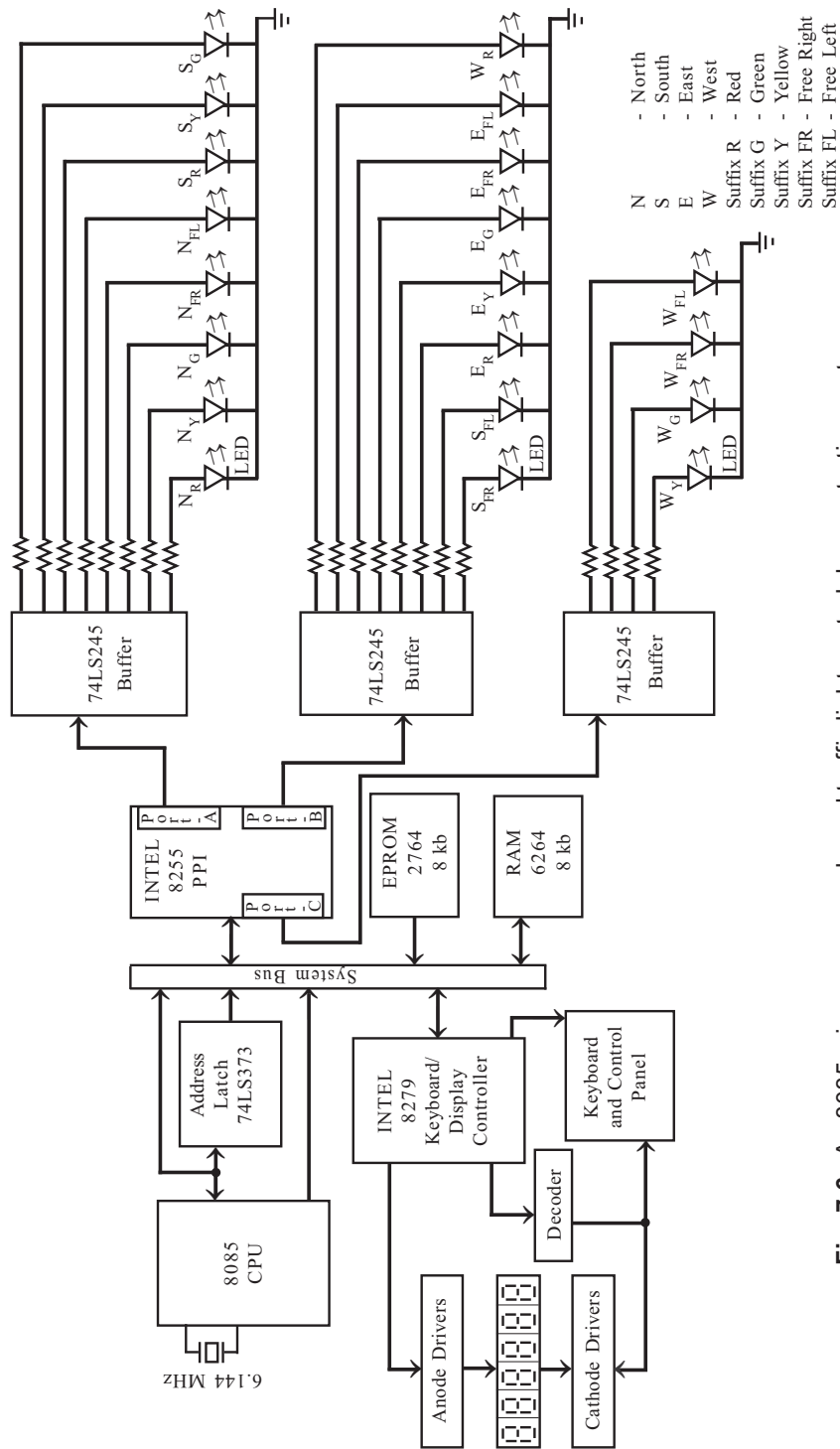


Fig. 7.6 : An 8085 microprocessor-based traffic light control demonstration system.

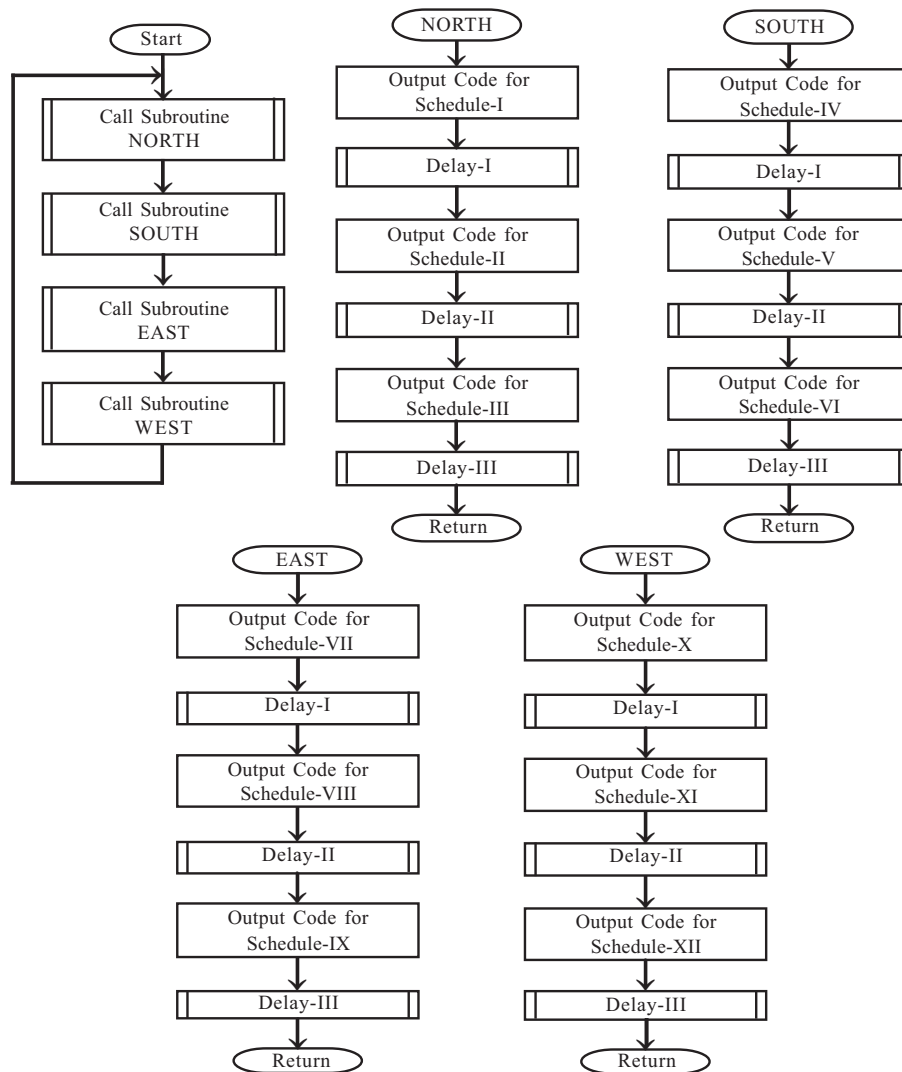
TABLE -7.1 : SWITCHING SCHEDULE FOR TRAFFIC LIGHTS

ON/OFF status of traffic lights																					
Switching schedule	PC <sub>3</sub> W <sub>FL</sub>	PC <sub>2</sub> W <sub>FR</sub>	PC <sub>1</sub> W <sub>G</sub>	PC <sub>0</sub> W <sub>Y</sub>	PB <sub>7</sub> W <sub>R</sub>	PB <sub>6</sub> E <sub>FL</sub>	PB <sub>5</sub> E <sub>FR</sub>	PB <sub>4</sub> E <sub>G</sub>	PB <sub>3</sub> E <sub>Y</sub>	PB <sub>2</sub> E <sub>R</sub>	PB <sub>1</sub> S <sub>FL</sub>	PB <sub>0</sub> S <sub>FR</sub>	PA <sub>7</sub> S <sub>G</sub>	PA <sub>6</sub> S <sub>Y</sub>	PA <sub>5</sub> S <sub>R</sub>	PA <sub>4</sub> N <sub>FL</sub>	PA <sub>3</sub> N <sub>FR</sub>	PA <sub>2</sub> N <sub>G</sub>	PA <sub>1</sub> N <sub>Y</sub>	PA <sub>0</sub> N <sub>R</sub>	
Schedule I	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	
Schedule II	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	
Schedule III	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	1	1	1	0	0	
Schedule IV	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	
Schedule V	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1	
Schedule VI	0	0	0	0	1	0	0	0	0	1	1	1	1	0	0	0	0	0	0	1	
Schedule VII	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1	
Schedule VIII	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	
Schedule IX	0	0	0	0	1	1	1	1	0	0	0	0	0	0	1	0	0	0	0	1	
Schedule X	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	
Schedule XI	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	
Schedule XII	1	1	1	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	

Note : "1" represents ON and "0" represents OFF.

A typical traffic light control system (demonstration type) is shown in Fig. 7.6. The system has been developed using 8085 as CPU. The system has EPROM memory for system program storage and RAM memory for stack operation. For manual control, a keyboard has been provided. It will be helpful for the operator if the direction of traffic flow is displayed during manual control. Hence 7-segment LEDs are interfaced to display the direction of traffic flow both during manual and automatic mode.

The primary function of the microprocessor in the system is to switch ON/OFF the Red/Yellow/Green lights in the specified sequence. In the demonstration system of Fig. 7.6, Red/Yellow/Green LEDs are provided instead of lights (lamps). The LEDs are interfaced to the system through buffer (74LS245) and ports of 8255.



**Fig. 7.7 :** Flowchart for traffic light control program.

In the practical implementation scheme, the lights can be turned ON/OFF using driver transistors and relays. In practical implementation, the output of buffer (74LS245) can be connected to the driver transistor. A relay placed at the collector of the transistor can be used to switch ON/OFF the light as shown in Fig. 7.8. A reverse biased diode is connected across relay coil to prevent relay chattering (for free-wheeling action).

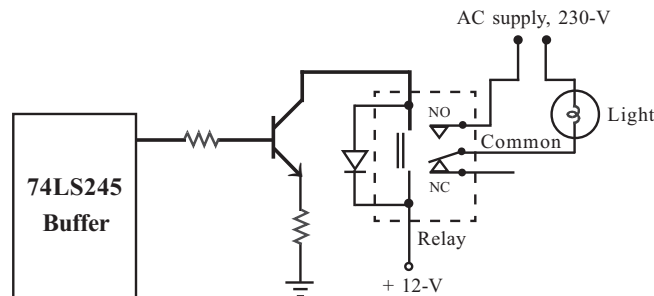


Fig. 7.8 : Switching circuit for traffic light.

The microprocessor sends **high** through a port line to switch ON the light and **low** to switch OFF the light. A switching schedule (or sequence) can be developed as shown in Table-7.1. In this switching sequence, it is assumed that the traffic is allowed only in one direction at a time. In Table-7.1, "1" represents ON condition and "0" represents OFF condition. These 1s and 0s can be directly output to 8255 ports to switch ON/OFF the light. A flowchart for traffic light control program is shown in Fig. 7.7.

The processor can output the codes for switching the lights for schedule-I and then waits. After a specified time delay, the processor output the codes for schedule-II and so on. For each schedule the processor can wait for a specified time. After schedule-XII, the processor can again return to schedule-I. On observing the schedules, we can conclude that three different delay routines are sufficient for implementing the twelve switching schedules.

## 7.6 STEPPER MOTOR CONTROL SYSTEM

The stepper motors are popularly used in computer peripherals, plotters, robots and machine tools for precise incremental rotation. In stepper motor, the stator windings are excited by electrical pulses and for each pulse the motor shaft advances by one angular step. (Since the stepper motor can be driven by digital pulses, it is also called digital motor.) The step size in the motor is determined by the number of poles in the rotor and the number of pairs of stator windings (one pair of stator winding is called one phase). The stator windings are also called control windings.

The motor is controlled by switching ON/OFF the control winding. The popular stepper motor used for demonstration in laboratories has a step size of  $1.8^\circ$  (i.e., 200 steps per revolution). This motor consist of four stator winding and require four switching sequence as shown in

Table-7.2. The basic step size of the motor is called full-step. By altering the switching sequence, the motor can be made to run with incremental motion of half the full-step value. The switching sequence for half step rotation is shown in Table-7.3.

**TABLE-7.2 : SWITCHING SEQUENCE FOR FULL-STEP ROTATION**

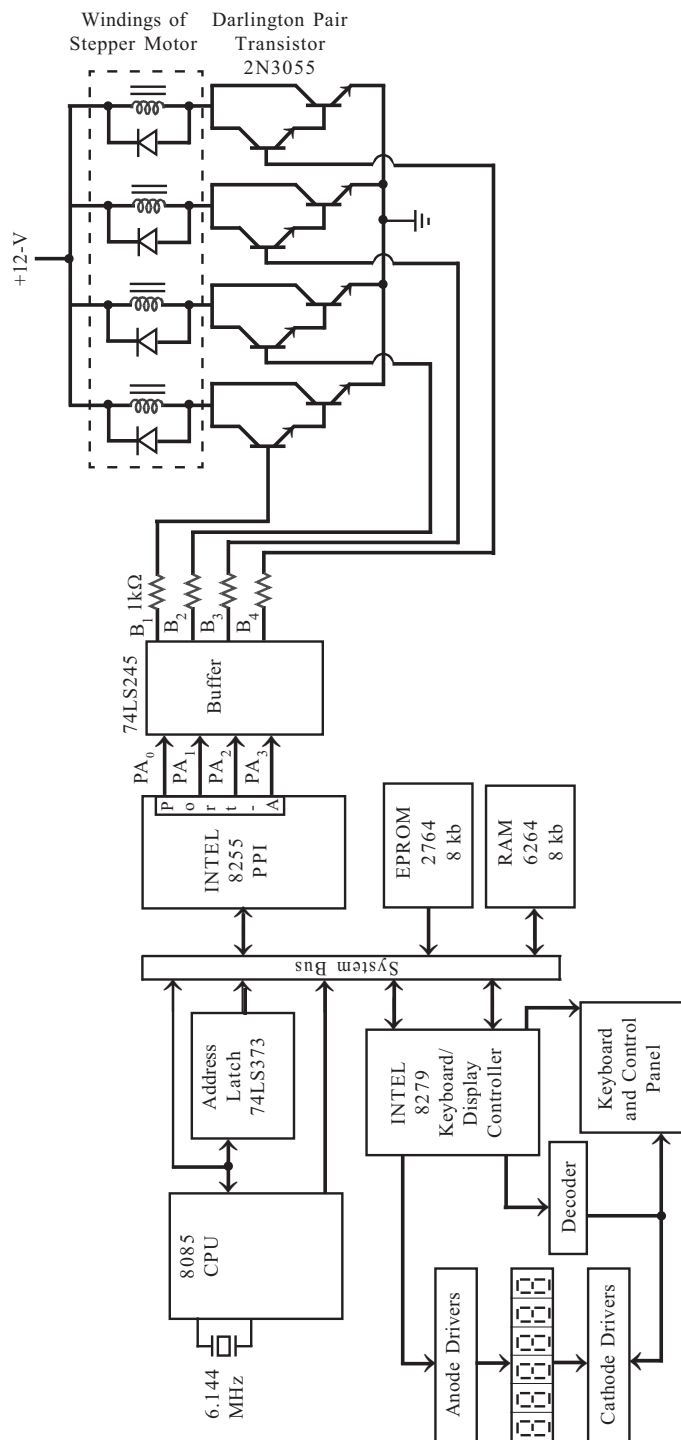
Switching sequence	Clockwise rotation				Anticlockwise rotation			
	PA <sub>3</sub>	PA <sub>2</sub>	PA <sub>1</sub>	PA <sub>0</sub>	PA <sub>3</sub>	PA <sub>2</sub>	PA <sub>1</sub>	PA <sub>0</sub>
Sequence-1	1	1	0	0	0	0	1	1
Sequence-2	0	1	1	0	0	1	1	0
Sequence-3	0	0	1	1	1	1	0	0
Sequence-4	1	0	0	1	1	0	0	1

**TABLE-7.3 : SWITCHING SEQUENCE FOR HALF-STEP ROTATION**

Clockwise rotation				Anticlockwise rotation			
PA <sub>3</sub>	PA <sub>2</sub>	PA <sub>1</sub>	PA <sub>0</sub>	PA <sub>3</sub>	PA <sub>2</sub>	PA <sub>1</sub>	PA <sub>0</sub>
1	1	0	0	0	0	1	1
0	1	0	0	0	0	1	0
0	1	1	0	0	1	1	0
0	0	1	0	0	1	0	0
0	0	1	1	1	1	0	0
0	0	0	1	1	0	0	0
1	0	0	1	1	0	0	1
1	0	0	0	0	0	0	1

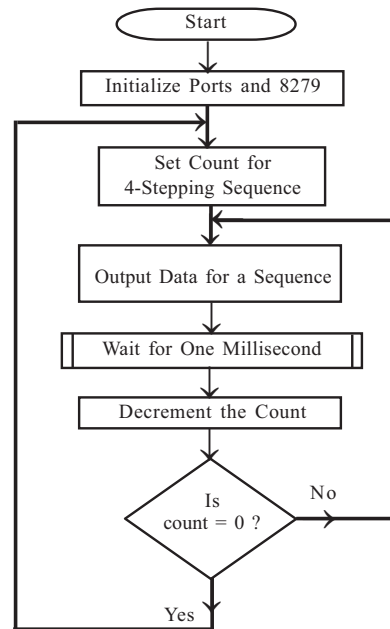
A typical stepper motor control system is shown in Fig. 7.9 a two-phase or four winding stepper motor is shown in Fig. 7.9. The system consists of 8085 microprocessor as CPU, EPROM and RAM memory for program and data storage and for stack. Using INTEL 8279, a keyboard and six number of 7-segment LED display have been interfaced in the system. Through the keyboard the operator can issue commands to control the system. The LED display has been provided to display messages to the operator.





**Fig. 7.9 :** An 8085 microprocessor-based stepper motor control system.

The windings of stepper motor are connected to the collector of darlington pair transistors. The transistors are switched ON/OFF by the microprocessor through the ports of 8255 and buffer (74LS245). A free-wheeling diode is connected across each winding for fast switching. The flowchart for the operational flow of the stepper motor control system is shown in Fig. 7.10. The processor has to output a switching sequence and wait for 1 to 5 milliseconds before sending next switching sequence. (The delay is necessary to allow the motor transients to die-out.)



**Fig. 7.10 :** Flowchart for stepper motor control program.

# APPENDIX I :8085A Instructions in Hexadecimal Order

OPCODE IN HEX	MNEMONIC	OPCODE IN HEX	MNEMONIC	OPCODE IN HEX	MNEMONIC
00	NOP	2B	DCX H	56	MOV D, M
01	LXI B, d16	2C	INR L	57	MOV D, A
02	STAX B	2D	DCR L	58	MOV E, B
03	INX B	2E	MVI L, d8	59	MOV E, C
04	INR B	2F	CMA	5A	MOV E, D
05	DCR B	30	SIM	5B	MOV E, E
06	MVI B, d8	31	LXI SP, d16	5C	MOV E, H
07	RLC	32	STA addr16	5D	MOV E, L
08	---	33	INX SP	5E	MOV E, M
09	DAD B	34	INR M	5F	MOV E, A
0A	LDAX B	35	DCR M	60	MOV H, B
0B	DCX B	36	MVI M, d8	61	MOV H, C
0C	INR C	37	STC	62	MOV H, D
0D	DCR C	38	---	63	MOV H, E
0E	MVI C, d8	39	DAD SP	64	MOV H, H
0F	RRC	3A	LDA addr16	65	MOV H, L
10	---	3B	DCX SP	66	MOV H, M
11	LXI D, d16	3C	INR A	67	MOV H, A
12	STAX D	3D	DCR A	68	MOV L, B
13	INX D	3E	MVI A, d8	69	MOV L, C
14	INR D	3F	CMC	6A	MOV L, D
15	DCR D	40	MOV B, B	6B	MOV L, E
16	MVI D, d8	41	MOV B, C	6C	MOV L, H
17	RAL	42	MOV B, D	6D	MOV L, L
18	---	43	MOV B, E	6E	MOV L, M
19	DAD D	44	MOV B, H	6F	MOV L, A
1A	LDAX D	45	MOV B, L	70	MOV M, B
1B	DCX D	46	MOV B, M	71	MOV M, C
1C	INR E	47	MOV B, A	72	MOV M, D
1D	DCR E	48	MOV C, B	73	MOV M, E
1E	MVI E, d8	49	MOV C, C	74	MOV M, H
1F	RAR	4A	MOV C, D	75	MOV M, L
20	RIM	4B	MOV C, E	76	HLT
21	LXI H, d16	4C	MOV C, H	77	MOV M, A
22	SHLD addr16	4D	MOV C, L	78	MOV A, B
23	INX H	4E	MOV C, M	79	MOV A, C
24	INR H	4F	MOV C, A	7A	MOV A, D
25	DCR H	50	MOV D, B	7B	MOV A, E
26	MVI H, d8	51	MOV D, C	7C	MOV A, H
27	DAA	52	MOV D, D	7D	MOV A, L
28	---	53	MOV D, E	7E	MOV A, M
29	DAD H	54	MOV D, H	7F	MOV A, A
2A	LHLD addr16	55	MOV D, L	80	ADD B

## Appendix I continued...

OPCODE IN HEX	MNEMONIC		OPCODE IN HEX	MNEMONIC		OPCODE IN HEX	MNEMONIC	
81	ADD	C	AC	XRA	H	D7	RST	2
82	ADD	D	AD	XRA	L	D8	RC	
83	ADD	E	AE	XRA	M	D9	---	
84	ADD	H	AF	XRA	A	DA	JC	addr16
85	ADD	L	B0	ORA	B	DB	IN	addr8
86	ADD	M	B1	ORA	C	DC	CC	addr16
87	ADD	A	B2	ORA	D	DD	---	
88	ADC	B	B3	ORA	E	DE	SBI	d8
89	ADC	C	B4	ORA	H	DF	RST	3
8A	ADC	D	B5	ORA	L	E0	RPO	
8B	ADC	E	B6	ORA	M	E1	POP	H
8C	ADC	H	B7	ORA	A	E2	JPO	addr16
8D	ADC	L	B8	CMP	B	E3	XTHL	
8E	ADC	M	B9	CMP	C	E4	CPO	addr16
8F	ADC	A	BA	CMP	D	E5	PUSH	H
90	SUB	B	BB	CMP	E	E6	ANI	d8
91	SUB	C	BC	CMP	H	E7	RST	4
92	SUB	D	BD	CMP	L	E8	RPE	
93	SUB	E	BE	CMP	M	E9	PCHL	
94	SUB	H	BF	CMP	A	EA	JPE	addr16
95	SUB	L	C0	RNZ		EB	XCHG	
96	SUB	M	C1	POP	B	EC	CPE	addr16
97	SUB	A	C2	JNZ	addr16	ED	---	
98	SBB	B	C3	JMP	addr16	EE	XRI	d8
99	SBB	C	C4	CNZ	addr16	EF	RST	5
9A	SBB	D	C5	PUSH	B	F0	RP	
9B	SBB	E	C6	ADI	d8	F1	POP	PSW
9C	SBB	H	C7	RST	0	F2	JP	addr16
9D	SBB	L	C8	RZ		F3	DI	
9E	SBB	M	C9	RET		F4	CP	addr16
9F	SBB	A	CA	JZ	addr16	F5	PUSH	PSW
A0	ANA	B	CB	---		F6	ORI	d8
A1	ANA	C	CC	CZ	addr16	F7	RST	6
A2	ANA	D	CD	CALL	addr16	F8	RM	
A3	ANA	E	CE	ACI	d8	F9	SPHL	
A4	ANA	H	CF	RST	1	FA	JM	addr16
A5	ANA	L	D0	RNC		FB	EI	
A6	ANA	M	D1	POP	D	FC	CM	addr16
A7	ANA	A	D2	JNC	addr16	FD	---	
A8	XRA	B	D3	OUT	addr8	FE	CPI	d8
A9	XRA	C	D4	CNC	addr16	FF	RST	7
AA	XRA	D	D5	PUSH	D	--	---	
AB	XRA	E	D6	SUI	d8	--	---	

d8 → 8-bit data  
 d16 → 16-bit data  
 addr8 → 8-bit address

addr16 → 16-bit address  
 M → Memory  
 PSW → Program Status Word

## APPENDIX II : 8085 Instructions in Alphabetical Order

OPCODE IN HEX	MNEMONIC	OPCODE IN HEX	MNEMONIC	OPCODE IN HEX	MNEMONIC
CE	ACI d8	E4	CPO addr16	0A	LDAX B
8F	ADC A	CC	CZ addr16	1A	LDAX D
88	ADC B	27	DAA	2A	LHLD addr16
89	ADC C	09	DAD B	01	LXI B,addr16
8A	ADC D	19	DAD D	11	LXI D,addr16
8B	ADC E	29	DAD H	21	LXI H,addr16
8C	ADC H	39	DAD SP	31	LXI SP,addr16
8D	ADC L	3D	DCR A	7F	MOV A,A
8E	ADC M	05	DCR B	78	MOV A,B
87	ADD A	0D	DCR C	79	MOV A,C
80	ADD B	15	DCR D	7A	MOV A,D
81	ADD C	1D	DCR E	7B	MOV A,E
82	ADD D	25	DCR H	7C	MOV A,H
83	ADD E	2D	DCR L	7D	MOV A,L
84	ADD H	35	DCR M	7E	MOV A,M
85	ADD L	0B	DCX B	47	MOV B,A
86	ADD M	1B	DCX D	40	MOV B,B
C6	ADI d8	2B	DCX H	41	MOV B,C
A7	ANA A	3B	DCX SP	42	MOV B,D
A0	ANA B	F3	DI	43	MOV B,E
A1	ANA C	FB	EI	44	MOV B,H
A2	ANA D	76	HLT	45	MOV B,L
A3	ANA E	DB	IN addr8	46	MOV B,M
A4	ANA H	3C	INR A	4F	MOV C,A
A5	ANA L	04	INR B	48	MOV C,B
A6	ANA M	0C	INR C	49	MOV C,C
E6	ANI d8	14	INR D	4A	MOV C,D
CD	CALL addr16	1C	INR E	4B	MOV C,E
DC	CC addr16	24	INR H	4C	MOV C,H
FC	CM addr16	2C	INR L	4D	MOV C,L
2F	CMA	34	INR M	4E	MOV C,M
3F	CMC	03	INX B	57	MOV D,A
BF	CMP A	13	INX D	50	MOV D,B
B8	CMP B	23	INX H	51	MOV D,C
B9	CMP C	33	INX SP	52	MOV D,D
BA	CMP D	DA	JC addr16	53	MOV D,E
BB	CMP E	FA	JM addr16	54	MOV D,H
BC	CMP H	C3	JMP addr16	55	MOV D,L
BD	CMP L	D2	JNC addr16	56	MOV D,M
BE	CMP M	C2	JNZ addr16	5F	MOV E,A
D4	CNC addr16	F2	JP addr16	58	MOV E,B
C4	CNZ addr16	EA	JPE addr16	59	MOV E,C
F4	CP addr16	E2	JPO addr16	5A	MOV E,D
EC	CPE addr16	CA	JZ addr16	5B	MOV E,E
FE	CPI d8	3A	LDA d16	5C	MOV E,H

## Appendix II continued...

OPCODE IN HEX	MNEMONIC	OPCODE IN HEX	MNEMONIC	OPCODE IN HEX	MNEMONIC
5D	MOV E,L	C1	POP B	97	SUB A
5E	MOV E,M	D1	POP D	90	SUB B
67	MOV H,A	E1	POP H	91	SUB C
60	MOV H,B	F1	POP PSW	92	SUB D
61	MOV H,C	C5	PUSH B	93	SUB E
62	MOV H,D	D5	PUSH D	94	SUB H
63	MOV H,E	E5	PUSH H	95	SUB L
64	MOV H,H	F5	PUSH PSW	96	SUB M
65	MOV H,L	17	RAL	D6	SUI d8
66	MOV H,M	1F	RAR	EB	XCHG
6F	MOV L,A	D8	RC	AF	XRA A
68	MOV L,B	C9	RET	A8	XRA B
69	MOV L,C	20	RIM	A9	XRA C
6A	MOV L,D	07	RLC	AA	XRA D
6B	MOV L,E	F8	RM	AB	XRA E
6C	MOV L,H	D0	RNC	AC	XRA H
6D	MOV L,L	C0	RNZ	AD	XRA L
6E	MOV L,M	F0	RP	AE	XRA M
77	MOV M,A	E8	RPE	EE	XRI d8
70	MOV M,B	E0	RPO	E3	XTHL
71	MOV M,C	0F	RRC		
72	MOV M,D	C7	RST 0		
73	MOV M,E	CF	RST 1		
74	MOV M,H	D7	RST 2		
75	MOV M,L	DF	RST 3		
3E	MVI A, d8	E7	RST 4		
06	MVI B, d8	EF	RST 5		
0E	MVI C, d8	F7	RST 6		
16	MVI D, d8	FF	RST 7		
1E	MVI E, d8	C8	RZ		
26	MVI H, d8	98	SBB B		
2E	MVI L, d8	99	SBB C		
36	MVI M, d8	9A	SBB D		
00	NOP	9B	SBB E		
B7	ORA A	9C	SBB H		
B0	ORA B	9D	SBB L		
B1	ORA C	9E	SBB M		
B2	ORA D	DE	SBI d8		
B3	ORA E	22	SHLD addr16		
B4	ORA H	30	SIM		
B5	ORA L	F9	SPHL		
B6	ORA M	32	STA addr16		
F6	ORI d8	02	STAX B		
D3	OUT addr8	12	STAX D		
E9	PCHL	37	STC		

d8 → 8-bit data  
 d16 → 16-bit data  
 addr8 → 8-bit address

addr16 → 16-bit address  
 M → Memory  
 PSW → Program Status Word

### APPENDIX III : List of Microprocessors Released By INTEL

MICROPROCESSOR	DATE OF INTRODUCTION	NUMBER OF TRANSISTORS	CLOCK SPEED
4004	15 <sup>th</sup> Nov, 1971	2,300	400 kHz
8008	Apr, 1972	3,500	500-800 kHz
8080	Apr, 1974	4,500	2 MHz
8085	Mar, 1976	6,500	5 MHz
8086	8 <sup>th</sup> Jun, 1978	29,000	5/8/10 MHz
8088	Jun, 1979	29,000	5/8 MHz
80186	1982	....	10/12 MHz
80286	Feb, 1982	134,000	6/10/12 MHz
INTEL386 DX	17 <sup>th</sup> Oct, 1985	275,000	16/20/25/33 MHz
INTEL386 SX	16 <sup>th</sup> Jun, 1988	275,000	16/20/25/33 MHz
INTEL386 SL	15 <sup>th</sup> Oct, 1990	855,000	20/25 MHz
INTEL486 DX	10 <sup>th</sup> Apr, 1989	1.2 million	25/33/50 MHz
INTEL486 SX	16 <sup>th</sup> Sep, 1991	900,000	16/20/25 MHz
INTEL486 SX	21 <sup>st</sup> Sep, 1992	1.185 million	33 MHz
INTEL486 SL	4 <sup>th</sup> Nov, 1992	1.4 million	20/25/33 MHz
INTELDX 2	3 <sup>rd</sup> Mar, 1992	1.2 million	50/66 MHz
INTELDX 4	7 <sup>th</sup> Mar, 1994	3.2 million	75/100 MHz
Pentium	22 <sup>nd</sup> Mar, 1993	3.1 million	60/66 MHz
Pentium	7 <sup>th</sup> Mar, 1994	3.2 million	75/90/100/120 MHz
Pentium	Jun, 1995	3.3 million	133/150/166/200 MHz
Pentium Pro	1 <sup>st</sup> Nov, 1995	5.5 million	150/166/180/200 MHz
Pentium (MMX)	8 <sup>th</sup> Jan, 1997	4.5 million	166/200/233 MHz
Mobile Pentium (MMX)	9 <sup>th</sup> Sep, 1997	4.5 million	200/233/266/300 MHz
Pentium II	7 <sup>th</sup> May, 1997	7.5 million	233/266/300/333/350/400/450 MHz
Mobile Pentium II	2 <sup>nd</sup> Apr, 1998	7.5 million	233/266/300 MHz
Mobile Pentium II	25 <sup>th</sup> Jan, 1999	27.4 million	333/366/400 MHz
Pentium II Xeon	29 <sup>th</sup> Jun, 1998	7.5 million	400/450 MHz
Celeron	15 <sup>th</sup> Apr, 1998	7.5 million	266/300 MHz
Celeron	24 <sup>th</sup> Aug, 1998	19 million	333 MHz to 2.7 GHz
Mobile Celeron	25 <sup>th</sup> Jan, 1999	18.9 million	266 MHz to 2.4 GHz
Pentium III	26 <sup>th</sup> Feb, 1999	9.5 million	450/500/550/600 MHz

*Appendix III continued...*

MICROPROCESSOR	DATE OF INTRODUCTION	NUMBER OF TRANSISTORS	CLOCK SPEED
Pentium III	25 <sup>th</sup> Oct, 1999	28 million	500 MHz to 1 GHz
Pentium III Xeon	17 <sup>th</sup> Mar, 1999	9.5 million	500/550 MHz
Pentium III Xeon	25 <sup>th</sup> Oct, 1999	28 million	600 to 900 MHz
Mobile Pentium III	25 <sup>th</sup> Oct, 1999	28 million	400 MHz to 1 GHz
Mobile Pentium III	30 <sup>th</sup> Jul, 2001	44 million	1/1.06/1.13/1.2/1.33 GHz
Pentium 4	20 <sup>th</sup> Nov, 2000	42 million	1.4/1.5/1.6/1.7/1.8/1.9/2 GHz
Pentium 4	27 <sup>th</sup> Aug, 2001	55 million	2 to 2.8 GHz
Pentium 4 (HT Technology)	14 <sup>th</sup> Nov, 2002	55 million	2.4 to 3.3 GHz
Mobile Pentium 4	4 <sup>th</sup> Mar, 2002	55 million	1.5 to 3.2 GHz
INTEL Xeon	21 <sup>st</sup> May, 2001	42 million	1.4/1.5/1.7/2 GHz
INTEL Xeon	9 <sup>th</sup> Jan, 2002	52 million	1.8/2/2.2/2.4/2.6/2.8 GHz
INTEL Xeon	18 <sup>th</sup> Nov, 2002	108 million	1.4 to 3.2 GHz
INTEL Itanium	May, 2001	25 million	733/800 MHz
INTEL Itanium 2	8 <sup>th</sup> Jul, 2002	220 million	900 MHz/1 GHz
INTEL Itanium 2	30 <sup>th</sup> Jun, 2003	410 million	1/1.4/1.5 GHz
INTEL Pentium-M	12 <sup>th</sup> Mar, 2003	77 million	900 MHz to 1.7 GHz

*Note : The date mentioned here is the date of introduction of the lowest clock version of the processor. For the date of introduction of higher clock version of a processor please refer to INTEL website [www.intel.com](http://www.intel.com).*



## GENERAL INDEX

### A

Access time 3.40  
Accumulator 1.17  
ACI 2.18  
ADC 2.18  
ADC0809

- Functional block diagram 6.89
- Interfacing with 8085 6.91
- Pin description 6.88

ADD 2.18  
Address 1.1  
Address bus 1.1  
Addressing 2.11  
Addressing modes 2.11  
ADI 2.18  
ALE 1.9  
ALU 1.1  
ANA 2.18  
Analog to digital converter

- Conversion time of ADC 6.87
- Resolution of ADC 6.87

ANI 2.18  
Architecture

- of 6800 1.35
- of 8085 1.17
- of 8086 1.26
- of Z80 1.31

Arithmetic instructions 2.12  
Array 5.16  
Assembler

- One pass assembler 5.4
- Two pass assembler 5.4

Assembler directive 5.9  
Assembly language 5.1  
Asynchronous data transfer 6.3  
Auxiliary carry flag 1.17, 1.28

### B

Baud rate 6.46

Bit 1.1  
Branching instructions 2.12  
Buffers 3.12  
Bus

- CPU bus 1.2
- System bus 1.2

Bus contention 3.40  
Bus idle cycle 2.9  
Bus interface unit 1.26  
Byte 1.1

### C

CALL 2.18  
Carry flag 1.17, 2.18  
CC 2.18  
Chip select signal 3.12  
Clock 7.3, 1.2  
CM 2.18  
CMA 2.18  
CMC 2.18  
CMP 2.18  
CNC 2.18  
CNZ 2.18  
Common anode 7-segment LED 6.58  
Common cathode 7-segment LED 6.58  
Compiler 5.2  
Condition code register 1.34  
Constants 5.9  
Control bits 1.28  
Control bus 1.1  
Conversion time of ADC 6.87  
Conversion time of DAC 6.87  
CP 2.19  
CPE 2.19  
CPI 2.19  
CPO 2.19  
CPU 1.1  
CPU bus 1.2  
Cycle stealing DMA 6.95  
CZ 2.19

**D**

DAA 2.19  
 DAC0800  
   - Block diagram 6.85  
   - Interfacing with 8085 6.85  
   - Pin description 6.84  
 DAD 2.19  
 Data bus 1.1  
 Data direction register 6.21  
 Data transfer  
   - Asynchronous data transfer 6.3  
   - DMA data transfer 3.19, 6.2  
   - Interrupt driven data transfer 3.19, 6.4  
   - Programmed data transfer 3.19, 6.2  
   - Synchronous data transfer 6.4  
 Data transfer instructions 2.12  
 DB [Define Byte] 5.10  
 DCR 2.19  
 DCX 2.19  
 Debouncing 6.56  
 Debugger 5.5  
 Decoder  
   - 2-to-4 decoder 3.16  
   - 3-to-8 decoder 3.16  
 Delay routine 5.12  
 Demultiplexing 1.9  
 Development system 5.3  
 DI 4.4, 2.19  
 Digital to analog converter  
   - Conversion time of DAC 6.84  
   - Resolution of DAC 6.82  
 Direct addressing 2.11  
 Direction flag 1.28  
 Display interface using ports 6.58  
 Display refreshing 6.62  
 DMA controller [8237]  
   - Command register 6.29  
   - Control words 6.30  
   - Functional block diagram 6.25  
   - Interfacing with 8085 6.34  
   - Internal addresses 6.28

  - Internal registers 6.27  
   - Mask register 6.32  
   - Mode register 6.30  
   - Request register 6.31  
   - Pin description 6.24, 6.26  
   - Software commands 6.33  
   - Status register 6.32  
 DMA controller [8257]  
   - Control word 6.41  
   - Count register 6.41  
   - Functional block diagram 6.39  
   - Interfacing with 8085 6.42  
   - Internal addresses 6.39  
   - Mode set register 6.41  
   - Pin description 6.38  
   - Status register 6.41  
 DMA data transfer  
   - Block transfer DMA 3.20, 6.2  
   - Burst transfer DMA 6.2  
   - Cycle stealing DMA 3.20, 6.2  
   - Demand transfer DMA 3.20  
   - Single transfer DMA 6.24  
   - Verify transfer 6.30

Double word 1.1

DRAM 3.11

DW [Define Word] 5.10

**E**

Editor 5.4  
 EEPROM 3.12  
 EI 4.4  
 Emulator 5.6  
 END 5.10  
 EPROM 3.4  
 EQU 5.10  
 Execute cycle 2.1  
 Execution unit 1.26

**F**

Fetch cycle 2.1  
 FIFO RAM 6.65  
 First-last flip-flop 6.28

Flag 1.42  
 Flag register  
     - of 8085 1.17  
     - of 8086 1.28  
     - of Z80 1.32  
 Flowchart 5.2  
 Full duplex transmission 6.45

## H

Half duplex transmission 6.45  
 Hardware 2.57  
 Hardware interrupts 4.2, 4.23  
 Hardware polling 4.9  
 HCMOS 1.5  
 High impedance state 3.15, 1.2  
 High level language 2.1  
 HLT 2.19  
 HMOS 1.4

## I

IO device 3.19  
 IO mapping 3.22, 3.23  
 IO read cycle 2.4  
 IO structure 3.19  
 IO write cycle 2.4  
 Immediate addressing 2.11  
 Implied addressing 2.12  
 IN 2.19  
 INR 2.19  
 Instruction cycle 2.1  
 Instruction format 2.11  
 Instruction pointer 1.28  
 Instruction queue 1.26  
 Interpreter 5.64  
 Interrupt 4.1  
 Interrupt acknowledge cycle  
     - with CALL instruction 2.7  
     - with RST n instruction 2.5  
 Interrupt controller [8259]  
     - Functional block diagram 4.15  
     - Interfacing with 8085 4.13

    - Initialization command word 4.18  
     - Master 8259 4.17  
     - Operation comand word 4.19  
     - Pin description 4.13  
     - Slave 8259 4.17

Interrupt driven data transfer 3.19, 4.1

Interrupt flag 1.28

INTR

    - Expanding INTR 4.11

INX 2.19

Isolated IO mapping 3.22

ISR (Interrupt Service Routine) 6.4

## J

JC 2.19  
 JM 2.19  
 JMP 2.19  
 JNC 2.19  
 JNZ 2.19  
 JP 2.19  
 JPE 2.19  
 JPO 2.19  
 JZ 2.19

## K

Keyboard/Display controller [8279]  
     - Block diagram 6.66  
     - Command words 6.68  
     - Interfacing with 8085 6.70  
     - Keycode entry in FIFO 6.67  
     - Pin description 6.65  
 Keyboard interface using ports 6.56  
 Keyboard scanning 7.1, 6.56  
 Keyboard switch 6.55  
 Key debouncing 6.56

## L

Latches 6.63  
 LDA 2.19  
 LDAX 2.19  
 LHLD 2.19  
 Library builder 5.5

LIFO 5.17  
 Linker 5.5  
 List 5.15  
 Locator 5.67  
 Logical instructions 2.12  
 LXI 2.19

## M

Machine control instructions 2.12  
 Machine cycle  
   - with wait state 2.9  
 Machine language 5.1  
 Macro 5.11  
 Mainframes 1.10  
 Main memory 3.1  
 Maskable interrupt 4.3  
 Master 8259 4.17  
 MAX232  
   - Pin description 6.49  
   - Typical circuit connection 6.49  
 Mega flops 1.40  
 Memory 3.38  
 Memory access time 3.40  
 Memory mapping 3.23  
 Memory mapping of IO device 3.22  
 Memory organization 3.16  
 Memory read cycle 2.3  
 Memory word size 3.13, 1.1  
 Memory write cycle 2.3  
 Microcomputers 1.7, 1.10  
 Microprocessor 1.6  
 Minicomputers 1.10  
 Minimum system 7.3  
 MIPS 1.5  
 Mnemonics 5.1  
 MODEM 6.46  
 MOV 2.19  
 Multiple word 1.1  
 Multiplexed display 7.1  
 Multiplexing 1.9  
 MVI 2.19

## N

NDRO 3.3  
 Nibble 1.1  
 Non-maskable interrupt 4.3  
 Non-vectored interrupt 4.2  
 Non-volatile memory 3.3  
 NOP 2.19  
 NVRAM 3.12

## O

Opcode 2.2  
 Opcode fetch cycle 2.2  
 Operand 2.59  
 ORA 2.19  
 ORG 5.10  
 ORI 2.19  
 OUT 2.19  
 Overflow flag 1.28

## P

Parallel data transfer 6.1  
 Parity flag 1.7, 1.28  
 PCHL 2.19  
 Pentium 1.5  
 Persistent of vision 6.61  
 Physical memory space 3.40  
 Polling 4.2, 4.7  
 POP 2.20  
 Port 3.41  
 Primary memory 3.1  
 Procedure 5.11  
 Processor cycle 2.1  
 Processor memory 3.1  
 Program 5.64  
 Program counter 1.19  
 Programmable IO port and timer [8155]  
   - Control word 6.8  
   - Interfacing with 8085 6.11, 6.12  
   - Internal address 6.6  
   - Internal block diagram 6.7  
   - Pin description 6.7

- Status register 6.10

- Timer modes 6.10

Programmable interrupt controller [8259] 4.12

Programmable peripheral devices 6.1

Programmable peripheral interface [8255]

- Control words 6.16

- Interfacing with 8085 6.14

- Internal address 6.14

- Internal block diagram 6.13

- IO modes 6.12

- Pin description 6.13

- Port-C pin assignment 6.17

- Status word 6.18

Programmable timer [8254] 6.72

Programmed data transfer 3.19

PROM 3.3

PSW (Program Status Word) 1.17

PUSH 2.20

## R

RAL 2.20

Random access memory 3.3

RAR 2.20

RC 2.20

Read only memory 3.3

Read/write memory 3.3

Register addressing 2.12

Register indirect addressing 2.12

Resolution of ADC 6.87

Resolution of DAC 6.82

RET 2.20

RIM 4.4, 2.20

RLC 2.20

RM 2.20

RNC 2.20

RNZ 2.20

ROM 3.3

Rotating priority 6.40, 6.42, 4.22

RP 2.20

RPE 2.20

RPO 2.20

RRC 2.20

RS-232C serial bus 6.47

RST 4.12

RST 5.5 4.5

RST 6.5 4.5

RST 7.5 4.5

RZ 2.20

## S

SBB 2.20

SBI 2.20

Secondary memory 3.1

Semiconductor memory 3.1

Serial data transfer 6.45

Seven segment LED

- Common anode 6.58

- Common cathode 6.58

Shadow registers 3.12, 1.44

SHLD 2.20

Sign flag 1.7, 1.28

SIM 4.4, 2.20

Simplex transmission 6.45

Simulator 5.6

Single step trap 1.28

Slave [8259] 4.17

Software 2.1

Software interrupts 4.2

Software polling 4.8

Speed power product 1.4

SPHL 2.20

STA 2.20

Stack 5.16

Stack pointer 5.16, 1.19

Standard IO mapping 3.22

Static RAM 3.7

STAX 2.20

STC 2.20

SUB 2.20

Subprogram 5.11

Subroutine 5.11

Successive approximation ADC 6.87

SUI 2.20  
Supercomputer 1.40  
Synchronous data transfer 6.4  
System bus 1.2  
System clock 7.2

## T

Timer [8254]  
- Control word 6.75  
- Functional block diagram 6.73  
- Interfacing with 8085 6.74  
- Operating modes 6.76  
- Pin description 6.73  
- Status word 6.76  
Timing diagram 2.2  
Trace flag 1.28  
TRAP 4.4  
Tristate logic 1.2  
T-state 1.24

## U

UART 6.46  
Unconditional CALL 2.44  
Unconditional jump 2.43  
USART 6.46  
USART [8251A]  
- Command word 6.53  
- Functional block diagram 6.51  
- Interfacing with 8085 6.54  
- Mode word 6.53  
- Pin description 6.50  
- Status word 6.53

## V

Variable 5.8  
Vector address 4.2  
Vectored interrupt 4.2  
Vectoring 4.26  
Verify transfer 6.30  
Volatile memory 3.3

## W

Wait State 2.9  
Word  
- Double word 1.1  
- Multiple word 1.1

## X

XCHG 2.20  
XRA 2.20  
XRI 2.20  
XTHL 2.20

## Z

Z80 1.29  
Zero flag 1.17, 1.28

## CHIP INDEX

### 2764

- Logic block diagram 3.5
- Pin description 3.5

### 6264

- Logic block diagram 3.8
- Pin description 3.8

### 6800

- Architecture 1.35
- Flags 1.34
- Pin description 1.33

### 74LS138

- Pin description 3.16
- Truth table 3.16

### 74LS139

- Pin description 3.16
- Truth table 3.16

### 8085

- Architecture 1.17
- Flags 1.17
- Pin description 1.13

### 8086

- Architecture 1.26
- Flags 1.28
- Pin description 1.20

### 8155

- Control word 6.8
- Interfacing with 8085 6.11, 6.12
- Internal address 6.6
- Internal block diagram 6.7
- Pin description 6.7
- Status register 6.10
- Timer modes 6.10

### 8212

- Internal block diagram 6.5
- Input control logic 6.6
- Output control logic 6.5
- Pin description 6.5

### 8237

- Command register 6.29
- Control words 6.30
- Functional block diagram 6.25
- Interfacing with 8085 6.34
- Internal addresses 6.28
- Internal registers 6.29
- Mask register 6.32
- Mode register 6.30
- Pin description 6.24, 6.26
- Request register 6.31
- Software commands 6.33
- Status register 6.32

### 8251

- Command word 6.53
- Functional block diagram 6.51
- Interfacing with 8085 6.54
- Mode word 6.53
- Pin description 6.50
- Status word 6.53

### 8254

- Control words 6.75
- Functional block diagram 6.73
- Interfacing with 8085 6.74
- Operating modes 6.76
- Pin description 6.73
- Status word 6.76

### 8255

- Control words 6.16

- Interfacing with 8085 6.14
- Internal address 6.14
- Internal block diagram 6.14
- IO modes 6.12
- Pin description 6.13
- Port-C pin assignment 6.17
- Status word 6.18

**8257**

- Control word 6.41
- Count register 6.41
- Functional block diagram 6.39
- Interfacing with 8085 6.42
- Internal addresses 6.39
- Mode set register 6.41
- Pin description 6.38
- Status register 6.41

**8259**

- Functional block diagram 4.15
- Initialization command word 4.18
- Interfacing with 8085 4.13
- Master 8259 4.17
- Operation command word 4.19
- Pin description 4.13
- Slave 8259 4.17

**8279**

- Block diagram 6.66
- Command words 6.68
- Interfacing with 8085 6.70
- Keycode entry in FIFO 6.67
- Pin description 6.65

**8355**

- Data direction register 6.21
- Internal address 6.20
- Internal block diagram 6.20
- Pin description 6.20

**8755**

- Data direction register 6.22
- Internal address 6.22
- Internal block diagram 6.21
- Pin description 6.21

**ADC0809/0808**

- Functional block diagram 6.89
- Interfacing with 8085 6.91
- Pin description 6.88

**DAC0800**

- Block diagram 6.85
- Interfacing with 8085 6.85
- Pin description 6.84

**MAX232**

- Pin description 6.49
- Typical circuit connection 6.49

**Z80**

- Architecture 1.31
- Flags 1.32
- Pin description 1.30