The McGraw·Hill Companies

8086 MICROPROCESSOR AND ITS APPLICATIONS

Second Edition

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.

The McGraw·Hill Companies

8086 MICROPROCESSOR AND ITS APPLICATIONS

Second Edition

A. Nagoor Kani

Founder RBA Group
Chennai

Tata McGraw Hill Education Private Limited
NEW DELHI

McGraw-Hill Offices

New Delhi New York St Louis San Francisco Auckland Bogotá Caracas
Kuala Lumpur Lisbon London Madrid Mexico City Milan Montreal
San Juan Santiago Singapore Sydney Tokyo Toronto

The McGraw·Hill Companies

Tata McGraw-Hill

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

8086 Microprocessor and Its Applications, 2e

ISBN-13: 978-0-07-107767-5
ISBN-10: 0-07-107767-7

Vice President and Managing Director—MHE: *Ajay Shukla*

Head—Higher Education Publishing and Marketing: *Vibha Mahajan*
Publishing Manager—SEM & Tech Ed.: *Shalini Jha*
Assistant Sponsoring Editor: *Smruti Snigdha*
Editorial Researcher: *Sourabh Maheshwari*
Copy Editor: *Preyoshi Kundu*
Senior Production Manager: *Satinder S Baveja*
Proof Reader: *Yukti Sharma*

Marketing Manager–Higher Education: *Vijay Sarathi*
Senior Product Specialist—SEM & Tech Ed.: *Tina Jajoriya*
Graphic Designer–Cover: *Meenu Raghav*

General Manager-Production: *Rajender P Ghansela*
Production Manager: *Reji Kumar*

Information contained in this work has been obtained by Tata McGraw-Hill, from sources believed to be reliable. However, neither Tata McGraw-Hill nor its authors guarantee the accuracy or completeness of any information published herein, and neither Tata McGraw-Hill nor its authors shall be responsible for any errors, omissions, or damages arising out of use of this information. This work is published with the understanding that Tata McGraw-Hill and its authors are supplying information but are not attempting to render engineering or other professional services. If such services are required, the assistance of an appropriate professional should be sought.

Typeset at Tej Composers, WZ-391, Madipur, New Delhi 110063, and

Cover printed at:

The McGraw·Hill Companies

Dedicated to my

Brother-in-law: Mr. M. Kadher Mydhen

Sister: Mrs. A. Fareedha Begum

Their Son: K. Mohammed Noufel

CONTENTS

CHAPTER - 1 INTRODUCTION

CHAPTER - 2 INTEL 8086 PINS, SIGNALS AND ARCHITECTURE

CHAPTER - 3 INSTRUCTION SET OF 8086

CHAPTER - 4 MEMORY AND IO INTERFACING

CHAPTER - 5 INTERRUPTS

CHAPTER-8 INTEL 80X86 FAMILY OF PROCESSORS

CHAPTER - 9 8086 MICROPROCESSOR-BASED SYSTEM

Preface

About the Book

8086 Microprocessor and its Applications, 2e is a book on microprocessor 8086 programming and interfacing which has been crafted and designed to meet students' 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 algorithms and flowcharts. The main objective of this book is to explore the basic concepts of popular INTEL 8086 microprocessor programming and interfacing techniques in a simple and easy-to-understand manner.

This book with its lucid writing style and germane pedagogical features will prove to be a master text for engineering students and practitioners. The peripheral interfacing techniques have 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 attempts to explain the basic concepts of programming and interfacing techniques by taking INTEL 8086 microprocessor as an example. It includes the system design applications based on 8086 Microprocessor. It discusses the concepts using numerous examples and programs and step-wise approach which makes it easier for the readers to grasp the concepts. The book has been designed as a self-study material for the students of engineering, polytechnic, arts and science colleges.

Salient Features

- Lucid and easy language for concept explanation
- Extensive coverage to Instruction sets, Memory and Peripheral Interfacing of 8086 Microprocessor
- Discusses programming concepts for 8086 using assembly language
- Summary at the end of each chapter
- Use of simple methodology (i.e., Problem analysis→Flowchart→Algorithm→Code →Sample Data) for programming examples
- Numerous solved examples, programs and chapter-end questions with answers

Chapter Organization

The book is organized into 9 chapters and 2 appendices.

Chapter 1 briefs about the evolution of microprocessor and basics of microprocessor-based system. The technical details of INTEL 8086 microprocessor like pin details, explanation of various signals of the processor, internal architecture, bus cycles and their timing diagrams are presented in **Chapter 2.**

Chapter 3 is fully devoted to the instruction set of 8086 processor. In this chapter, the format of 8086 instruction and the addressing modes of 8086 instructions are explained in detail. A brief discussion about semiconductor memory and their interfacing with 8086 microprocessor are presented in **Chapter 4**. Design examples are also included for better understanding of the concept of the memory and IO interfacing with 8086 microprocessor.

The importance of interrupts and the various interrupts of 8086 processor are discussed in **Chapter 5**. The implementation of interrupt scheme and expansion of interrupts in 8086 system are also presented in this chapter.

The concepts of assembly language programming are discussed in **Chapter 6**. A number of assembly language programs using 8086 microprocessor instructions are included in this chapter. The concepts of DOS and BIOS services, and writing assembly language programs using these services are also presented in this chapter. The example programs presented in this book are assembled using MASM assembler and verified in RBA-8086 trainer kit and personal computer.

The concepts of keyboard and 7-segment display and their interfacing are discussed in **Chapter 7**. Simple discussions about USART, DMA controllers, programmable timer, ADC and DAC, and their interfacing with 8086 microprocessor are also presented in Chapter 7.

A brief discussion about the architectures of 80x86 and Pentium family of processors are presented in **Chapter 8**. The 8086 microprocessor-based system design concepts are discussed in **Chapter 9**.

The instruction templates of 8086 microprocessor instructions are listed in Appendix-I and the BIOS and DOS interrupts are listed in Appendix-II for the use of assembly language programmers.

I have taken utmost care to present the concepts of INTEL 8086 microprocessor in a simple manner and I hope that the teaching and student community will welcome the book. The readers can feel free to convey their criticism and suggestions to *kani@vsnl.com* for further improvement of the book.

A. Nagoor Kani

Publisher's Note

Remember to write to us. We look forward to receiving your feedback, comments and ideas to enhance the quality of this book. You can reach us at *tmh.csefeedback@gmail.com*. Kindly mention the title and the author's name as the subject.

In case you spot piracy of this book, please do let us know.

Acknowledgements

I express my heartfelt thanks to my wife Mrs. 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 the manuscript, and Ms. A. Selvi, Ms. M. Faritha for typesetting and preparing the layout of the book.

My sincere thanks to all the reviewers for their valuable suggestions and comments which helped me in exploring the subject to a greater depth.

I am also grateful to Ms. Vibha Mahajan, Mr. Ebi John, Ms. Smruti Snigdha, Mr. Sourabh Maheshwari, Ms. Preyoshi Kundu, and Ms. Yukit Sharma of Tata McGraw Hill Education for their concern and care in publishing this work. My special thanks to Ms Smruti Snigdha of McGraw Hill Education for her care in bringing out this work at the right time. I thank all my office staff for their cooperation 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

CHAPTER 1

INTRODUCTION

1.1 TERMS USED IN MICROPROCESSOR LITERATURE

Bit : A digit of the binary number or code is called bit.

Nibble : The 4-bit (4-digit) binary number or code is called nibble.

Byte : The 8-bit (8-digit) binary number or code is called byte.

Word : The 16-bit (16-digit) binary number or code is called word.

Double Word : The 32-bit (32-digit) binary number or code is called double word.

Multiple Word : The 64, 128, ... bit/digit binary numbers or codes are called multiple words.

Data : 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.

Address : Address is an identification number in binary for memory locations. The 8086 processor uses a 20-bit address for memory.

Memory Word Size (or Addressability) : 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 an 8086 processor-based system is 8-bit.

[Address and program codes in a 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 addresses.]*

Microprocessor : The microprocessor is a program controlled semiconductor device(IC), which fetches (from memory), decodes and executes instructions. It is used as CPU (**C**entral **P**rocessing **U**nit) in computers.

The basic functional blocks of a microprocessor are ALU (**A**rithmetic **L**ogic **U**nit), an array of registers and a control unit. The microprocessor is identified with the size of data, the ALU of the processor can work with at a time. The 8086 processor has a 16-bit ALU, hence it is called a 16-bit processor. The 80486 processor has a 32-bit ALU, hence it is called a 32-bit processor.

Bus : A bus is a group of conducting lines that carries data, address 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 address is called address bus.

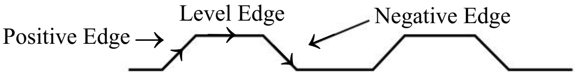
The group of conducting lines that carries control signals is called control bus.

CPU Bus : The group of conducting lines that are directly connected to the microprocessor is called CPU bus. In a CPU bus, the signals are multiplexed, i.e., more than one signal is passed through the same line but at different timings.

System Bus : The group of conducting lines that carries data, address and control signals in a microcomputer system is called System bus. Multiplexing is not allowed in a system bus.

[In microprocessor-based systems, each bit of information (data/address/control signal) is sent 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 a multiplexed CPU bus. For example, in an 8086 processor the address and data are sent through the 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 bus, separate conducting lines will be provided for each bit of data, address and control signals.]

Clock : A clock is a square wave 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.



A clock has three edges : rising edge (positive edge), level edge and falling edge (negative edge). The device is made sensitive to any one of the edges for better functioning (it means that the device will recognize the clock only when the edge is asserted or arrived).

Tristate Logic : Almost all the devices used in a microprocessor-based system use tristate logic. In devices with tristate logic, three logic levels will be available : **High** state, **Low** state and **High impedance** state.

The **high** and **low** level states are normal logic levels for data, address or control signals. The **high impedance** state is an 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 tells us that it was the ancient Babylonians who first began using the abacus (a primitive calculator made of beads) in about 500 B.C. This simple calculating machine eventually sparked human mind into the development of calculating machine that use 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 **I**ntegrated **C**ircuit (IC) led to development of the microprocessor and microprocessor-based computer systems.

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 (**T**ransistor **T**ransistor **L**ogic) levels.

The first generation processors required a lot of additional support ICs to form a system, sometimes as high as 30 ICs. 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	
● NATIONAL IMP/16	16-bit processors
● NATIONAL PACE	

Second Generation Microprocessors

The second generation microprocessors appeared in 1973 and were manufactured using the 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
- INTEL 8085
- FAIRCHILD F - 8
- MOTOROLA M6800
- MOTOROLA M6809
- NATIONAL CMP -8
- RCA COSMAC
- MOS TECH. 6500
- SIGNETICS 2650
- ZILOG Z80

8-bit processors

- INTERSIL 6100
- TOSHIBA TLCS - 12

12 -bit processors

- TI TMS 9900
- DEC - W.D. MCP - 1600
- GENERAL INSTRUMENT CP 1600
- DATA GENERAL µN601

16- bit processors

Characteristics of second generation microprocessors

- Larger chip size (170 × 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 (**H**igh density **MOS**) technology. Some of the third generation microprocessors are given below:

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

The HMOS technology offers better **S**peed **P**ower **P**roduct (SPP) and higher packing density than NMOS.

Speed Power Product = Speed × Power
= Nanosecond × Milliwatt
= Picojoules

- 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 those of NMOS.
  Packing density of NMOS = 1852.5 gates/mm^2
  Packing density of HMOS = 4128 gates/mm^2 (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. These generation processors are 32-bit processors and are fabricated using the low-power version of the HMOS technology called HCMOS. These 32-bit microprocessors have increased sophistications that compete strongly with 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 latest processor by INTEL is the **pentium** which is considered as a fifth generation processor. The pentium is a 32-bit processor with 64-bit data bus and is available in a wide range of clock speeds from 60 MHz to 3.2 GHz. With improvement in semiconductor technology, the processing speed of microprocessors has increased tremendously. The 8085 released in the year 1976 executes 0.5 **M**illion **I**nstructions **P**er **S**econd (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 1 GHz clock can execute 2000 **M**illion **I**nstructions **P**er **S**econd (2000 MIPS). The various processors released by INTEL are listed in Appendix-III.

1.3 BASIC FUNCTIONAL BLOCKS OF A MICROPROCESSOR

A 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, **P**rogram **C**ounter (PC)/**I**nstruction **P**ointer (IP), Instruction decoding unit, Timing and Control unit.

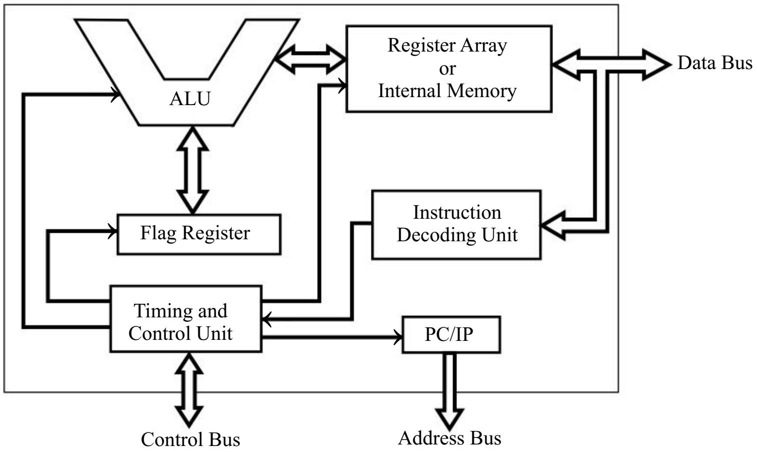


Fig. 1.1 : Block diagram showing basic functional blocks of a microprocessor.

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 sign flag. One of the bit position of the flag register is called sign flag and it is used to store the status of the sign of the result of the 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 its manufacturer. For doing any useful work with the microprocessor, we have to first write a program using these instructions, and store them in a memory device external to the microprocessor.

The instruction pointer generates the address of the instructions to be fetched from the memory and sends it through the 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 it sends information to the timing and control unit. The data is stored in the register array for processing by the ALU.

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

1.4 MICROPROCESSOR-BASED SYSTEMS (ORGANIZATION OF A MICROCOMPUTER)

A microprocessor is a semiconductor device (or Integrated Circuit) manufactured by the VLSI (**V**ery **L**arge **S**cale **I**ntegration) 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 the microprocessor as a CPU (**C**entral **P**rocessing **U**nit) and interfacing memory, the input and output devices to it. A system designed using a microprocessor as its CPU is called a microcomputer or single board microcomputer. A microprocessor-based system consists of a microprocessor as the CPU, semiconductor memories like EPROM and RAM, an input device, an 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	Static RAM
INTEL 2708 (1 kb)	MOTOROLA 6208 (1 kb)
INTEL 2716 (2 kb)	MOTOROLA 6216 (2 kb)
INTEL 2732 (4 kb)	MOTOROLA 6232 (4 kb)
INTEL 2764 (8 kb)	MOTOROLA 6264 (8 kb)

Note : kb refer to kilo bytes.

The popular input devices are the 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 the operations.

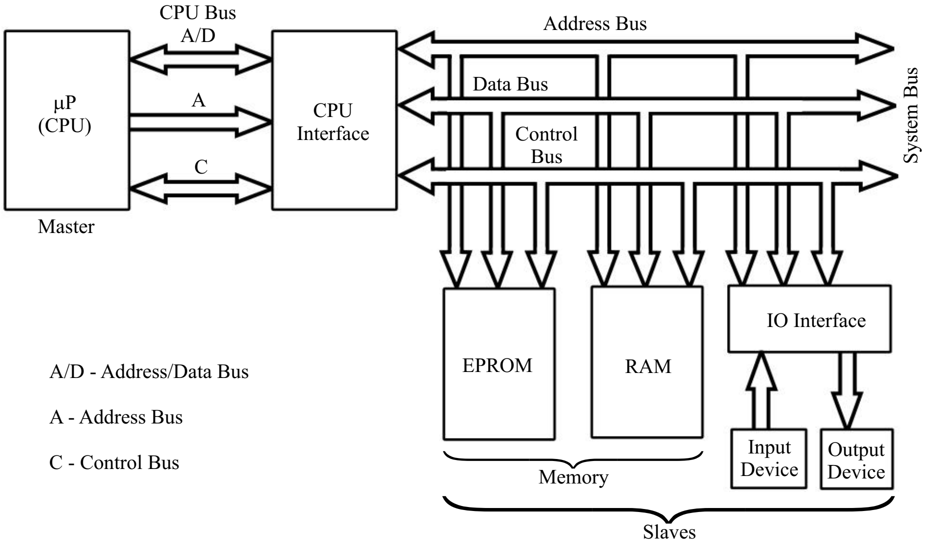


Fig. 1.2 : Microprocessor-based system (organization of microcomputer).

Buses are a group of lines that carry data, address 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.

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 remain in a **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 used for examining the results. Since the speed of IO devices does not match with the speed of the microprocessor, an interface device is provided between the system bus and the 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 the memory. The program consists of a set of instructions stored in consecutive memory locations. 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 the processor decodes the instruction and carries out the task specified by the instruction.

1.5 CONCEPT OF MULTIPLEXING IN MICROPROCESSORS

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

Due to 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 (e.g., in 8086, the lower 16-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 INTEL 8086).

Whenever multiplexing is used, the CPU interface of the system must include the necessary hardware to demultiplex those lines to produce 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 an 8086 Processor

In order to demultiplex the address/data lines (of the processor), the processor provides a signal called ALE (**A**ddress **L**atch **E**nable). The ALE is asserted **high** and then **low** by the processor at the beginning of every bus cycle. At the same time, the address is given out through AD_0 - AD_{15} lines and A_{16} - A_{19}/status lines. Demultiplexing of address/data lines and address/status lines using 8-bit D-latch 74LS373 is shown in Fig. 1.3.

The ALE is connected to the **En**able Pin (EN) of the external 8-bit latches. When ALE is asserted **high** and then **low**, the addresses are latched into the output lines of the latch. It holds the address until the next bus cycle. After latching the address, the AD_0 - AD_{15} lines are free for data transfer and A_{16} - A_{19}/status lines are free for carrying status information. The first T-state of every bus cycle is used for address latching in 8086 and the remaining T states are used for reading or writing operation.

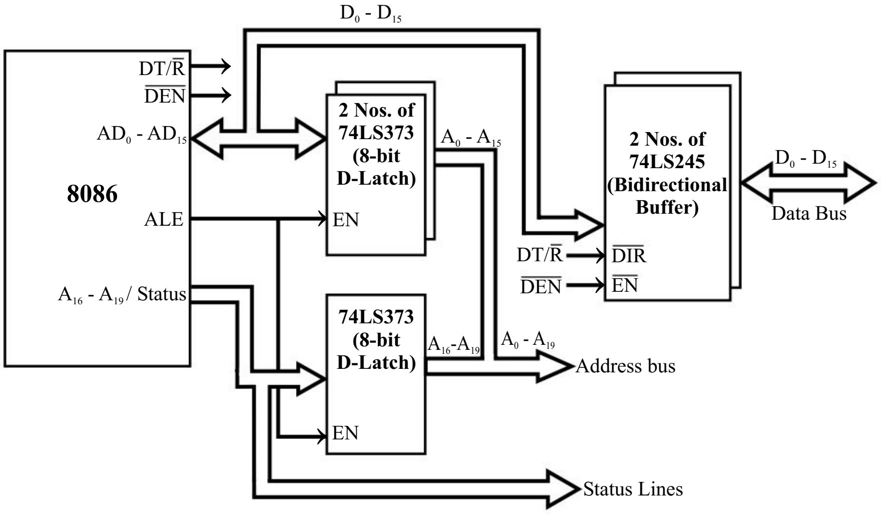


Fig. 1.3 : Demultiplexing of address and data lines in an 8086 processor.

The data bus is provided with a bidirectional buffer in order to drive the data to a longer distance in the bus. The 8086 provides two control signals DT/$\overline{R}$ and $\overline{DEN}$ for controlling the data buffers. The DT/$\overline{R}$ is used to decide the direction of data flow and $\overline{DEN}$ is used to enable the data buffer.

1.6 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 ?

A bus is a group of conducting lines that carries data, address 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 bidirectional?

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

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

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

In 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 an 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 the address and chip select signal. When the slave is selected, it comes to normal logic and it communicates with the master.

1.10 What is HMOS and HCMOS?

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

HCMOS is High density n-type Complementary Metal Oxide Silicon field effect transistor. It is a 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?

First generation processors are fabricated using PMOS technology and they have drawbacks like slow speed, low output currents and are not compatible with TTL logic levels.

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

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

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

A microprocessor is the master of a system, which controls all the activities of the system. It issues address and control signals and fetches the instruction and data from the memory. It also executes the instructions to take appropriate action.

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

A microprocessor-based system consists of a microprocessor as the CPU, semiconductor memories like EPROM and RAM, input device, output device and interfacing devices.

1.15 Why is interfacing needed for IO devices?

Generally, IO devices are slow devices. Therefore, the speed of IO devices do not match with the speed of the microprocessors. Therefore, an interface is provided between the 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 the system bus.)

1.17 What is multiplexing and what is its advantage?

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

CHAPTER 2

INTEL 8086 PINS, SIGNALS AND ARCHITECTURE

2.1 INTRODUCTION TO INTEL 8086

INTEL 8086 is the first 16-bit processor released by INTEL in the year 1978. The 8086 was designed using the HMOS technology and it is now 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-V supply.

The 8086 does not have an 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 Mega) of memory space. One mega-byte (1 Mb) of addressable memory space of 8086 are organized as two memory banks of 512 kilo bytes each (512 kb + 512 kb = 1 Mb). The memory banks are called even (or lower) bank and odd (or upper) bank. The address line A_0 is used to select the even bank and the control signal $\overline{BHE}$ is used to select the odd bank.

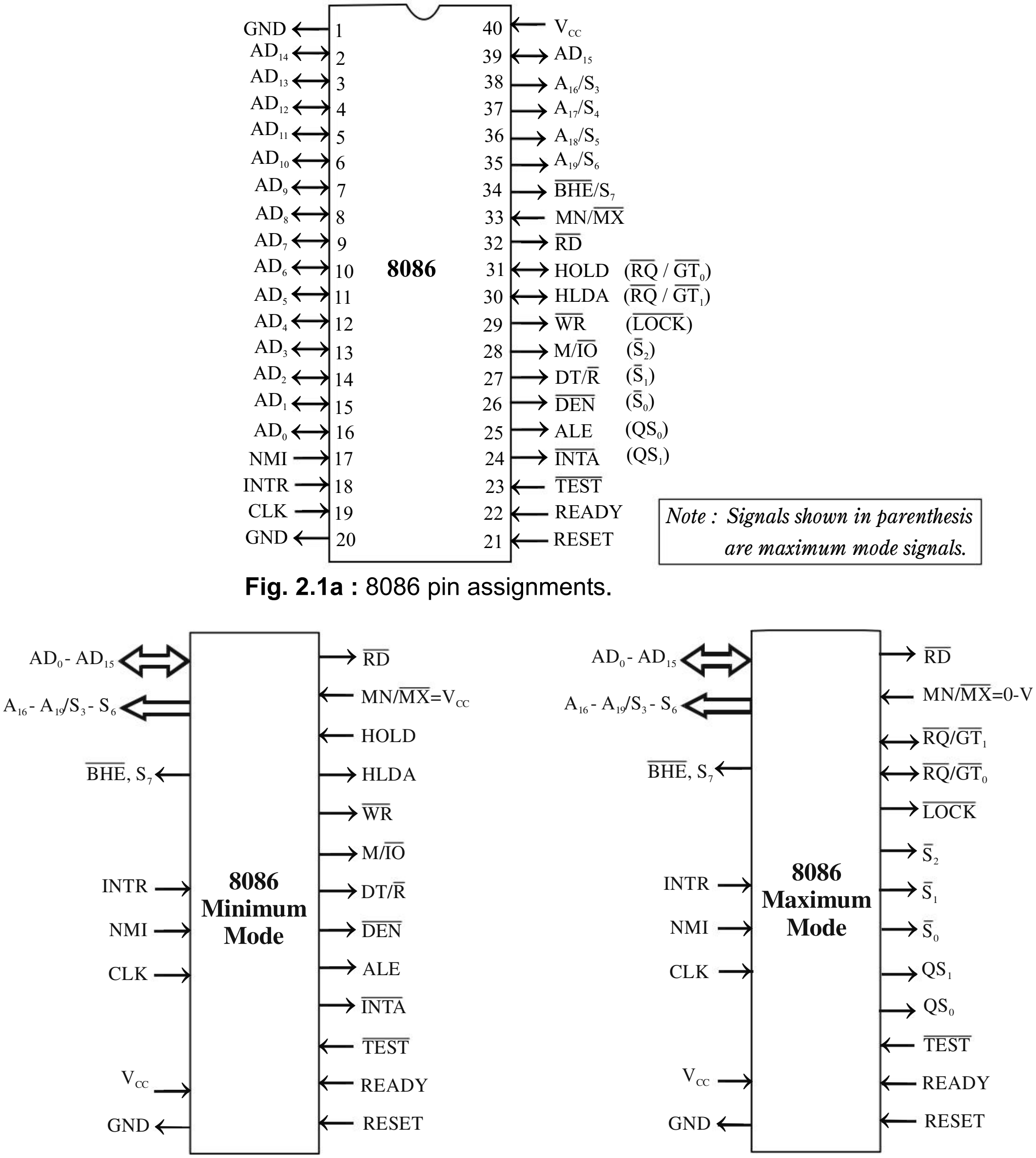
For accessing IO-mapped devices, the 8086 uses a separate 16-bit address, and so the 8086 can generate 64 k(2^{16}) IO addresses. The signal $M/\overline{IO}$ is used to differentiate the memory and IO addresses. For memory address, the signal $M/\overline{IO}$ is asserted **high** and for IO address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

The 8086 can operate in two modes: 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 a uniprocessor system. When $MN/\overline{MX}$ is tied **low**, it works in maximum mode and the system is called a 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 the 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 the 8086 uses 16-bit data bus externally. The 8086 accesses memory in words but the 8088 accesses memory in bytes. IBM designed its first **P**ersonal **C**omputer (PC) using an INTEL 8088 microprocessor as the CPU.

2.2 PINS AND SIGNALS OF INTEL 8086

The 8086 pins and signals are shown in Fig. 2.1. 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. 2.1a : 8086 pin assignments.

Fig. 2.1b : 8086-Minimum mode.

Fig. 2.1c : 8086-Maximum mode.

Fig. 2.1 : 8086 pin and signals.

TABLE - 2.1 : 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 state control	Input
RESET	System reset	Input
NMI	Non-maskable interrupt request	Input
INTR	Interrupt request	Input
CLK	System clock	Input
V_{cc}	+ 5-V	Power supply input
GND	Ground	Power supply ground

TABLE - 2.2 : 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 - 2.3 : MAXIMUM MODE SIGNALS [MN/$\overline{MX}$ = 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 common signals for minimum and maximum mode are listed in Table-2.1. The lower sixteen lines of the address are multiplexed with data and the upper four lines of the 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 periods of a bus cycle, the data and status signals will be available on these lines.

The status signals on S_3 and S_4 specify 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-2.4.

TABLE - 2.4 : 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 up to 4 Mb. The status line S_5 indicates the status of the 8086 interrupt enable flag. A **low** on the line S_6 indicates that the 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 $\overline{\text{BHE}}$ on the first T-state of a bus cycle is maintained as status signal S_7 during all other T states of the bus cycles.

The 8086 outputs a **low** on the $\overline{\text{BHE}}$ pin during read, write and interrupt acknowledge cycles when the data is to be transferred to the high order data bus. The $\overline{\text{BHE}}$ can be used in conjunction with address bit $A_0(AD_0)$ to select memory banks.

When the processor reads from the memory or an IO location it asserts $\overline{\text{RD}}$ as **low**. The $\overline{\text{TEST}}$ 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 $\overline{\text{TEST}}$ is made **low** by an external hardware. This is used to synchronize an external activity to the processor's internal operation. $\overline{\text{TEST}}$ 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 non-maskable 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 microseconds. For reset while working, it is held **high** for at least four clock cycles. When the processor is reset, the DS, SS, ES, IP and flag register are cleared, **C**ode **S**egment (CS) register is initialized to $FFFF_H$ and queue is emptied. After reset, the processor will start fetching instructions from the 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 the basic timing for the 8086 and bus controller. The 8086 does not have an 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 an 8086 must be connected to the 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 output 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 signals to the 8086.

Minimum Mode Signals

The minimum mode signals of an 8086 are listed in Table-2.2. 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:

DT/$\overline{\textbf{R}}$ - (***D****ata* ***T****ransmit/****R****eceive*). It is an output signal from the processor to control the direction of data flow through the data transceivers.

$\overline{\textbf{DEN}}$ - (***D****ata* ***En****able*) - It is an output signal from the processor used as output enable for the data transceivers.

ALE - (***A****ddress* ***L****atch* ***E****nable*) - It is used to demultiplex the address and data lines using external latches.

M/$\overline{\textbf{IO}}$ - 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.**

$\overline{\textbf{WR}}$ - It is a write control signal and it is asserted **low** whenever the processor writes data to memory or IO port.

$\overline{\textbf{INTA}}$ - (***Int****errupt* ***A****cknowledge*) - 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 - (***H****o****ld*** ***A****cknowledge*) - 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-2.3. 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 the bus timing and control signals. The status signals are decoded as shown in Table 2.5.

TABLE - 2.5 : 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

$\overline{\text{RQ}}/\overline{\text{GT}}_0$, $\overline{\text{RQ}}/\overline{\text{GT}}_1$ - (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 GT_0 will have higher priority than GT_1.

The bus request to 8086 work as follows:

1. When a local bus master requires system bus control, it sends a low pulse to the 8086.
2. At the end of the 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 requested the bus control.
3. On receiving the acknowledge the local master will take control of the system bus. After completing its work, at the end, the local bus master sends a low signal on the same pin to the 8086 to inform the end of control. Now 8086 regains the control of the bus.

$\overline{\text{LOCK}}$ - 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 **$\overline{\text{LOCK}}$** pin while executing an instruction prefixed by LOCK to prevent other bus masters from gaining control of the system bus.

QS_1, QS_0 - (**Q**ueue **S**tatus) - The processor provides the status of queue on these lines. The queue status can be used by the external device to track the internal status of the queue in the 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-2.6.

TABLE - 2.6 : 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

2.3 ARCHITECTURE OF INTEL 8086

The 8086 has a pipelined architecture. In pipelined architecture, the processor will have a number of functional units and the execution time of each functional unit overlaps. Each functional unit works independently most of the time. The simplified block diagram of the internal architecture of an 8086 is shown in Fig. 2.2. The architecture of the 8086 can be internally divided into two separate functional units : **B**us **I**nterface **U**nit (BIU) and **E**xecution **U**nit (EU).

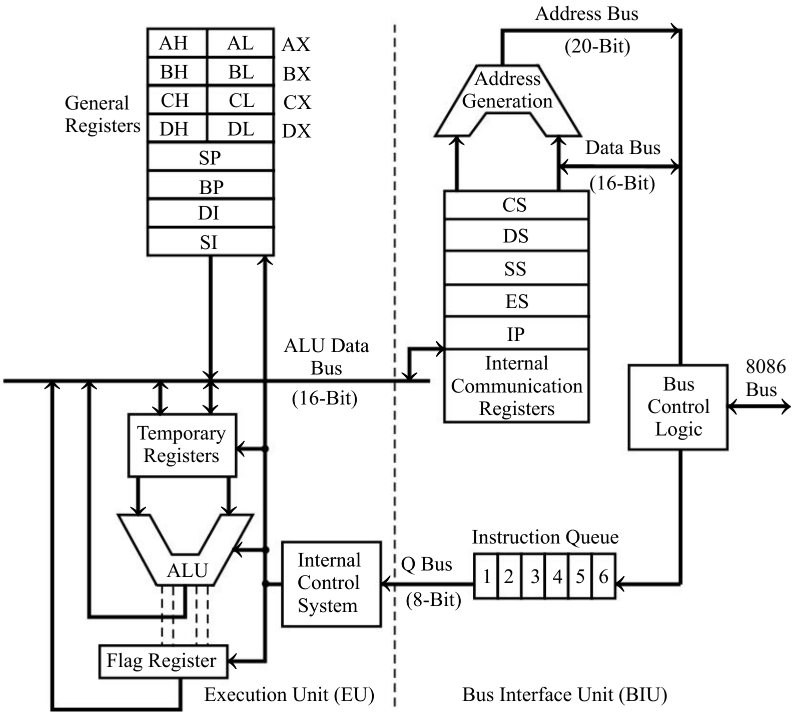


Fig. 2.2 : Internal architecture of 8086.

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 function independently.

The instruction queue is a FIFO (**F**irst-**I**n-**F**irst-**O**ut) group of registers. The size of queue is 6 bytes. The BIU fetches instruction code from the memory and stores it in the 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 (64 kb). 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 × 64 = 256 kb within 1 Mb memory space). This feature of 8086 allows the system designer to allocate separate areas 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 the segment registers. The memory segment can be continuous, partially overlapped, fully overlapped or disjoint.

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

The dedicated address generation unit generates the 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 the 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 the CS to the left four times and then adding it to the content of the IP (**I**nstruction **P**ointer). 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 the SS as base address and the content of the SP (**S**tack **P**ointer) as the 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 the instructions.

The EU has a 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.

TABLE - 2.7 : 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.

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

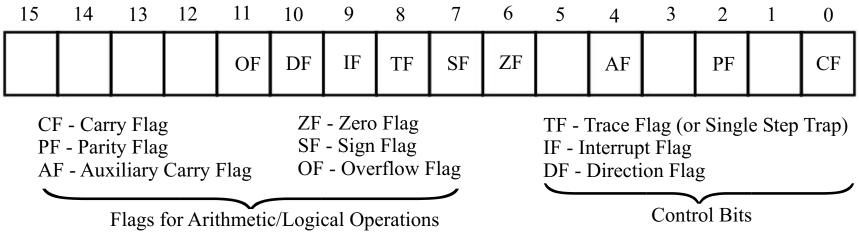
BX - can be used as BH and BL ; DX - can be used as DH and DL

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-2.7.

8086 Flag Register

The size of an 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 the arithmetic or logical operations. Three flags are used to control the processor operation and so they are also called control bits. The various flags of an 8086 processor and their bit position in the flag register are shown in Fig. 2.3.

Carry **F**lag (CF) is set if there is a carry from addition or borrow from subtraction. **A**uxiliary carry **F**lag (AF) is set if there is a carry from low nibble to high nibble of the low order 8-bit of a 16-bit number.



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				OF	DF	IF	TF	SF	ZF		AF		PF		CF

Fig. 2.3 : Bit positions of various flags in the flag register of 8086.

Overflow **F**lag (OF) is set to **one** if there is an arithmetic overflow, that is, if the size of the result exceeds the capacity of the destination location. **S**ign **F**lag (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. **P**arity **F**lag (PF) is set to **one** if the result has even parity and PF is cleared to **zero** for odd parity of the result. **Z**ero **F**lag (ZF) is set to **one** if the result is zero and ZF is cleared to **zero** for non zero result.

The three control bits in the flag register can be set or reset by the programmer. The **D**irection **F**lag (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 **I**nterrupt **F**lag (IF) to **one** causes the 8086 to recognize external maskable interrupts and clearing IF to **zero** disables the interrupts.

Setting **T**race **F**lag (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.

2.4 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 it 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, the $\overline{RD}$ is asserted **high** and at this instant the content of data bus is latched into two empty locations of instruction queue. Then BIU starts 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 are empty. When a branch instruction is encountered, the queue is emptied and then filled with program codes from the 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 requires 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/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 it to an effective or offset address specified by the instruction.

2.5 EVEN AND ODD MEMORY BANKS

The 8086 microprocessor uses a 20-bit address to access memory. With 20-bit address the processor can generate 2^{20} = 1 Mega address. The basic memory word size of the memories used in the 8086 system is 8-bit or 1-byte (i.e., in one memory location an 8-bit binary information can be stored). Hence, the physical memory space of the 8086 is 1Mb (1 Mega-byte).

For the programmer, the 8086 memory address space is a sequence of one mega-byte in which one location stores an 8-bit binary code/data and two consecutive locations store 16-bit binary code/data. But physically (i.e., in the hardware), the 1Mb memory space is divided into two banks of 512kb (512kb + 512kb = 1Mb). The two memory banks are called Even (or Lower) bank and Odd (or Upper) bank. The organization of even and odd memory banks in the 8086-based system is shown in Fig. 2.4.

The 8086-based system will have two sets of memory IC's. One set for even bank and another set for odd bank. The data lines D_0-D_7 are connected to even bank and the data lines D_8-D_{15} are connected to odd bank. The even memory bank is selected by the address line A_0 and the odd memory bank is selected by the control signal $\overline{BHE}$. The memory banks are selected when these signals are **low**(active low). Any memory location in the memory bank is selected by the address line A_1 to A_{19}.

The organization of memory into two banks and providing bank select signals allows the programmer to read/write the byte (8-bit) operand in any memory address through 16-bit data bus. It also allows the programmer to read/write the word (16-bit) operand starting from even address or odd address.

The memory access for byte and word operand from the even and odd bank by the 8086 processor will be as follows:

Case i **: Byte access from even bank**

For read/write operation of a byte in even memory address, A_0 is asserted **low** and $\overline{BHE}$ is asserted **high** (i.e., $A_0 = 0$ and $\overline{BHE} = 1$). Now the even bank alone is enabled and the data transfer take place through D_0-D_7 data lines.

Case ii **: Byte access from odd bank**

For read/write operation of a byte in odd memory address, A_0 is asserted **high** and $\overline{BHE}$ is asserted **low** (i.e., $A_0 = 1$ and $\overline{BHE} = 0$). Now odd bank alone is enabled and the data transfer take place through D_8-D_{15} data lines.

Case iii : Word access from even boundary

For read/write operation of a word (16-bit) in even boundary (i.e., low byte in even address and high byte in next address (odd address)), both A_0 and $\overline{BHE}$ are asserted **low** (i.e., $A_0 = 0$ and $\overline{BHE} = 0$). Now both the memory banks are enabled simultaneously and the processor read/write the 16-bit operand in one bus cycle through D_0-D_{15} data lines.

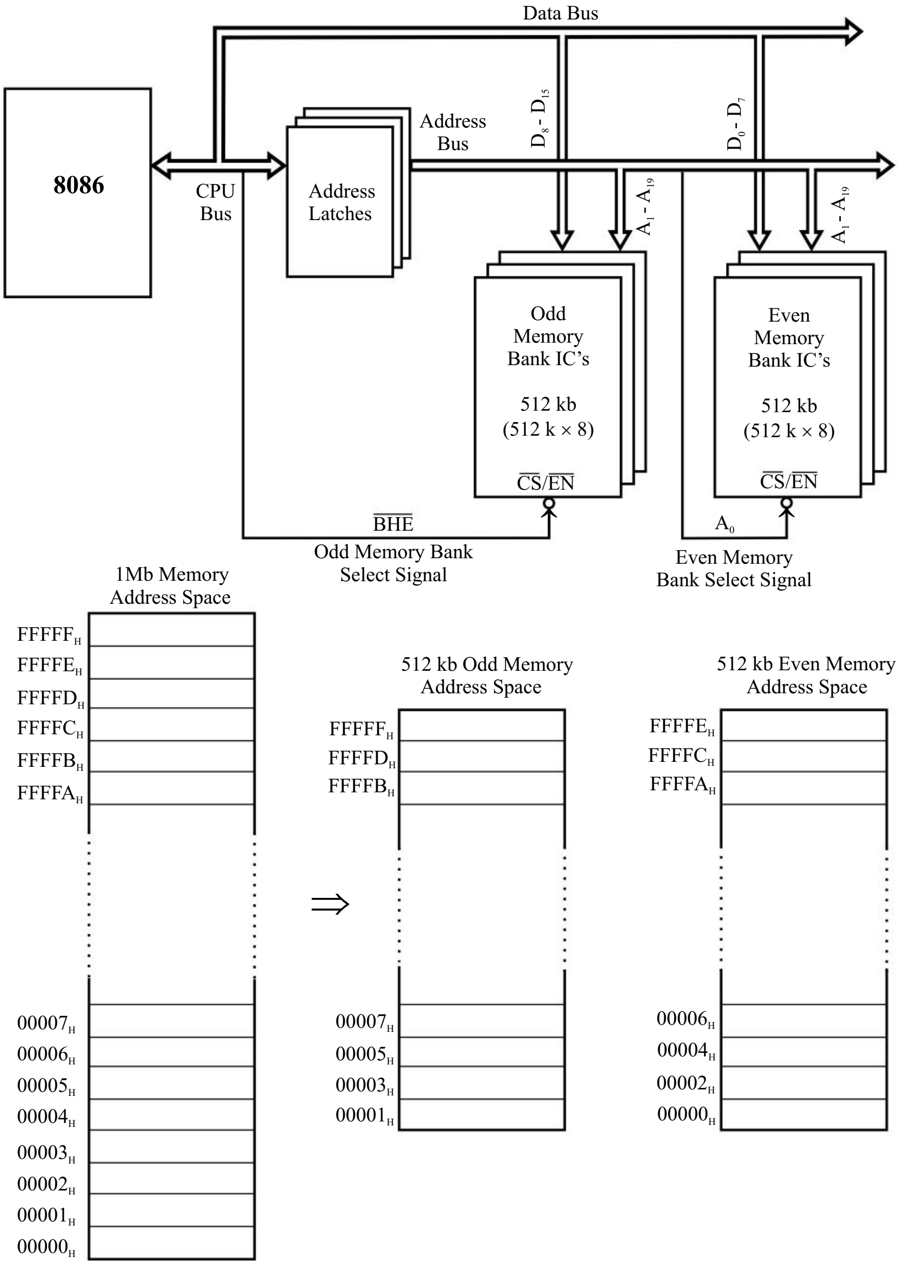


Fig. 2.4 : Organization of even and odd memory banks in 8086-based system.

Case iv **: Word access from odd boundary**

For read/write operation of a word (16-bit) in odd boundary (i.e., low byte in odd address and high byte in next address (even address)), the processor executes two bus cycles to read/write the word (16-bit) operand. In the first bus cycle A_0 is asserted **high** and $\overline{BHE}$ is asserted **low** (i.e., $A_0 = 1$ and $\overline{BHE} = 0$). Now the odd bank alone is enabled and the low byte of 16-bit operand is read/write through D_8-D_{15} data lines. In the second bus cycle A_0 is asserted **low** and $\overline{BHE}$ is asserted **high** (i.e., $A_0 = 0$ and $\overline{BHE} = 1$). Now the even bank alone is enabled and the high byte of 16-bit operand is read/write through D_0-D_7 data lines.

The status of A_0 and $\overline{BHE}$ for byte and word memory access are listed in Table-2.8.

TABLE - 2.8 : STATUS OF A_0 AND $\overline{BHE}$ DURING MEMORY ACCESS

Memory bank	Operand type	Status of		Data lines used for memory access	No. of bus cycle
		A_0	$\overline{BHE}$		
Even	Byte	0	1	D_0 - D_7	One
Odd	Byte	1	0	D_8 - D_{15}	One
Even	Word	0	0	D_0 - D_{15}	One
Odd	Word	1	0	D_8 - D_{15}	First cycle
		0	1	D_0 - D_7	Second cycle

Note : 1. *The processor may access the low byte operand via the upper data lines and high byte operand via the lower data lines, but while placing the operand in the registers it places in the appropriate locations.*

2. *When word operand is accessed from odd boundary the instruction execution will take extra time due to two bus cycle memory access.*

2.6 BUS CYCLES AND TIMING DIAGRAM

The 8086 processor has two functional units called **B**us **I**nterface **U**nit (BIU) and **E**xecution **U**nit (EU). Most of the time each unit works independently. The BIU takes care of fetching instruction codes from memory, and data from memory and IO devices. The EU takes care of executing the instructions prefetched by BIU.

The BIU initiates all external operations which are also called bus activity. The external bus activities are repetitions of certain basic operations. The basic operations performed by the CPU bus are called bus cycles. Depending on the activities of 8086, the bus cycles can be classified as follows:

1) Memory read cycle (Four T states)
2) Memory write cycle (Four T states)
3) IO read cycle (Four T states)
4) IO write cycle (Four T states)
5) Interrupt acknowledge cycle (Eight T states)

The processor takes a definite time to perform a bus cycle. The time taken to perform a bus cycle are specified in terms of T states. In an 8086 processor the time duration of one T-state is equal to one time period of the internal clock of the processor. The T-state starts in the middle of falling edge of the clock signal as shown in Fig. 2.5.

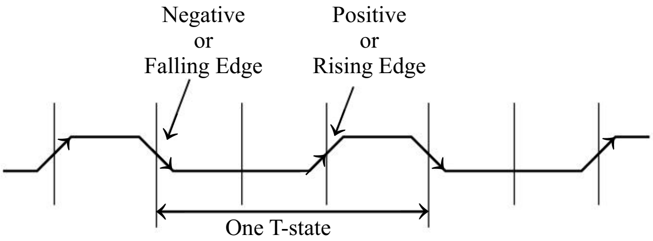


Fig. 2.5 : Clock signal and one T-state of 8086.

The normal time taken by 8086 to perform read/write cycle is four T states. The processor also has facility to extend the timing of bus cycles by introducing extra T states called wait states using READY control signal. If READY is permanently tied **high** then the bus cycles are executed in normal timing.

The memory or IO access time allowed by the 8086 processor with 5 MHz clock is 400 ns. If the memory or IO devices used in the system has access time more than 400 ns then wait states have to be introduced in the bus cycles, between the second and third T states (T_2 and T_3) to extend the timing of the bus cycle.

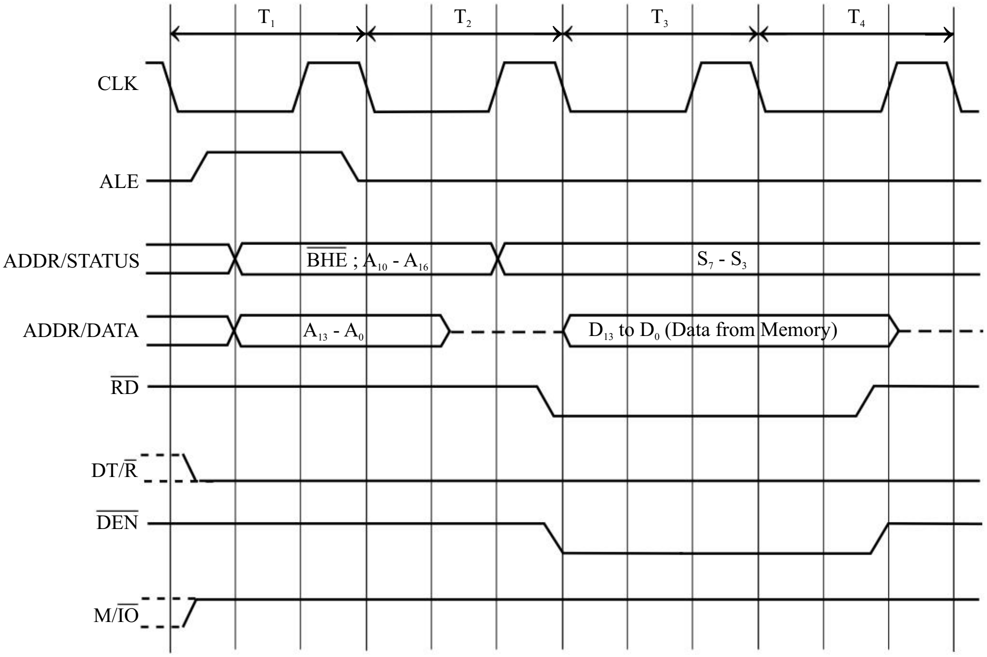
In order to introduce wait states the READY is made **low** by an external hardware in the beginning of second T-state and then made **high** after required time delay. The processor samples READY signal at the end of second T-state of every bus cycle. If READY is **low** at this time then the processor introduces one wait state. Again it samples READY signal at the end of wait state and if READY is still **low** then it introduces another wait state. This process is continued until READY is **high**. Once READY is made **high** the processor will resume the bus activity and completes the bus cycle.

Timing Diagram

The timing diagram provides information about the various conditions (**high** state or **low** state or **high impedance** state) of the signals while a bus 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 a microprocessor as the CPU.

Memory Read Cycle

The memory read cycle is initiated by BIU of 8086 to read a program code or data from memory. The normal time taken by memory read cycle is four clock periods. The timings of various signals involved in reading a word (16-bit) starting from even address of memory in minimum mode are shown in Fig. 2.6. The activities of the bus in each T-state are given below :



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

Fig. 2.6 : Memory read cycle of 8086.

Activities during T_1 :

i) The 8086 outputs a 20-bit memory address on AD_0-AD_{15} lines and ADDR/STATUS lines.

ii) The address latch enable signal ALE is asserted **high** in the beginning of T_1 and then asserted **low** at the end of T_1. This enables the external address latches to latch the address (at the falling edge of ALE) and keep on their output lines.

iii) The direction control signal DT/$\overline{R}$ is asserted **low** to inform the external bidirectional data buffer that the processor has to receive data. (If DT/$\overline{R}$ is already **low** in the previous bus cycle then it remains **low** as such.)

iv) The M/$\overline{IO}$ signal is asserted **high** to indicate memory access. (If M/$\overline{IO}$ is already **high** then it remains **high** as such.)

v) The $\overline{BHE}$ is asserted **low** to enable the odd/upper memory bank.

Activities during T_2 :

i) The AD_0-AD_{15} lines becomes inactive.

ii) The address is withdrawn from the ADDR/STATUS lines and status signals S_7-S_3 are issued on these lines. (The $\overline{BHE}$ becomes the status signal S_7.)

iii) At the end of T_2 the read control signal $\overline{RD}$ is asserted **low** to enable the output buffer of memory. The time during which $\overline{RD}$ remains **low** is the time allowed for memory to load data in the data bus.

iv) The $\overline{DEN}$ signal is asserted **low** to enable the external bidirectional data buffers.

v) The 8086 samples READY signal during T_2. (If READY is **high** then T_3 and T_4 are executed otherwise wait states are introduced.)

Activities during T_3 :

No activities are performed during T_3. The status of the signals at the end of T_2 are maintained throughout T_3.

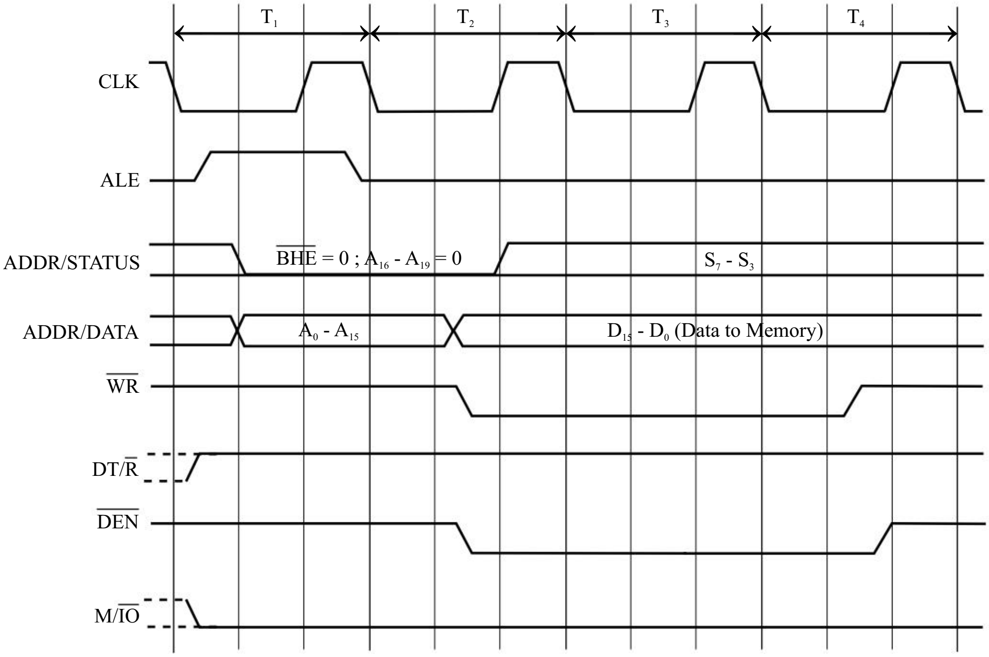
Activities during T_4 :

i) The $\overline{RD}$ is asserted **high** and at this time (i.e., at the rising edge of $\overline{RD}$) the data is latched into 8086.

ii) The $\overline{DEN}$ is made **high** to disable the data buffer.

Memory Write Cycle

The memory write cycle is initiated by BIU of 8086 to write a data in memory. The normal time taken by memory write cycle is four clock periods. The timings of various signals involved in writing a word (16-bit) starting from even address of memory in the minimum mode are shown in Fig. 2.7. The activities of the bus in each T-state are explained below:



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

Fig. 2.7 : Memory write cycle of 8086.

Activities during T_1 :

The activities during T_1 is same as that of read cycle except DT/$\overline{R}$ signal. In memory write cycle, the DT/$\overline{R}$ signal is asserted **high** to inform the external bidirectional data buffer that the processor is going to transmit data. (If DT/$\overline{R}$ is already **high** in the previous bus cycle then it remains, **high** as such.)

Activities during T_2 :

i) The address is withdrawn from AD_0-AD_{15} lines and data is output on these lines.

ii) The address is withdrawn from ADDR/STATUS lines and status signals are issued on these lines (The $\overline{BHE}$ becomes the status signal S_7.)

iii) When data is output on data bus, the control signals $\overline{WR}$ and $\overline{DEN}$ are also asserted **low** to enable the input buffer of memory and external data buffer on the data bus respectively.

iv) The 8086 samples READY signal during T_2. (If READY is **high** then T_3 and T_4 are executed otherwise wait states are introduced.)

Activities during T_3 :

No activities are performed during T_3.The status of the signals at the end of T_2 are maintained throughout T_3.

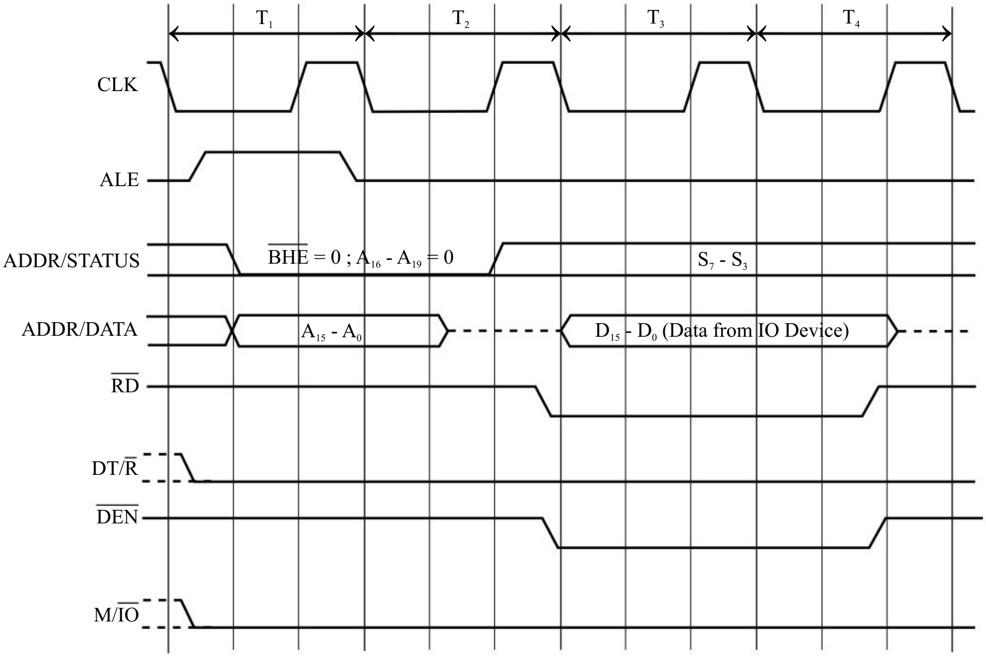
Activities during T_4 :

i) The $\overline{WR}$ is asserted **high** and at this time (i.e., at the rising edge of $\overline{WR}$), the data is latched into memory.

ii) The $\overline{DEN}$ is made **high** to disable the data buffers.

IO Read Cycle

The IO read cycle is initiated by BIU of 8086 to read a data from IO-mapped device or IO port. The normal time taken by IO read cycle is four clock periods. The timings of various signals involved in reading an IO port in the minimum mode are shown in Fig. 2.8. The activities of the bus in each T-state are given below:



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

Fig. 2.8 : IO read cycle of 8086.

Activities during T_1 :

i) The 8086 outputs a 16-bit IO address on AD_0-AD_{15} lines. Logic **low** is output on the $\overline{BHE}$ and ADDR/STATUS lines.

ii) The ALE is asserted **high** and then **low**. This enables the external address latches to latch the address (at the falling edge of ALE) and keep on their output lines.

iii) The DT/$\overline{R}$ signal is asserted **low** to inform the external bidirectional data buffer that the processor has to receive data. (If DT/$\overline{R}$ is already **low** in the previous bus cycle then it remains **low** as such.)

iv) The M/$\overline{IO}$ signal is asserted **low** to indicate IO access. (If M/$\overline{IO}$ is already **low** then it remains **low** as such.)

Activities during T_2 :

i) The AD_0-AD_{15} lines becomes inactive.

ii) The status signals S_7-S_3 are issued on ADDR/STATUS lines.

iii) At the end of T_2 the read control signal $\overline{RD}$ is asserted **low** to enable the IO device for read operation. The time during which $\overline{RD}$ remains **low** is the time allowed for IO device to load data in the data bus.

iv) The $\overline{DEN}$ signal is asserted **low** to enable the external bidirectional data buffers.

v) The 8086 samples READY signal during T_2. (If READY is **high** then T_3 and T_4 are executed otherwise wait states are introduced.)

Activities during T_3 :

No activities are performed during T_3. The status of the signals at the end of T_2 are maintained throughout T_3.

Activities during T_4 :

i) The $\overline{RD}$ is asserted **high** and at this time (i.e., at the rising edge of $\overline{RD}$), the data is latched into 8086.

ii) The $\overline{DEN}$ is made **high** to disable the data buffer.

IO Write Cycle

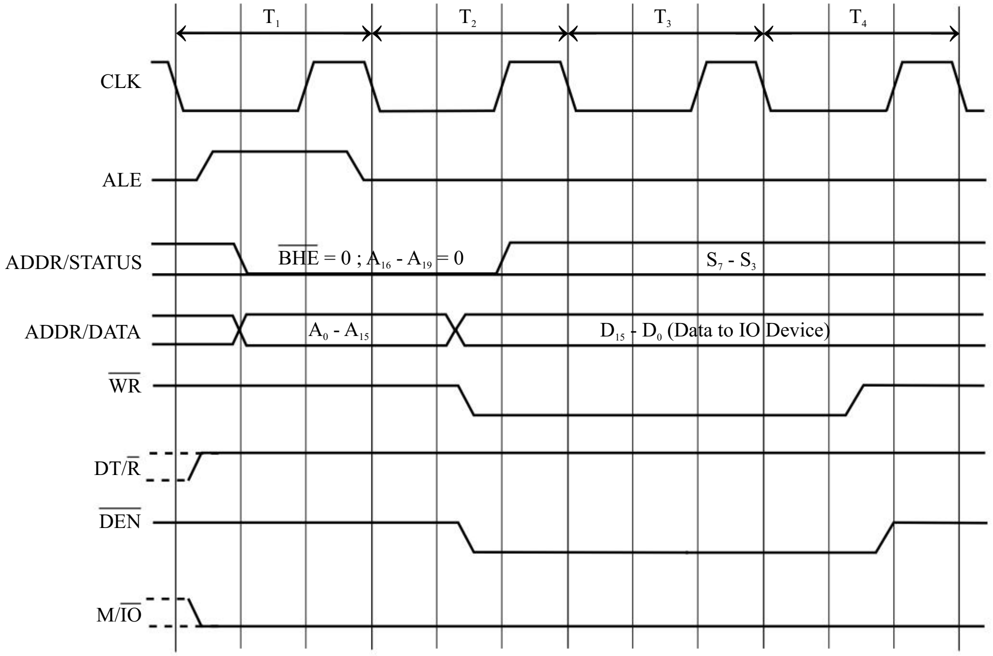
The IO write cycle is initiated by BIU of 8086 to send a data to IO device. The normal time taken by IO write cycle is four clock periods. The timings of various signals involved in sending a word to IO device in the minimum mode are shown in Fig. 2.9.

The activities during IO write cycle will be same as IO read cycle except the following.

i) During T_1, DT/$\overline{R}$ is asserted **high** to inform the external bidirectional data buffer that the processor is going to transmit data.

ii) During T_2, the address is withdrawn from AD_0-AD_{15} lines and data is output on these lines. At the same time $\overline{WR}$ is asserted **low** to enable the IO device for write operation and $\overline{DEN}$ is asserted **low** to enable the data buffer on the bus. Here $\overline{RD}$ remains **high**.

iii) During T_4, $\overline{WR}$ is asserted **high** and at this time (i.e., at the rising edge of $\overline{WR}$) the data is latched into IO device.



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

Fig. 2.9 : IO write cycle of 8086.

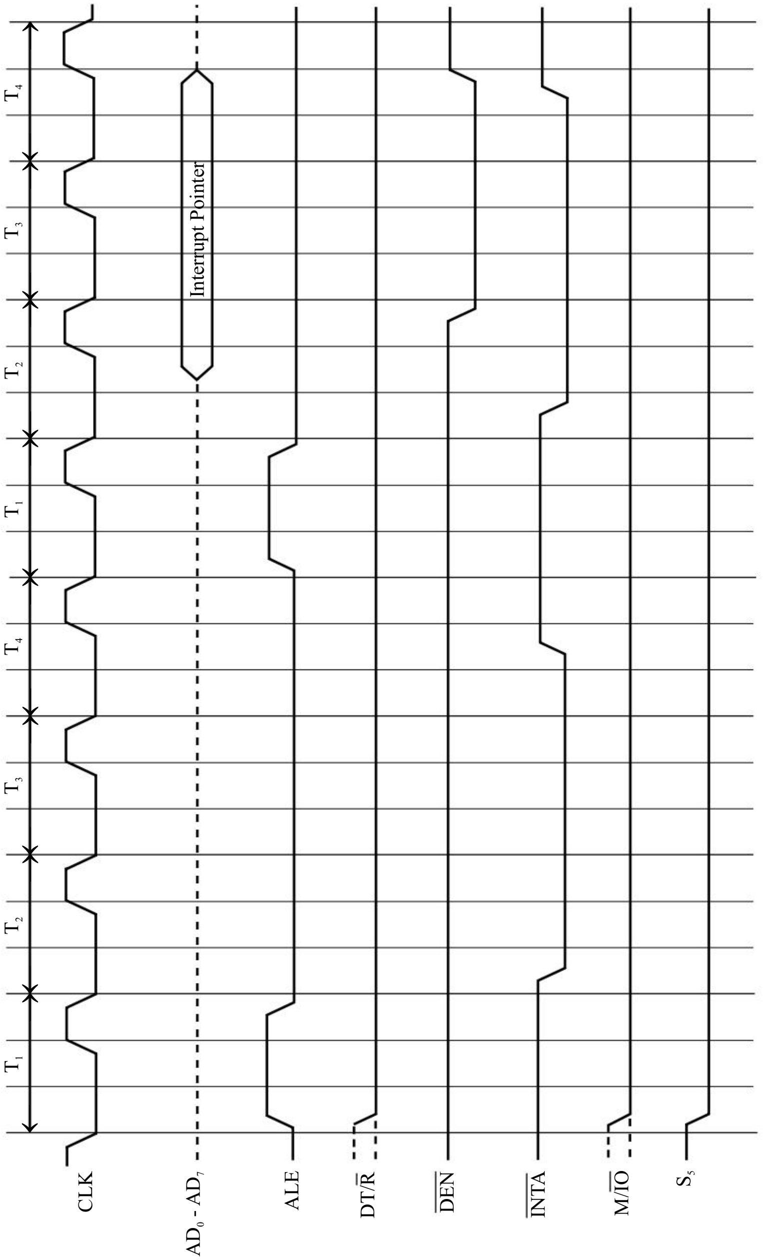
Interrupt Acknowledge Cycle

The interrupt acknowledge cycle is executed in response to an interrupt request through the INTR pin of 8086. The 8086 samples the status of INTR pin during the last T-state of an instruction (or at the end of instruction execution). If INTR is **high** at the time of sampling and Interrupt flag is enabled (i.e., IF = 1) then the processor saves (or pushes) the content of flag register, CS-register and IP in stack, and clears IF and TF flags, and then executes interrupt acknowledge cycle.

The time taken by 8086 to execute an interrupt acknowledge cycle is eight T states. It is actually two cycles with each cycle extending for 4T states. In the first cycle the processor send $\overline{\text{INTA}}$ to the interrupting device to inform the acceptance of interrupt. In the second cycle the processor requests the interrupting device to supply interrupt type number or pointer and read type number from the interrupting device by using $\overline{\text{INTA}}$ signal.

The timings of various signals during interrupt acknowlegde cycle in minimum mode are shown in Fig. 2.10. During T_1 of both the cycles ALE is made **high** and **low** which results in loading a junk value in address latches.

During T_1 of first cycle DT/$\overline{R}$, M/$\overline{IO}$ and S_5 are asserted **low**. The DT/$\overline{R}$ is asserted **low** to inform the data buffer that the processor has to receive data. The M/$\overline{IO}$ is asserted **low** to indicate IO operation. The S_5 is asserted **low** to inform the peripheral devices that the interrupt system is disabled. (Actually S_5 is the status of interrupt flag.)



($\overline{WR}$ and $\overline{RD}$ are **high.**)

Fig. 2.10 : Interrupt acknowledge cycle of 8086.

In both the cycles the $\overline{\text{INTA}}$ is asserted **low** during T_2 and then **high** during T_4. In the second cycle, when $\overline{\text{INTA}}$ is **low** the processor expects an 8-bit interrupt pointer on the lower eight lines (AD_0-AD_7) of the data bus. The time allowed to the interruping device to load the pointer is the time during which $\overline{\text{INTA}}$ remains **low**. The processor samples the interrupt pointer on the rising edge of $\overline{\text{INTA}}$ signal in the second cycle.

2.7 SHORT QUESTIONS AND ANSWERS

2.1 ***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.

2.2 ***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.

2.3 ***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.

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

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

2.5 ***How is a 20-bit physical address computed in 8086?***

In 8086, the 20-bit physical address is computed by multiplying a segment base address to 16_{10} and adding to a 16-bit effective address. For program codes, the CS-register will hold the segment base address and IP will hold the effective address. For data the DS/ES/SS-register will hold the segment base address and the method of calculating effective address will be specified in the instruction.

2.6 ***How is a 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 the clock generated by 8284 is thrice that of internal clock frequency of 8086. The 8284 divides the generated clock by three and modify the duty cycle to 33% and then supply as clock signal to 8086. The maximum internal clock frequency of 8086 is 5 MHz.

2.7 ***What is an ALE?***

ALE (**A**ddress **L**atch **E**nable) 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.

2.8 ***How is the READY signal used in a microprocessor system?***

The READY is an input signal that can be used by slow peripherals to get extra time in order to communicate with 8086. The 8086 will work only when READY is tied to logic **high**. Whenever READY is tied to logic **low**, the 8086 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 bus cycle, so that the processor will remain in wait state during this extra time.

2.9 What is HOLD and HLDA? How is it used?

The HOLD and HLDA signals are used for the **D**irect **M**emory **A**ccess (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 8086. 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 acknowledgement signal to DMA controller through HLDA pin. Upon receiving the acknowledgement 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.

2.10 What happens to the 8086 processor when it is reset?

When the processor is reset, the DS, SS, ES, IP and flag register are cleared, **C**ode **S**egment (CS) register is initialized to FFFF$_H$ and queue is emptied. After reset the processor will start fetching instruction from the 20-bit physical address FFFF0$_H$.

2.11 What is pipelined architecture?

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

2.12 What are the functional units available in 8086 architecture?

The **B**us **I**nterface **U**nit (BIU) and **E**xecution **U**nit (EU) are the two functional units available in 8086 architecture.

2.13. List the segment registers of 8086.

The segment registers of 8086 are **C**ode **S**egment (CS), **D**ata **S**egment (DS), **S**tack **S**egment (SS) and **E**xtra **S**egment (ES) registers.

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

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

2.15 What is a 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.

2.16 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.

2.17 Write the flags of 8086.

The 8086 has nine flags and they are:

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

2.18 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.

2.19 *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.

2.20 *How is memory organized in 8086?*

The 1MB physical memory space of 8086 is organized as two banks of 512 kb each. The two banks are known as odd (or upper) bank and even (or lower) bank. The odd bank is enabled using $\overline{\text{BHE}}$ and the even bank is enabled using the address line A_o.

2.21 *What is a bus cycle?*

A bus cycle is the basic external operation performed by the processor. It is also known as processor cycle or machine cycle. To execute an instruction, the processor will run one or more bus cycles in a particular order.

2.22. *List the bus cycles of 8086?*

The various bus cycles of 8086 are:

(i) Memory read cycle
(ii) Memory write cycle
(iii) IO read cycle
(iv) IO write cycle
(v) Interrupt acknowledge cycle

2.23. *What is T-state?*

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.

2.24 *What is the need for timing diagram?*

The timing diagram provides information regarding the status of various signals, when a bus 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.

2.25 *What operation is performed during the first T-state of every bus cycle in 8086?*

In 8086, during the first T-state of every bus cycle the address is latched into the external latches using ALE signal.

2.26 *When does the 8086 processor check for an interrupt?*

The 8086 checks for an interrupt in the last T-state of the last bus cycle of an instruction. (i.e., at the end of an instruction execution).

2.27. *What is interrupt acknowledge cycle?*

The interrupt acknowledge cycle is a bus cycle executed by 8086 processor after acceptance of an interrupt to get the interrupt type number or pointer, in-order to service the interrupting device.

2.28. *When does the READY signal sampled by the processor?*

The 8086 processor samples or checks the READY signal at the second T-state of every bus cycle.

2.29 *What are wait states?*

The T-state introduced between T_2 and T_3 of a bus cycle by the slow peripherals (to get extra time for read/write operation) are called wait states.

2.30 *When does the 8086 processor enter wait state?*

The 8086 processor will check for READY signal at the second T-state of a bus 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**.

CHAPTER 3

INSTRUCTION SET OF 8086

3.1 INTRODUCTION

The 8086 instructions can be classified into following six groups.

1. Data transfer instructions
2. Arithmetic instructions
3. Logical instructions
4. String manipulating instructions
5. Control transfer instructions
6. Processor control instructions

The data transfer group includes instructions for moving data between registers, register and memory, register and stack memory, and accumulator and IO device.

The arithmetic group includes instructions for addition and subtraction of binary, BCD and ASCII data, and instructions for multiplication and division of signed and unsigned binary data.

The logical group includes instructions for performing logical operations like AND, OR, Exclusive-OR, Complement, Shift, Rotate, etc. The string manipulation group includes instructions for moving string data between two memory locations and comparing string data word by word or byte by byte.

The control transfer group includes instructions to call a procedure/subroutine in the main program. It also includes instructions to jump from one part of a program to another part either conditionally (after checking flags) or unconditionally (without checking flags).

The processor control group includes instructions to set/clear the flags, to delay and halt the processor execution.

3.2 INSTRUCTIONS FORMAT

The size of 8086 instruction is one to six bytes. Some examples of 8086 instruction formats are shown in Fig. 3.1 and the general format of 8086 instruction is shown in Fig. 3.2. The exact format of individual instructions are summarized in Appendix-I : *Templates for 8086 instructions*.

In general, the first byte of the instruction will have a 6-bit opcode and two special bit indicators d-bit and w-bit or (s-bit and w-bit) or (v-bit and w-bit). Some instructions will have an 8-bit opcode and some instructions will have a 7-bit opcode followed by special bit indicator w-bit or z-bit. Some of the instructions will have a 2-bit or 3-bit register field in the first byte of the instruction. The usage of special one-bit indicators are given below:

w-bit : This bit appears in the format of instructions which can operate on both byte and word data.

If w = 0, then the data operated by the instruction is 8-bit/byte.

If w = 1, then the data operated by the instruction is 16-bit/word.

d-bit : This bit appears in the format of instructions which has a double operand. In double operand instructions, one of the operand should be a register specified by reg field. The d-bit is used to specify whether the register specified by reg field is source operand or destination operand.

If d = 0, then the register specified by reg field is source operand.

If d = 1, then the register specified by reg field is destination operand.

s-bit : This bit appears in the format of arithmetic instructions which operate on immediate data. If s = 1, w = 1 and immediate data is 8-bit then the immediate data is sign extended to 16-bit and used for arithmetic operation.

sw = 00 → 8-bit operation with an 8-bit immediate data.

sw = 01 → 16-bit operation with a 16-bit immediate data.

sw = 11 → 16-bit operation with a sign extended 8-bit immediate operand.

v-bit : This bit appears in the format of shift and rotate instructions.

If v = 0, then the shift/rotate operation is performed one time.

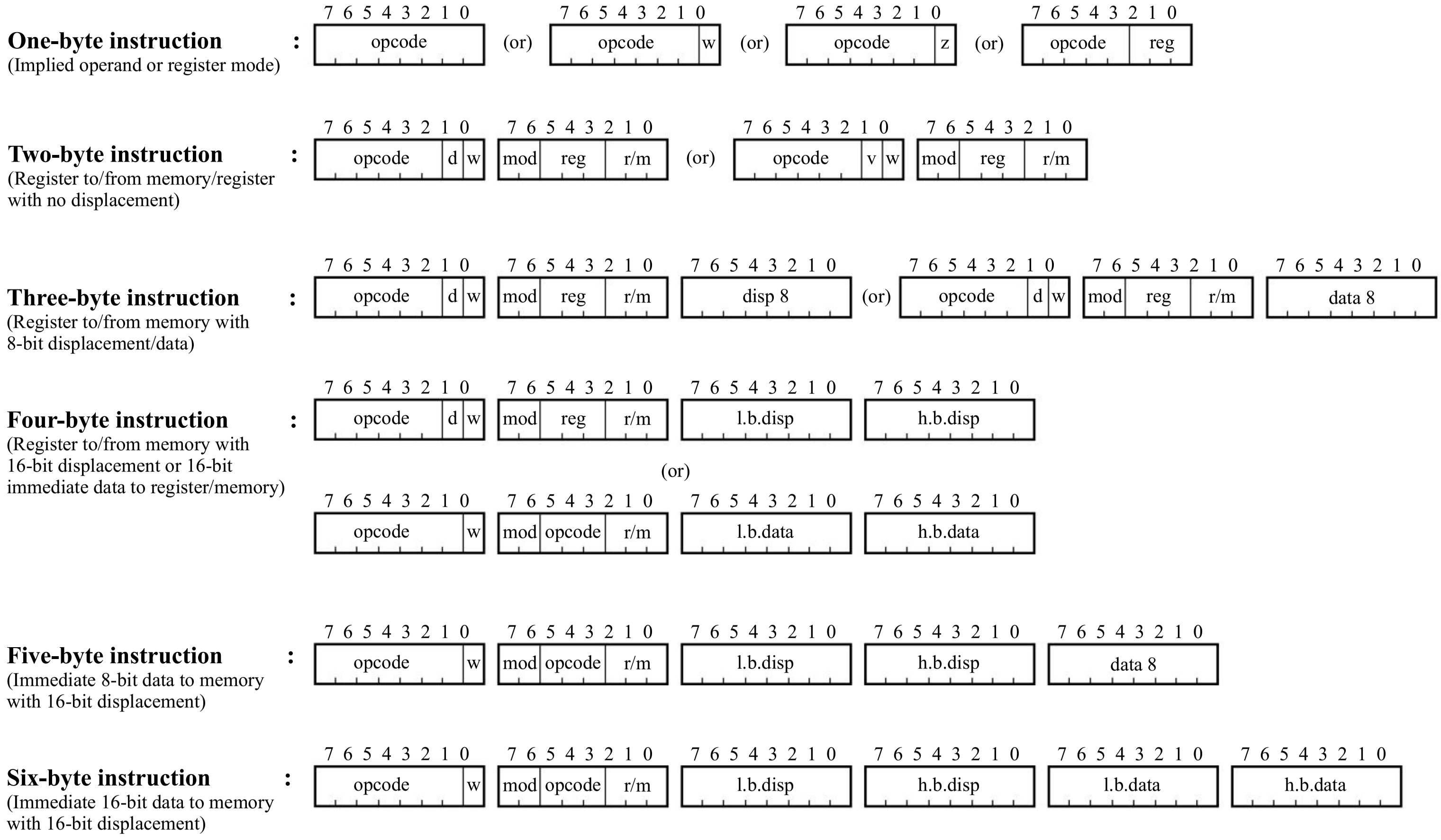
If v = 1, then the content of CL is count value for number of shift/rotate operations.

z-bit : This bit appears in the format of REP prefix for string instructions and is used for comparing with zero flag.

If z = 0, then repeat execution of string instruction until zero flag is zero.

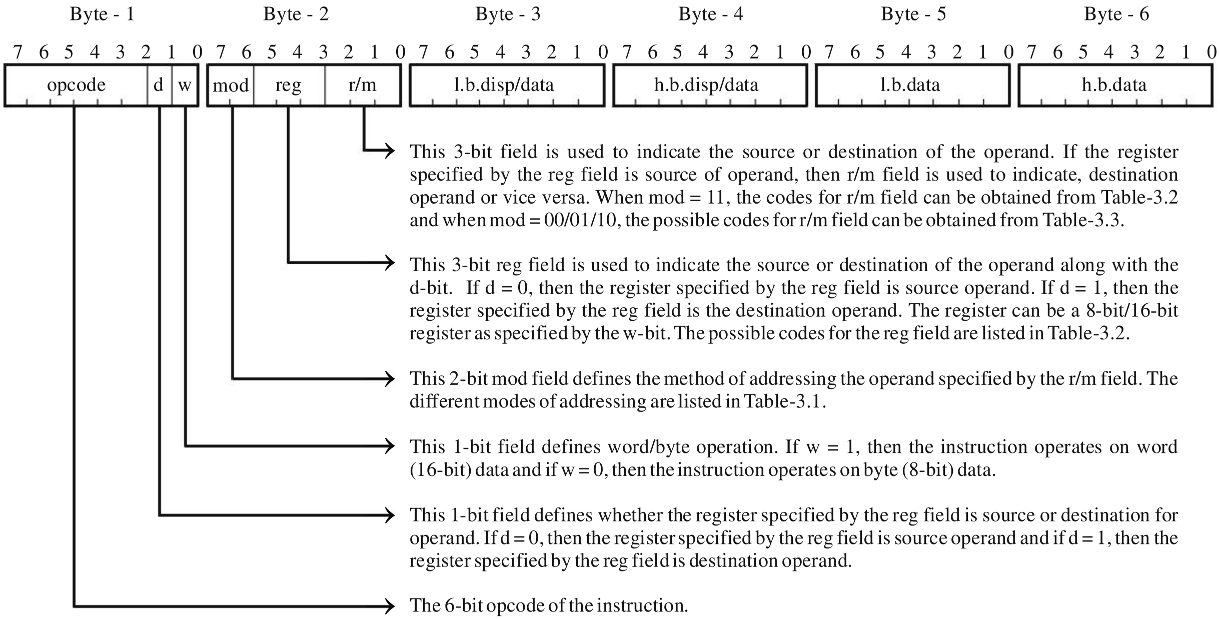
If z = 1, then repeat execution of string instruction until zero flag is one.

In multi-byte instructions, the second byte will specify the addressing mode of the operands. The second byte usually has three fields : mod, reg and r/m. The mod field is 2-bit wide and it defines the method of addressing the operand specified by r/m field. The r/m field is 3-bit wide and it is used to indicate the source or destination operand in memory/register. The reg field is 3-bit wide and it is used to indicate the source or destination operand in register. If register specified by reg field is source operand then r/m field is used to indicate destination operand or vice versa. The mod and r/m field are used to calculate the effective address of memory operand as shown in Table-3.3.



Note : *reg - register ; mod - mode ; r/m - register/memory ; l.b.disp - low byte of displacement ; h.b.disp - high byte of displacement ; l.b.data - low byte of data ; h.b.data - high byte of data ; data8 - 8-bit data.*

Fig. 3.1 : Examples of 8086 instruction formats.



This 3-bit field is used to indicate the source or destination of the operand. If the register specified by the reg field is source of operand, then r/m field is used to indicate, destination operand or vice versa. When mod = 11, the codes for r/m field can be obtained from Table-3.2 and when mod = 00/01/10, the possible codes for r/m field can be obtained from Table-3.3.

This 3-bit reg field is used to indicate the source or destination of the operand along with the d-bit. If d = 0, then the register specified by the reg field is source operand. If d = 1, then the register specified by the reg field is the destination operand. The register can be a 8-bit/16-bit register as specified by the w-bit. The possible codes for the reg field are listed in Table-3.2.

This 2-bit mod field defines the method of addressing the operand specified by the r/m field. The different modes of addressing are listed in Table-3.1.

This 1-bit field defines word/byte operation. If w = 1, then the instruction operates on word (16-bit) data and if w = 0, then the instruction operates on byte (8-bit) data.

This 1-bit field defines whether the register specified by the reg field is source or destination for operand. If d = 0, then the register specified by the reg field is source operand and if d = 1, then the register specified by the reg field is destination operand.

The 6-bit opcode of the instruction.

Note : *l.b.disp/data - low byte displacement or data ; h.b. disp/data - high byte displacement or data ; l.b. data - low byte data ; h.b. data - high byte data.*

Fig. 3.2 : General format of 8086 instruction.

In multi-byte instructions the bytes following the opcode and address mode bytes (1st and 2nd bytes) may be any one of the following:

i) no additional bytes.
ii) two-byte effective address.
iii) one-byte (8-bit) signed displacement or two-byte (16-bit) unsigned displacement.
iv) one-byte (8-bit) immediate data or two-byte (16-bit) immediate data (operand).
v) one/two byte displacement followed by one/two-byte immediate data (operand).
vi) two-byte effective address followed by two-byte segment address.

Note : If a displacement or immediate data is two bytes long, then the low order byte always appears first.

TABLE - 3.1 : CODES FOR mod FIELD

Code for mod field	Name of the mode
00	Memory mode with no displacement
01	Memory mode with 8-bit signed displacement
10	Memory mode with 16-bit unsigned displacement
11	Register mode

TABLE - 3.2 : CODES FOR reg FIELD

Code for reg field	Name of the register represented by the code when w = 0 or 1	
	When w = 0	When w = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

TABLE - 3.3 : CODES FOR r/m FIELD

Code for r/m field	Effective address calculation when mod = 00/01/10		
	mod = 00	mod = 01	mod = 10
000	[BX + SI]	[BX + SI + disp8]	[BX + SI + disp16]
001	[BX + DI]	[BX + DI + disp8]	[BX + DI + disp16]
010	[BP + SI]	[BP + SI + disp8]	[BP + SI + disp16]
011	[BP + DI]	[BP + DI + disp8]	[BP + DI + disp16]
100	[SI]	[SI + disp8]	[SI + disp16]
101	[DI]	[DI + disp8]	[DI + disp16]
110	[disp16]	[BP + disp8]	[BP + disp16]
111	[BX]	[BX + disp8]	[BX + disp16]

Note : disp 8 → 8-bit signed displacement.
disp16 → 16-bit unsigned displacement.

TABLE - 3.3 : MEMORY ADDRESS CALCULATION IN 8086 USING DEFAULT SEGMENT REGISTER

S.No.	Addressing mode	Effective address EA	Physical address MA/MA_S
1.	[BX + SI]	EA = (BX) + (SI)	MA = (DS) × 16_{10} + EA
2.	[BX + SI + disp8]	$disp8 \xrightarrow{\text{sign extend}} disp16$ EA = (BX) + (SI) + disp16	MA = (DS) × 16_{10} + EA
3.	[BX + SI + disp16]	EA = (BX) + (SI) + disp16	MA = (DS) × 16_{10} + EA
4.	[BX + DI]	EA = (BX) + (DI)	MA = (DS) × 16_{10} + EA
5.	[BX + DI + disp8]	$disp8 \xrightarrow{\text{sign extend}} disp16$ EA = (BX) + (DI) + disp16	MA = (DS) × 16_{10} + EA
6.	[BX + DI + disp16]	EA = (BX) + (DI) + disp16	MA = (DS) × 16_{10} + EA
7.	[BP + SI]	EA = (BP) + (SI)	MA_S = (SS) × 16_{10} + EA
8.	[BP + SI + disp8]	$disp8 \xrightarrow{\text{sign extend}} disp16$ EA = (BP) + (SI) + disp16	MA_S = (SS) × 16_{10} + EA
9.	[BP + SI + disp16]	EA = (BP) + (SI) + disp16	MA_S = (SS) × 16_{10} + EA
10.	[BP + DI]	EA = (BP) + (DI)	MA_S = (SS) × 16_{10} + EA
11.	[BP + DI + disp8]	$disp8 \xrightarrow{\text{sign extend}} disp16$ EA = (BP) + (DI) + disp16	MA_S = (SS) × 16_{10} + EA
12.	[BP + DI + disp16]	EA = (BP) + (DI) + disp16	MA_S = (SS) × 16_{10} + EA
13.	[SI]	EA = (SI)	MA = (DS) × 16_{10} + EA
14.	[SI + disp8]	$disp8 \xrightarrow{\text{sign extend}} disp16$ EA = (SI) + disp16	MA = (DS) × 16_{10} + EA
15.	[SI + disp16]	EA = (SI) + disp16	MA = (DS) × 16_{10} + EA
16.	[DI]	EA = (DI)	MA = (DS) × 16_{10} + EA
17.	[DI + disp8]	$disp8 \xrightarrow{\text{sign extend}} disp16$ EA = (DI) + disp16	MA = (DS) × 16_{10} + EA
18.	[DI + disp16]	EA = (DI) + disp16	MA = (DS) × 16_{10} + EA
19.	[disp16]	EA = disp16	MA = (DS) × 16_{10} + EA
20.	[BP + disp8]	$disp8 \xrightarrow{\text{sign extend}} disp16$ EA = (BP) + disp16	MA_S = (SS) × 16_{10} + EA
21.	[BP + disp16]	EA = (BP) + disp16	MA_S = (SS) × 16_{10} + EA
22.	[BX]	EA = (BX)	MA = (DS) × 16_{10} + EA
23.	[BX + disp8]	$disp8 \xrightarrow{\text{sign extend}} disp16$ EA = (BX) + disp16	MA = (DS) × 16_{10} + EA
24.	[BX + disp16]	EA = (BX) + disp16	MA = (DS) × 16_{10} + EA

Note : Segment registers used in address calculation can be modified using segment override prefix.
MA → Memory address of data segment ; MA_s → Memory address of stack segment.

3.3 ADDRESSING MODES OF 8086

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 8086 has 12 addressing modes and they can be classified into following five groups.

1. Register addressing	Group I	: Addressing modes for register and immediate data
2. Immediate addressing		
3. Direct addressing	Group II	: Addressing modes for memory data
3. Register indirect addressing		
5. Based addressing		
6. Indexed addressing		
7. Based index addressing		
8. String addressing		
9. Direct IO port addressing	Group III	: Addressing modes for IO ports
10. Indirect IO port addressing		
11. Relative addressing	Group IV	: Relative addressing mode
12. Implied addressing	Group V	: Implied addressing mode

Note : 1. *The "register" or "register + constant" enclosed by square brackets in the operand field of instructions refer to the method of effective address calculation of memory. The 16-bit constant enclosed by square brackets in the operand field of instructions refer to the effective address of memory data. The 8-bit/16-bit constants which are not enclosed by square brackets in the operand field refer to immediate data.*

2. *The term MA used in the symbolic description of instructions refer to physical memory address of data segment memory and MA_S refer to physical memory address of stack segment memory and MA_E refer to physical memory address of extra segment memory.*

3. *The register/memory enclosed by brackets in symbolic description refer to the content of register/memory.*

4. *For hexa-decimal constant (data/address) the letter H is included at the end of 8-bit/16-bit constants (data/address), and the numeral 0 is included in the front of hexadecimal constant starting with A through F.*

Register Addressing

In register addressing, the instruction will specify the name of the register which holds the data to be operated by the instruction.

Examples *:*

a) MOV CL,DH **(CL) ← (DH)**

The content of 8-bit register DH is moved to another 8-bit register CL.

b) MOV BX,DX **(BX) ← (DX)**

The content of 16-bit register DX is moved to another 16-bit register BX.

Immediate Addressing

In immediate addressing mode, an 8-bit or 16-bit data is specified as part of the instruction.

Examples :

a) MOV DL,08H **(DL) ← 08_H**

The 8-bit data (08_H) given in the instruction is moved to DL-register.

b) MOV AX,0A9FH **(AX) ← $0A9F_H$**

The 16-bit data ($0A9F_H$) given in the instruction is moved to AX-register.

Direct Addressing

*In direct addressing, an unsigned 16-bit displacement or signed 8-bit displacement will be specified in the instruction. The displacement is the **E**ffective **A**ddress(EA) or offset. In case of 8-bit displacement, the effective address is obtained by sign extending the 8-bit displacement to 16-bit.*

The 20-bit physical address of memory is calculated by multiplying the content of DS-register by 16_{10} (or 10_H) and adding to effective address. When segment override prefix is employed, the content of segment register specified in the override prefix will be used for segment base address calculation instead of DS-register.

Examples :

a) MOV DX,[08H]

EA = 0008_H (sign extended 8-bit displacement)

BA = (DS) ×16_{10} ; MA = BA + EA

(DX) ← (MA) or DL ← (MA)
DH ← (MA+1)

*The **E**ffective **A**ddress(EA) is obtained by sign extending the 8-bit displacement given in the instruction to 16-bit. The segment **B**ase **A**ddress(BA) is computed by multiplying the content of DS by 16_{10}. The **M**emory **A**ddress(MA) is computed by adding the **E**ffective **A**ddress (EA) to segment **B**ase **A**ddress(BA).*

The content of memory whose address is calculated as explained above is moved to DL-register and the content of next memory location is moved to DH-register.

b) MOV AX, [089DH]

EA = $089D_H$; BA = (DS) × 16_{10} ; MA = BA + EA

(AX) ← (MA) or (AL) ← (MA)
(AH) ← (MA+1)

*Here the 16-bit displacement given in the instruction is the effective address. The segment **B**ase **A**ddress(BA) is computed by multiplying the content of DS by 16_{10}. The **M**emory **A**ddress(MA) is computed by adding the **E**ffective **A**ddress(EA) to segment **B**ase **A**ddress(BA).*

The content of memory whose address is calculated as explained above is moved to AL-register and the content of next memory location is moved to AH-register.

Register Indirect Addressing

*In register indirect addressing, the name of the register which holds the **E**ffective **A**ddress(EA) will be specified in the instruction. The register used to hold the effective address are BX, SI and DI. The content of DS is used for segment base address calculation. When segment override prefix is employed, the content of segment register specified in the override prefix will be used for base address calculation instead of DS-register.*

The base address is obtained by multiplying the content of segment register by 16_{10}. The 20-bit physical address of memory is computed by adding the effective address to base address.

Examples :

a) MOV CX, [BX]

EA = (BX) ; BA = (DS) × 16_{10} ; MA = BA + EA

(CX) ← (MA) or (CL) ← (MA)
(CH) ← (MA+1)

*The content of BX is the **E**ffective **A**ddress(EA). The segment **B**ase **A**ddress(BA) is computed by multiplying the content of DS by 16_{10}. The **M**emory **A**ddress(MA) is obtained by adding BA and EA.*

The content of memory whose address is calculated as explained above is moved to CL-register and the content of next memory location is moved to CH-register.

b) MOV AX,[SI]

EA = (SI) ; BA = (DS) × 16_{10} ; MA = BA + EA

(AX) ← (MA) or (AL) ← (MA)

(AH) ← (MA + 1)

*The content of SI is the **E**ffective **A**ddress (EA). The segment **B**ase **A**ddress(BA) is computed by multiplying the content of DS by 16_{10}. The memory address is obtained by adding BA and EA.*

The content of memory whose address is calculated as explained above is moved toAL-register and the content of next memory location is moved to AH-register.

Based Addressing

*In this addressing mode, the BX or BP-register is used to hold a base value for effective address and a signed 8-bit or unsigned 16-bit displacement will be specified in the instruction. The displacement is added to base value in BX or BP to obtain the **E**ffective **A**ddress(EA). In case of 8-bit displacement, it is sign extended to 16-bit before adding to basevalue.*

When BX is used to hold base value for EA, the 20-bit physical address of memory is calculated by multiplying the content of DS by 16_{10} adding to EA.

When BP is used to hold base value for EA, the 20-bit physical address of memory is calculated by multiplying the content of SS by 16_{10} and adding to EA.

Example :

MOV AX, [BX+08H]

0008_H 08_H ; EA = (BX) + 0008_H

BA = (DS) × 16_{10} ; MA = BA + EA

(AX) ← (MA) or (AL) ← (MA)

(AH) ← (MA+1)

*The effective address is calculated by sign extending the 8-bit displacement given in the instruction to 16-bit and adding to the content of BX-register. The **B**ase **A**ddress(BA) is obtained by multiplying the content of DS by 16_{10}. The **M**emory **A**ddress(MA) is obtained by adding BA and EA.*

The content of memory whose address is calculated as explained above is moved to AL-register and the content of next memory is moved to AH-register.

Indexed Addressing

*In this addressing mode, the SI or DI-register is used to hold an index value for memory data and a signed 8-bit displacement or unsigned 16-bit displacement will be specified in the instruction. The displacement is added to index value in SI or DI-register to obtain the **E**ffective **A**ddress(EA). In case of 8-bit displacement it is sign extended to 16-bit before adding to index value.*

*The 20-bit memory address is calculated by multiplying the content of **D**ata **S**egment(DS) by 16_{10} and adding to EA.*

Note : In general the effective address = Reference + modifier.
In this context, the based and indexed addressing looks similar, but in based addressing, base value is the reference and displacement is the modifier, whereas in indexed addressing, displacement is the reference and index value is the modifier.

Example :

MOV CX, [SI+0A2H]

$\mathbf{FFA2_H \xleftarrow{\text{sign extend}} A2_H}$; $\mathbf{EA = (SI) + FFA2_H}$

$\mathbf{BA = (DS) \times 16_{10}}$; $\mathbf{MA = BA + EA}$

$\mathbf{(CX) \leftarrow (MA)}$ **or** $\mathbf{(CL) \leftarrow (MA)}$

$\mathbf{(CH) \leftarrow (MA + 1)}$

*The effective address is calculated by sign extending the 8-bit displacement given in the instruction to 16-bit and adding to the content of SI-register. The **B**ase **A**ddress (BA) is obtained by multiplying the content of DS by* 16_{10}*. The **M**emory **A**ddress(MA) is obtained by adding BA and EA.*

The content of memory whose address is calculated as explained above is moved to CL-register and the content of next memory is moved to CH-register.

Based Indexed Addressing

In this addressing mode, the effective address is given by sum of base value, index value and an 8-bit or 16-bit displacement specified in the instruction. The base value is stored in BX or BP-register. The index value is stored in SI or DI-register. In case of 8-bit displacement, it is sign extended to 16-bit before adding to base value. This type of addressing will be useful in addressing two dimensional arrays where we require two modifiers.

When BX is used to hold base value for EA, the 20-bit physical address of memory is calculated by multiplying the content of DS by 16_{10} *and adding to EA.*

When BP is used to hold the base value for EA, the 20-bit physical address of memory is obtained by multiplying the content of SS-register by 16_{10} *and adding it to EA.*

Example :

MOV DX, [BX+SI+0AH]

$\mathbf{000A_H \xleftarrow{\text{sign extend}} 0A_H}$; $\mathbf{EA = (BX) + (SI) + 000A_H}$

$\mathbf{BA = (DS) \times 16_{10}}$; $\mathbf{MA = BA + EA}$

$\mathbf{(DX) \leftarrow (MA)}$ **or** $\mathbf{(DL) \leftarrow (MA)}$

$\mathbf{(DH) \leftarrow (MA + 1)}$

*The **E**ffective **A**ddress(EA) is calculated by sign extending the 8-bit displacement given in the instruction to 16-bit and adding it to the content of BX and SI-register. The **B**ase **A**ddress(BA) is obtained by multiplying the content of DS by* 16_{10}*. The 20-bit **M**emory **A**ddress(MA) is obtained by adding BA and EA.*

The content of memory whose address is calculated as explained above is moved to DL-register and the content of the next memory location is moved to DH-register.

STRING ADDRESSING

*This addressing mode is employed in string instructions to operate on string data. In string addressing mode, the **E**ffective **A**ddress(EA) of source data is stored in SI-register and the EA of destination data is stored in DI-register.*

The segment register used for calculating base address for source data is DS and can be overridden. The segment register used for calculating base address for destination is ES and cannot be overridden.

*This addressing mode also supports auto increment/decrement of index registers SI and DI depending on **D**irection **F**lag(DF). If DF=1, then the content of index registers are decremented to point to next byte/word of the string after execution of a string instruction. If DF=0, then the content of index registers are incremented to point to previous byte/word of the string after execution of a string instruction. (For word operand, the content of index registers are incremented/decremented by two and for byte operand, the content of index registers are incremented/decremented by one.)*

Example :

MOVS BYTE

$EA = (SI)$; $BA = (DS) \times 16_{10}$; $MA = BA + EA$

$EA_E = (DI)$; $BA_E = (ES) \times 16_{10}$; $MA_E = BA_E + EA_E$

$(MA_E) \leftarrow (MA)$

If $DF = 1$, then $(SI) \leftarrow (SI) - 1$ and $(DI) \leftarrow (DI) - 1$

If $DF = 0$, then $(SI) \leftarrow (SI) + 1$ and $(DI) \leftarrow (DI) + 1$

This instruction move a byte of string data from one memory location to another memory location. The address of source memory location is calculated by multiplying the content of DS by 16_{10} and adding to SI. The address of destination memory location is calculated by multiplying the content of ES by 16_{10} and adding to DI.

After the move operation if DF = 1, then the content of index registers DI and SI are decremented by one. If DF = 0, then the content of index registers DI and SI are incremented by one.

Direct IO Port Addressing

This addressing mode is used to access data from standard IO-mapped devices or ports. In the direct port addressing mode, an 8-bit port address is directly specified in the instruction.

Example :

IN AL, [09H]

$PORT_{addr} = 09_H$

$(AL) \leftarrow (PORT)$

The content of the port with address 09_H is moved to AL-register.

Indirect IO Port Addressing

This addressing mode is used to access data from standard IO-mapped devices or ports. In the indirect port addressing mode, the instruction will specify the name of the register which holds the port address. In 8086, the 16-bit port address is stored in DX-register.

Example :

OUT [DX], AX

$PORT_{addr} = (DX)$; $(PORT) \leftarrow (AX)$

The content of AX is moved to the port whose address is specified by DX-register.

Relative Addressing

In this addressing mode the effective address of a program instruction is specified relative to the **I**nstruction **P**ointer (IP) by an 8-bit signed displacement.

Example :

JZ 0AH

$000A_H \xleftarrow{\text{sign extend}} 0A_H$

If $ZF = 1$, then $(IP) \leftarrow (IP) + 000A_H$; $EA_C = (IP) + 000A_H$

$BA_C = (CS) \times 16_{10}$; $MA_C = BA_C + EA_C$

Note : Suffix C refers to code memory

If ZF = 1, then the program control jumps to a new code address as calculated above. If ZF = 0, then the next instruction of the program is executed.

Implied Addressing

In implied addressing mode, the instruction itself will specify the data to be operated by the instruction.

Example :

CLC - Clear carry ; CF ← 0

Execution of this instruction will clear the **C**arry **F**lag(CF).

3.4 INSTRUCTION EXECUTION TIME

The execution time of each instruction of 8086 is specified by INTEL in terms of clock cycles or periods of the processor. The execution time can be determined by multiplying the number of clock cycles needed to execute the instruction by the time period of the processor clock. The execution time of 8086 instructions are listed in the Table-3.5 : *INTEL 8086 instruction set*. The execution time specified in Table-3.5, assumes that the instruction to be executed has already been fetched and stored in the instruction queue.

When the instruction execution involves memory access then extra time is needed for memory address calculation. This extra time is denoted as EACT (**E**ffective **A**ddress **C**alculation **T**ime) in Table-3.5. The time required for address calculation (EACT) depends on addressing mode used in the instruction. The clock cycles required for address calculation for various memory addressing modes are listed in Table-3.6.

Instructions like multiply, divide, shift and rotate will have variable execution time depending on the type of data operated by the instruction.

The conditional branch instructions will have different timings for TRUE and FALSE condition. When the condition is TRUE, branch operation takes place which involves modifying the IP and CS, clearing the queue and then filling the queue with instruction codes from branch address, and so instruction execution takes a longer time. When the condition is FALSE, the next instruction is executed and so the instruction takes lesser time.

The execution time specified in Table-3.5 also assumes that word operand is located in an even address. If the word operand is located in an odd address then the processor executes two bus cycles for each memory access and so four extra clock cycle has to be added to execution time of the instruction for each memory access. The number of memory access while executing the instruction is also listed in Table-3.5.

TABLE - 3.5 : INTEL 8086 INSTRUCTION SET

S.No.	Mnemonic	Size of instruction (Number of bytes)	Clock period (or T-state) needed for instruction execution	Number of memory access (or transfer)
GROUP I : DATA TRANSFER INSTRUCTIONS				
1.	MOV reg2/mem, reg1/mem			
a)	MOV reg2, reg1	2	2	-
b)	MOV mem, reg1	2 to 4	9 + EACT	1
c)	MOV reg2, mem	2 to 4	8 + EACT	1
2.	MOV reg/mem, data			
a)	MOV reg, data	3 to 4	4	-
b)	MOV mem, data	3 to 6	10 + EACT	1
3.	MOV reg, data	2 to 3	4	-
4.	MOV A, mem			
a)	MOV AL, mem	3	10	1
b)	MOV AX, mem	3	10	1

Table - 3.5 continued...

S.No.	Mnemonic	Size of instruction (Number of bytes)	Clock period (or T-state) needed for instruction execution	Number of memory access (or transfer)
5.	MOV mem, A			
a)	MOV mem, AL	3	10	1
b)	MOV mem, AX	3	10	1
6.	MOV segreg, reg16/mem			
a)	MOV segreg, reg16	2	2	-
b)	MOV segreg, mem	2 to 4	8 + EACT	1
7.	MOV reg16/mem, segreg			
a)	MOV reg16, segreg	2	2	-
b)	MOV mem, segreg	2 to 4	9 + EACT	1
8.	PUSH reg16/mem			
a)	PUSH reg16	2	11	1
b)	PUSH mem	2 to 4	16 + EACT	2
9.	PUSH reg16	1	11	1
10.	PUSH segreg	1	10	1
11.	PUSHF	1	10	1
12.	POP reg16/mem			
a)	POP reg16	2	8	1
b)	POP mem	2 to 4	17 + EACT	2
13.	POP reg16	1	8	1
14.	POP segreg	1	8	1
15.	POPF	1	8	1
16.	XCHG reg2/mem, reg1			
a)	XCHG reg2, reg1	2	4	-
b)	XCHG mem, reg1	2 to 4	17 + EACT	2
17.	XCHG AX, reg16	1	3	-
18.	XLAT	1	11	1
19.	IN A, [DX]			
a)	IN AL, [DX]	1	8	1 } IO
b)	IN AX, [DX]	1	8	1 } access

Table - 3.5 continued...

S.No.	Mnemonic	Size of instruction (Number of bytes)	Clock period (or T-state) needed for instruction execution	Number of memory access (or transfer)
20.	IN A, addr8			
a)	IN AL, addr8	2	10	1 } IO
b)	IN AX, addr8	2	10	1 } access
21.	OUT [DX], A			
a)	OUT [DX], AL	1	8	1 } IO
b)	OUT [DX], AX	1	8	1 } access
22.	OUT addr8, A			
a)	OUT addr8, AL	2	10	1 } IO
b)	OUT addr8, AX	2	10	1 } access
23.	LEA reg16, mem	2 to 4	2 + EACT	-
24.	LDS reg16, mem	2 to 4	16 + EACT	2
25.	LES reg16, mem	2 to 4	16 + EACT	2
26.	LAHF	1	4	-
27.	SAHF	1	4	-
GROUP II : ARITHMETIC INSTRUCTIONS				
28.	ADD reg2/mem, reg1/mem			
a)	ADD reg2, reg1	2	3	-
b)	ADD reg2, mem	2 to 4	9 + EACT	1
c)	ADD mem, reg1	2 to 4	16 + EACT	2
29.	ADD reg/mem, data			
a)	ADD reg, data	3 to 4	4	-
b)	ADD mem, data	3 to 6	17 + EACT	2
30.	ADD A, data			
a)	ADD AL, data8	2	4	-
b)	ADD AX, data16	3	4	-
31.	ADC reg2/mem, reg1/mem			
a)	ADC reg2, reg1	2	3	-
b)	ADC reg2, mem	2 to 4	9 + EACT	1
c)	ADC mem, reg1	2 to 4	16 + EACT	2

Table - 3.5 continued...

S.No.	Mnemonic	Size of instruction (Number of bytes)	Clock period (or T-state) needed for instruction execution	Number of memory access (or transfer)
32.	ADC reg/mem, data			
a)	ADC reg, data	3 to 4	4	-
b)	ADC mem, data	3 to 6	17 + EACT	2
33.	ADC A, data			
a)	ADC AL, data8	2	4	-
b)	ADC AX, data16	3	4	-
34.	AAA	1	4	-
35.	DAA	1	4	-
36.	SUB reg2/mem, reg1/mem			
a)	SUB reg2, reg1	2	3	-
b)	SUB reg2, mem	2 to 4	9 + EACT	1
c)	SUB mem, reg1	2 to 4	16 + EACT	2
37.	SUB reg/mem, data			
a)	SUB reg, data	3 to 4	4	-
b)	SUB mem, data	3 to 6	17 + EACT	2
38.	SUB A, data			
a)	SUB AL, data8	2	4	-
b)	SUB AX, data16	3	4	-
39.	SBB reg2/mem, reg1/mem			
a)	SBB reg2, reg1	2	3	-
b)	SBB reg2, mem	2 to 4	9 + EACT	1
c)	SBB mem, reg1	2 to 4	16 + EACT	2
40.	SBB reg/mem, data			
a)	SBB reg, data	3 to 4	4	-
b)	SBB mem, data	3 to 6	17 + EACT	2
41.	SBB A, data			
a)	SBB AL, data8	2	4	-
b)	SBB AX, data16	3	4	-
42.	AAS	1	4	-
43.	DAS	1	4	-

Table - 3.5 continued...

S.No.	Mnemonic	Size of instruction (Number of bytes)	Clock period (or T-state) needed for instruction execution	Number of memory access (or transfer)
44.	MUL reg/mem			
a)	MUL reg			
	i) MUL reg8	2	70 to 77	-
	ii) MUL reg16	2	118 to 133	-
b)	MUL mem			
	i) MUL mem8	2 to 4	(76 to 83) + EACT	1
	ii) MUL mem16	2 to 4	(124 to 139)+EACT	1
45.	IMUL reg/mem			
a)	IMUL reg			
	i) IMUL reg8	2	80 to 98	-
	ii) IMUL reg16	2	128 to 154	-
b)	IMUL mem			
	i) IMUL mem8	2 to 4	(86 to 104) + EACT	1
	ii) IMUL mem16	2 to 4	(134 to 160)+EACT	1
46.	AAM	2	83	-
47.	DIV reg/mem			
a)	DIV reg			
	i) DIV reg8	2	80 to 90	-
	ii) DIV reg16	2	144 to162	-
b)	DIV mem			
	i) DIV mem8	2 to 4	(86 to 96) + EACT	1
	ii) DIV mem16	2 to 4	(150 to 168)+EACT	1
48.	IDIV reg/mem			
a)	IDIV reg			
	i) IDIV reg8	2	101 to 112	-
	ii) IDIV reg16	2	165 to 184	-
b)	IDIV mem			
	i) IDIV mem8	2 to 4	(107 to 118)+EACT	1
	ii) IDIV mem16	2 to 4	(171 to 190)+EACT	1
49.	AAD	2	60	-
50.	NEG mem/reg			
a)	NEG reg	2	3	1
b)	NEG mem	2 to 4	16 + EACT	2
51.	INC reg8/mem			
a)	INC reg8	2	3	-
b)	INC mem	2 to 4	15 + EACT	2

Table - 3.5 continued...

S.No.	Mnemonic	Size of instruction (Number of bytes)	Clock period (or T-state) needed for instruction execution	Number of memory access (or transfer)
52.	INC reg16	1	2	-
53.	DEC reg8/mem			
a)	DEC reg8	2	3	-
b)	DEC mem	2 to 4	15 + EACT	2
54.	DEC reg16	1	2	-
55.	CBW	1	2	-
56.	CWD	1	5	-
57.	CMP reg2/mem, reg1/mem			
a)	CMP reg2, reg1	2	3	-
b)	CMP reg2, mem	2 to 4	9 + EACT	1
c)	CMP mem, reg1	2 to 4	9 + EACT	1
58.	CMP reg/mem, data			
a)	CMP reg, data	3 to 4	4	-
b)	CMP mem, data	3 to 6	10 + EACT	1
59.	CMP A, data			
a)	CMP AL, data8	2	4	-
b)	CMP AX, data16	3	4	-
GROUP III : LOGICAL INSTRUCTIONS				
60.	AND reg2/mem, reg1/mem			
a)	AND reg2, reg1	2	3	-
b)	AND reg2, mem	2 to 4	9 + EACT	1
c)	AND mem, reg1	2 to 4	16 + EACT	2
61.	AND reg/mem, data			
a)	AND reg, data	3 to 4	4	-
b)	AND mem, data	3 to 6	17 + EACT	2
62.	AND A, data			
a)	AND AL, data8	2	4	-
b)	AND AX, data16	3	4	-

Table - 3.5 continued...

S.No.	Mnemonic	Size of instruction (Number of bytes)	Clock period (or T-state) needed for instruction execution	Number of memory access (or transfer)
63.	OR reg2/mem, reg1/mem			
a)	OR reg2, reg1	2	3	-
b)	OR reg2, mem	2 to 4	9 + EACT	1
c)	OR mem, reg1	2 to 4	16 + EACT	
64.	OR reg/mem, data			
a)	OR reg, data	3 to 4	4	-
b)	OR mem, data	3 to 6	17 + EACT	2
65.	OR A, data			
a)	OR AL, data8	2	4	-
b)	OR AX, data16	3	4	-
66.	XOR reg2/mem, reg1/mem			
a)	XOR reg2, reg1	2	3	-
b)	XOR reg2, mem	2 to 4	9 + EACT	1
c)	XOR mem, reg1	2 to 4	16 + EACT	2
67.	XOR reg/mem, data			
a)	XOR reg, data	3 to 4	4	-
b)	XOR mem, data	3 to 6	17 + EACT	2
68.	XOR A, data			
a)	XOR AL, data8	2	4	-
b)	XOR AX, data16	3	4	-
69.	TEST reg2/mem, reg1/mem			
a)	TEST reg2, reg1	2	3	-
b)	TEST reg2, mem	2 to 4	9 + EACT	1
c)	TEST mem, reg1	2 to 4	9 + EACT	1
70.	TEST reg/mem, data			
a)	TEST reg, data	3 to 4	5	-
b)	TEST mem, data	3 to 6	11 + EACT	1
71.	TEST A, data			
a)	TEST AL, data8	2	4	-
b)	TEST AX, data16	3	4	-
72.	NOT reg/mem			
a)	NOT reg	2	3	-
b)	NOT mem	2 to 4	16 + EACT	2

Table - 3.5 continued...

S.No.	Mnemonic	Size of instruction (Number of bytes)	Clock period (or T-state) needed for instruction execution	Number of memory access (or transfer)
73.	SHL reg/mem			
	(or SAL reg/mem)			
a)	SHL reg (or SAL reg)			
	i) SHL reg, 1	2	2	-
	(or SAL reg, 1)			
	ii) SHL reg, CL	2	8 + 4 per bit	-
	(or SAL reg, CL)			
b)	SHL mem (or SAL mem)			
	i) SHL mem, 1	2 to 4	15 + EACT	2
	(or SAL mem, 1)			
	ii) SHL mem, CL	2 to 4	20+EACT+4 per bit	2
	(or SAL mem, CL)			
74.	SHR reg/mem			
a)	SHR reg			
	i) SHR reg, 1	2	2	-
	ii) SHR reg, CL	2	8 + 4 per bit	-
b)	SHR mem			
	i) SHR mem, 1	2 to 4	15 + EACT	2
	ii) SHR mem, CL	2 to 4	20+ EACT+4 per bit	2
75.	SAR reg/mem			
a)	SAR reg			
	i) SAR reg, 1	2	2	-
	ii) SAR reg, CL	2	8 + 4 per bit	-
b)	SAR mem			
	i) SAR mem, 1	2 to 4	15 + EACT	2
	ii) SAR mem, CL	2 to 4	20+EACT+4 per bit	2
76.	ROL reg/mem			
a)	ROL reg			
	i) ROL reg, 1	2	2	-
	ii) ROL reg, CL	2	8 + 4 per bit	-
b)	ROL mem			
	i) ROL mem, 1	2 to 4	15 + EACT	2
	ii) ROL mem, CL	2 to 4	20+EACT+4 per bit	2
77.	RCL reg/mem			
a)	RCL reg			
	i) RCL reg, 1	2	2	-
	ii) RCL reg, CL	2	8 + 4 per bit	-

Table - 3.5 continued...

S.No.	Mnemonic	Size of instruction (Number of bytes)	Clock period (or T-state) needed for instruction execution	Number of memory access (or transfer)
b)	RCL mem			
	i) RCL mem, 1	2 to 4	15 + EACT	2
	ii) RCL mem, CL	2 to 4	20+EACT+4 per bit	2
78.	ROR reg/mem			
a)	ROR reg			
	i) ROR reg, 1	2	2	-
	ii) ROR reg, CL	2	8 + 4 per bit	-
b)	ROR mem			
	i) ROR mem, 1	2 to 4	15 + EACT	2
	ii) ROR mem, CL	2 to 4	20+EACT+4 per bit	2
79.	RCR reg/mem			
a)	RCR reg			
	i) RCR reg, 1	2	2	-
	ii) RCR reg, CL	2	8 + 4 per bit	-
b)	RCR mem			
	i) RCR mem, 1	2 to 4	15 + EACT	2
	ii) RCR mem, CL	2 to 4	20+EACT+4 per bit	2
GROUP IV : STRING MANIPULATION INSTRUCTIONS				
80.	REP			
a)	REPZ/REPE	1	2	-
b)	REPNZ/REPNE	1	2	-
81.	MOVS		18 or 9 + 17 per repetition	
a)	MOVSB }	1		2
b)	MOVSW }			
82.	CMPS		22 or 9 + 22 per repetition	
a)	CMPSB }	1		2
b)	CMPSW }			
83.	SCAS		15 or 9 + 15 per repetition	
a)	SCASB }	1		1
b)	SCASW }			
84.	LODS		12 or 9 + 13 per repetition	
a)	LODSB }	1		1
b)	LODSW }			
85.	STOS		11 or 9 + 10 per repetition	
a)	STOSB }	1		1
b)	STOSW }			

Table - 3.5 continued...

S.No.	Mnemonic	Size of instruction (Number of bytes)	Clock period (or T-state) needed for instruction execution	Number of memory access (or transfer)
GROUP V : CONTROL TRANSFER INSTRUCTIONS				
86.	CALL disp16	3	19	1
87.	CALL reg/mem			
a)	CALL reg	2	16	1
b)	CALL mem	2 to 4	21 + EACT	2
88.	CALL addr$_{offset}$, addr$_{base}$	5	28	2
89.	CALL mem	2 to 4	37 + EACT	4
90.	RET (Return from call within segment)	1	8	1
91.	RET data16 (Return from call within segment adding immediate data to SP)	3	12	1
92.	RET (Return from intersegment call)	1	18	2
93.	RET data16 (Return from intersegment call adding immediate data to SP)	3	17	2
94.	JMP disp16 (Unconditional jump near-direct within segment)	3	15	-
95.	JMP disp8 (Unconditional jump short-direct within segment)	2	15	-
96.	JMP reg/mem (Unconditional jump near-indirect within segment)			
a)	JMP reg	2	11	-
b)	JMP mem	2 to 4	18 + EACT	1
97.	JMP addr$_{offset}$, addr$_{base}$ (Unconditional jump far-direct intersegment)	5	15	-
98.	JMP mem (Unconditional jump far-indirect intersegment)	2 to 4	24 + EACT	2
99.	JE/JZ disp8	2	16 or 4	-
100.	JL/JNGE disp8	2	16 or 4	-

Table - 3.5 continued...

S.No.	Mnemonic	Size of instruction (Number of bytes)	Clock period (or T-state) needed for instruction execution	Number of memory access (or transfer)
101.	JLE/JNG disp8	2	16 or 4	-
102.	JB/JNAE/JC disp8	2	16 or 4	-
103.	JBE/JNA disp8	2	16 or 4	-
104.	JP/JPE disp8	2	16 or 4	-
105.	JNB/JAE/JNC disp8	2	16 or 4	-
106.	JNBE/JA disp8	2	16 or 4	-
107.	JNP/JPO disp8	2	16 or 4	-
108.	JNO disp8	2	16 or 4	-
109.	JNS disp8	2	16 or 4	-
110.	JO disp8	2	16 or 4	-
111.	JS disp8	2	16 or 4	-
112.	JNE/JNZ disp8	2	16 or 4	-
113.	JNL/JGE disp8	2	16 or 4	-
114.	JNLE/JG disp8	2	16 or 4	-
115.	JCXZ disp8	2	18 or 6	-
116.	LOOP disp8	2	17 or 5	-
117.	LOOPZ/LOOPE disp8	2	18 or 6	-
118.	LOOPNZ/LOOPNE disp8	2	19 or 5	-
119.	INT type	2	51	5
120.	INT 3	1	52	5
121.	INTO	1	53 or 4	5
122.	IRET	1	24	3
GROUP VI : PROCESSOR CONTROL INSTRUCTIONS				
123.	CLC	1	2	-
124.	CMC	1	2	-
125.	STC	1	2	-
126.	CLD	1	2	-
127.	STD	1	2	-
128.	CLI	1	2	-
129.	STI	1	2	-
130.	HLT	1	2	-
131.	WAIT	1	3 + 5n	-
132.	ESC opcode, mem/reg			
a)	ESC opcode, mem	2 to 4	8 + EACT	1
b)	ESC opcode, reg	2	2	-
133.	LOCK	1	2	-
134.	NOP	1	3	-

TABLE - 3.6 : EFFECTIVE ADDRESS CALCULATION TIME

S.No.	Method of addressing memory	Number of clock cycles for EACT
1.	Direct addressing	6
2.	Register indirect addressing [BX] or [SI] or [DI]	5
3.	Based addressing [BX + disp] or [BP + disp]	9
4.	Indexed addressing [SI + disp] or [DI + disp]	9
5.	Based indexed addressing	
	a) Without displacement	
	i) [BP + DI] or [BX + SI]	7
	ii) [BP + SI] or [BX + DI]	8
	b) With displacement	
	i) [BP + DI + disp] or [BX + SI + disp]	11
	ii) [BP + SI + disp] or [BX + DI + disp]	12

TABLE - 3.7 : MEANINGS OF VARIOUS TERMS USED IN THE OPERAND FIELD OF INSTRUCTIONS IN TABLE - 3.5

Term	Meaning
reg/reg1/reg2	8-bit or 16-bit register
reg8	8-bit register
reg16	16-bit register
segreg	segment register
mem	8-bit or 16-bit memory
mem8	8-bit memory
mem16	16-bit memory
data	8-bit or 16-bit data

Term	Meaning
data8	8-bit data
data16	16-bit data
addr8	8-bit address
addr$_{offset}$	16-bit offset/effective address
addr$_{base}$	16-bit base address
disp8	8-bit displacement
disp16	16-bit displacement

Note : 1. *Possible choice for reg/reg1/reg2 are AL, AH, BL, BH, CL, CH, DL, DH, AX, BX, CX, DX, SI, DI, SP and BP.*

2. *Possible choice for reg8 are AL, AH, BL, BH, CL, CH, DL and DH.*

3. *Possible choice for reg16 are AX, BX, CX, DX, SI, DI, SP and BP.*

4. *Possible choice for segreg are DS, ES and SS.*

5. *The term mem stands for the 24 different methods of addressing memory data as shown in Table-3.4.*

TABLE - 3.8 : MEANING/EXPANSION OF MNEMONICS USED IN 8086 INSTRUCTION SET

S.No.	Mnemonic	Meaning
1.	AAA	ASCII adjust after addition.
2.	AAD	ASCII adjust before division.
3.	AAM	ASCII adjust after multiply.
4.	AAS	ASCII adjust after subtraction.
5.	ADC	Add two specified data along with carry.
6.	ADD	Add two specified data.
7.	AND	AND two specified data bit by bit.
8.	CALL	Call a procedure/subroutine.
9.	CBW	Convert byte to word (Sign extend byte to word).
10.	CLC	Clear carry flag (CF = 0).
11.	CLD	Clear direction flag (DF = 0).
12.	CLI	Clear interrupt enable flag (IF = 0).
13.	CMC	Complement the state of the carry flag (CF = $\overline{\text{CF}}$).
14	CMP	Compare two specified data.
15.	CMPS/CMPSB/CMPSW	Compare two string byte or two string word.
16.	CWD	Convert word to double word. (Sign extend the word to double word.)
17.	DAA	Decimal adjust after addition.
18.	DAS	Decimal adjust after subtraction.
19.	DEC	Decrement specified data.
20.	DIV	Divide unsigned word by byte, or unsigned double word by word.
21.	ESC	Escape to external coprocessor such as 8087.
22.	HLT	Halt until interrupt.
23.	IDIV	Divide signed word by byte, or signed double word by word.
24.	IMUL	Multiply signed byte by byte or signed word by word.
25.	IN	Copy a byte/word from specified port to accumulator.
26.	INC	Increment specified data.
27.	INT	Interrupt program execution. (Call interrupt service procedure.)
28.	INTO	Interrupt program execution, if OF = 1.
29.	IRET	Interrupt return.
30.	JA/JNBE	Jump if above/Jump if not below nor equal.
31.	JAE/JNB	Jump if above or equal/Jump if not below.
32.	JB/JNAE	Jump if below/Jump if not above nor equal.
33.	JBE/JNA	Jump if below or equal/Jump if not above.
34.	JC	Jump if CF = 1.
35.	JCXZ	Jump if CX = 0.
36.	JE/JZ	Jump if equal/Jump if ZF = 1.

Table - 3.8 continued...

S.No.	Mnemonic	Meaning
37.	JG/JNLE	Jump if greater/Jump if not less than nor equal.
38.	JGE/JNL	Jump if greater than or equal/Jump if not less than.
39.	JL/JNGE	Jump if less than/Jump if not greater than.
40.	JLE/JNG	Jump if less than or equal/Jump if not greater than.
41.	JMP	Jump to specified address to get next instruction.
42.	JNC	Jump if no carry (Jump if CF = 0).
43.	JNE/JNZ	Jump if not equal/Jump if not zero (ZF = 0).
44.	JNO	Jump if no overflow (Jump if OF = 0).
45.	JNP/JPO	Jump if not parity/Jump if parity odd. (PF = 0).
46.	JNS	Jump if not sign (Jump if sign flag = 0).
47.	JO	Jump if OF = 1.
48.	JP/JPE	Jump if parity/Jump if parity even (PF = 1).
49.	JS	Jump if SF = 1.
50.	LAHF	Load AH with the low byte of the flag register.
51.	LDS	Load DS-register and other specified register from memory.
52.	LEA	Load effective address of operand into specified register.
53.	LES	Load ES-register and other specified register from memory.
54.	LOCK	An instruction prefix which prevents another processor from taking bus while the adjacent instruction (i.e., instruction prefixed with lock) executes.
55.	LODS/LODSB /LODSW	Load string byte into AL or string word into AX.
56.	LOOP	Loop through a sequence of instructions until CX = 0.
57.	LOOPE/LOOPZ	Loop through a sequence of instructions while ZF = 1 and $CX \neq 0$.
58.	LOOPNE/LOOPNZ	Loop through a sequence of instructions while ZF = 0 and $CX \neq 0$.
59.	MOV	Move (copy) a byte/word from specified source to specified destination.
60.	MOVS/MOVSB /MOVSW	Move a byte/word from one string to another.
61.	MUL	Multiply two specified unsigned data.
62.	NEG	Negative of specified data (2's complement value of specified data).
63.	NOP	No action (operation) except fetch and decode.
64.	NOT	Invert/complement each bit of specified data.
65.	OR	OR two specified data bit by bit.
66.	OUT	Copy a byte/word from accumulator to specified port.
67.	POP	Move the top of stack to specified location.

Table - 3.8 continued...

S.No.	Mnemonic	Meaning
68.	POPF	Move the top of stack to flag register.
69.	PUSH	Push (copy) the specified register to top of stack.
70.	PUSHF	Push (copy) the flag register to top of stack.
71.	RCL	Rotate left through carry.
72.	RCR	Rotate right through carry.
73.	REP	An instruction prefix. Repeat adjacent instruction (i.e., instruction prefixed with REP) until CX = 0.
74.	REPE/REPZ	An instruction prefix. Repeat adjacent instruction until CX = 0 or ZF ≠ 1.
75.	REPNE/REPNZ	An instruction prefix. Repeat adjacent instruction until CX = 0 or ZF = 1.
76.	RET	Return from procedure to calling program.
77.	ROL	Rotate left to carry.
78.	ROR	Rotate right to carry.
79.	SAHF	Store (copy) AH-register to low byte of flag register.
80.	SAR	Arithmetic Right shift.
81.	SBB	Subtract specified data and carry flag from another specified data.
82.	SCAS/SCASB/ SCASW	Scan (compare) a string byte/word with accumulator.
83.	SHL/SAL	Logical left shift/Arithmetic left shift.
84.	SHR	Logical right shift.
85.	STC	Set carry flag (CF = 1).
86.	STD	Set direction flag (DF = 1).
87.	STI	Set interrupt enable flag (IF = 1).
88.	STOS/STOSB/ STOSW	Store byte from AL or word from AX into string.
89.	SUB	Subtract a specified data from another specified data.
90.	TEST	Test by performing logical AND operation of specified operands and modify flags.
91.	WAIT	Wait until signal on the test pin is **low**.
92.	XCHG	Exchange bytes or exchange words.
93.	XLAT	Translate a byte in AL using a table in memory.
94.	XOR	Exclusive-OR two specified data bit by bit.

3.5 INSTRUCTIONS AFFECTING FLAGS

The 8086 microprocessor has 9 flags. In this, six flags are altered by arithmetic and logical instructions, and three flags are used to control the processor operation.

The flags which are altered by arithmetic and logical instructions are carry flag, auxiliary carry flag, parity flag, zero flag, sign flag and overflow flag. The flags which are used to control the processor operation are trace flag (or single step trap), interrupt flag and direction flag.

The status of various flags after execution of arithmetic and logical instructions are listed in Table-3.9. The 8086 processor has instructions to directly set or clear the interrupt flag, direction flag and carry flag.

While servicing an interrupt the 8086 processor, save the status of flags in stack and the status of the flags are restored at the end of service procedure by executing IRET instruction.

The 8086 also has instruction to directly save the flags in stack(PUSHF) and to restore the saved flags(POPF).

TABLE - 3.9 : 8086 INSTRUCTIONS AFFECTING FLAGS

Instruction	Flags								
	O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
AAA	u	-	-	-	u	u	+	u	+
AAD	u	-	-	-	+	+	u	+	u
AAM	u	-	-	-	+	+	u	+	u
AAS	u	-	-	-	u	u	+	u	+
ADC	+	-	-	-	+	+	+	+	+
ADD	+	-	-	-	+	+	+	+	0
AND	0	-	-	-	+	+	u	+	0
CLC	-	-	-	-	-	-	-	-	0
CLD	-	0	-	-	-	-	-	-	-
CLI	-	-	0	-	-	-	-	-	-
CMC	-	-	-	-	-	-	-	-	+
CMP	+	-	-	-	+	+	+	+	+
CMPS	+	-	-	-	+	+	+	+	+
DAA	u	-	-	-	+	+	+	+	+
DAS	u	-	-	-	+	+	+	+	+
DEC	+	-	-	-	+	+	+	+	-
DIV	u	-	-	-	u	u	u	u	u
IDIV	u	-	-	-	u	u	u	u	u

Table - 3.9 continued...

Instruction	Flags								
	O	D	I	T	S	Z	A	P	C
IMUL	+	-	-	-	u	u	u	u	+
INC	+	-	-	-	+	+	+	+	-
INT	-	-	0	0	-	-	-	-	-
INTO	-	-	0	0	-	-	-	-	-
IRET	r	r	r	r	r	r	r	r	r
MUL	+	-	-	-	u	u	u	u	+
NEG	+	-	-	-	+	+	+	+	+
OR	0	-	-	-	+	+	u	+	0
POPF	r	r	r	r	r	r	r	r	r
RCL	+	-	-	-	-	-	-	-	+
RCR	+	-	-	-	-	-	-	-	+
ROL	+	-	-	-	-	-	-	-	+
ROR	+	-	-	-	-	-	-	-	+
SAHF	-	-	-	-	r	r	r	r	r
SAL/SHL	+	-	-	-	+	+	u	+	+
SAR	+	-	-	-	+	+	u	+	+
SBB	+	-	-	-	+	+	+	+	+
SCAS	+	-	-	-	+	+	+	+	+
SHR	+	-	-	-	+	+	u	+	+
STC	-	-	-	-	-	-	-	-	1
STD	-	1	-	-	-	-	-	-	-
STI	-	-	1	-	-	-	-	-	-
SUB	+	-	-	-	+	+	+	+	+
TEST	0	-	-	-	+	+	u	+	0
XOR	0	-	-	-	+	+	u	+	0

Note : "+" → *Flag is altered and defined (i.e., set or cleared according to the result).*
"u" → *Flag is undefined (i.e., altered but not defined).*
"–" → *Flag is not altered / affected.*
"r" → *The flag is restored from previous saved value.*
"1" → *Set to 1.*
"0" → *Cleared to 0.*

3.6 DATA TRANSFER INSTRUCTIONS

The instruction set of 8086 microprocessor includes a variety of instructions to transfer data/address into registers, memory locations and IO ports. The various mnemonics used for data transfer instructions are MOV, XCHG, PUSH, POP, IN, OUT, etc., and they perform any one of the following operations:

- Copy the content of a register to another register.
- Copy the content of a register to memory or vice versa.
- Load the immediate operand to memory/register.
- Copy the content of a register/memory to segment register (excluding CS-register) or vice versa.
- Exchange the content of two registers or register and memory.
- Copy the content of accumulator to port or vice versa.
- Load effective address in segment registers.

The data transfer instructions generally involve two operands : source operand and destination operand. The source and destination operands should be of same size, either both the operand size should be byte or word. This means that only 8-bit data can be moved to 8-bit register/memory and only 16-bit data can be moved to 16-bit register/memory. Moving the content of 8-bit register to 16-bit register/memory or vice versa is illegal.

The source can be a register or a memory location or an immediate data. The destination can be a register or a memory location. In double operand instructions, the source and destination cannot refer to memory locations in the same instruction. Therefore, copying the content of one memory location to another memory location in a single instruction is not possible (except PUSH instruction).

The data transfer instructions (except POPF and SAHF instructions) do not affect the flags of 8086. While executing the POPF instruction, the previously stored status of the flag is restored in the flag register. The instruction SAHF is used to modify the content of the flag register.

The data transfer instructions of 8086 are listed in Table-3.10, with a brief description about each instruction.

Note : 1. *The terms MA, MA_S and MA_E used in symbolic description of instructions refer to physical memory address of data segment, stack segment and extra segment respectively.*

2. *The register or memory enclosed by brackets in symbolic description refer to content of register or memory.*

TABLE - 3.10 : DATA TRANSFER INSTRUCTIONS

S.No.	Instruction	Symbolic representation	Explanation
1.	MOV reg2/mem, reg1/mem		
a)	MOV reg2, reg1	(reg2) ← (reg1)	The content of register1 is transferred to register2.
b)	MOV mem, reg1	(mem) ← (reg1)	The content of register1 is transferred to memory.
c)	MOV reg2, mem	(reg2) ← (mem)	The content of memory is transferred to register2.
2.	MOV reg/mem, data		
a)	MOV reg, data	(reg) ← data	The data given in the instruction is transferred to register.
b)	MOV mem, data	(mem) ← data	The data given in the instruction is transferred to memory.
3.	MOV reg, data	(reg) ← data	The data given in the instruction is transferred to register.
4.	MOV A, mem		
a)	MOV AL, mem	(AL) ← (mem)	The content of (8-bit) memory is transferred to 8-bit accumulator (AL).
b)	MOV AX, mem	(AX) ← (mem)	The content of (16-bit) memory is transferred to accumulator (AX).
5.	MOV mem, A		
a)	MOV mem, AL	(mem) ← (AL)	The content of 8-bit accumulator(AL) is transferred to memory.
b)	MOV mem, AX	(mem) ← (AX)	The content of 16-bit accumulator (AX) is transferred to memory.
6.	MOV segreg, reg16/mem		
a)	MOV segreg, reg16	(segreg) ← (reg16)	The content of 16-bit register is transferred to segment register.
b)	MOV segreg, mem	(segreg) ← (mem)	The content of (16-bit) memory is transferred to segment register.

Table - 3.10 continued...

S.No.	Instruction	Symbolic representation	Explanation
7.	MOV reg16/mem, segreg		
a)	MOV reg16, segreg	(reg16) ← (segreg)	The content of segment register is transferred to 16-bit register.
b)	MOV mem, segreg	(mem) ← (segreg)	The content of segment register is transferred to 16-bit memory.
8.	PUSH reg16/mem	(SP) ← (SP) – 2	
a)	PUSH reg16	$MA_S = (SS) \times 16_{10} + (SP)$ $(MA_S ; MA_S + 1) \leftarrow (reg16)$	The stack pointer is decremented by 2 and the content of 16-bit register is pushed to stack memory pointed by SP.
b)	PUSH mem	(SP) ← (SP) – 2 $MA_S = (SS) \times 16_{10} + (SP)$ $(MA_S ; MA_S + 1) \leftarrow (mem)$	The stack pointer is decremented by 2 and the content of (16-bit) memory is pushed to stack memory pointed by SP.
9.	PUSH reg16	(SP) ← (SP) – 2 $MA_S = (SS) \times 16_{10} + (SP)$ $(MA_S ; MA_S + 1) \leftarrow (reg16)$	The stack pointer is decremented by 2 and the content of 16-bit register is pushed to stack memory pointed by SP.
10.	PUSH segreg	(SP) ← (SP) – 2 $MA_S = (SS) \times 16_{10} + (SP)$ $(MA_S ; MA_S + 1) \leftarrow (segreg)$	The stack pointer is decremented by 2 and the content of segment register is pushed to stack memory pointed by SP.
11.	PUSHF	(SP) ← (SP) – 2 $MA_S = (SS) \times 16_{10} + (SP)$ $(MA_S ; MA_S + 1) \leftarrow (Flags)$	The stack pointer is decremented by 2 and the content of 16-bit flag register is pushed to stack memory pointed by SP.
12.	POP reg16/mem		
a)	POP reg16	$MA_S = (SS) \times 16_{10} + (SP)$ $(reg16) \leftarrow (MA_S ; MA_S + 1)$ (SP) ← (SP) + 2	The content of stack memory pointed by SP is moved to 16-bit register and the stack pointer is incremented by 2.
b)	POP mem	$MA_S = (SS) \times 16_{10} + (SP)$ $(mem) \leftarrow (MA_S ; MA_S + 1)$ (SP) ← (SP) + 2	The content of (16-bit) stack memory pointed by SP is moved to memory and the stack pointer is incremented by 2.

Table - 3.10 continued...

S.No.	Instruction	Symbolic representation	Explanation
13.	POP reg16	$MA_S = (SS) \times 16_{10} + (SP)$ $(reg16) \leftarrow (MA_S ; MA_S + 1)$ $(SP) \leftarrow (SP) + 2$	The content of (16-bit) stack memory pointed by SP is moved to 16-bit register and the stack pointer is incremented by 2.
14.	POP segreg	$MA_S = (SS) \times 16_{10} + (SP)$ $(segreg) \leftarrow (MA_S ; MA_S + 1)$ $(SP) \leftarrow (SP) + 2$	The content of (16-bit) stack memory pointed by SP is moved to segment register and the stack pointer is incremented by 2.
15.	POPF	$MA_S = (SS) \times 16_{10} + (SP)$ $(Flags) \leftarrow (MA_S ; MA_S + 1)$ $(SP) \leftarrow (SP) + 2$	The content of (16-bit) stack memory pointed by SP is moved to flag register and the stack pointer is incremented by 2.
16.	XCHG reg2/mem, reg1		
a)	XCHG reg2, reg1	$(reg2) \leftrightarrow (reg1)$	The content of two registers are exchanged.
b)	XCHG mem, reg1	$(mem) \leftrightarrow (reg1)$	The content of memory and register are exchanged.
17.	XCHG AX, reg16	$(AX) \leftrightarrow (reg16)$	The content of accumulator and 16-bit register are exchanged.
18.	XLAT	$MA = (DS) \times 16_{10} + (BX) + (AL)$ $(AL) \leftarrow (MA)$	The content of (8-bit) memory is transferred to AL. The effective address of memory is given by sum of BX and AL.
19.	IN A, [DX]	$PORT_{addr} = (DX)$	
a)	IN AL, [DX]	$(AL) \leftarrow (PORT)$	The content of (8-bit) port whose address is specified by DX-register is transferred to 8-bit accumulator (AL).
b)	IN AX, [DX]	$PORT_{addr} = (DX)$ $(AX) \leftarrow (PORT)$	The content of (16-bit) port whose address is specified by DX-register is transferred to accumulator (AX).
20.	IN A, addr8		
a)	IN AL, addr8	$(AL) \leftarrow (addr8)$	The content of (8-bit) port whose address is given in the instruction is transferred to 8-bit accumulator (AL).
b)	IN AX, addr8	$(AX) \leftarrow (addr\ 8)$	The content of (16-bit) port whose address is given in the instruction is transferred to accumulator (AX).

Table - 3.10 continued...

S.No.	Instruction	Symbolic representation	Explanation
21.	OUT [DX], A	$\text{PORT}_{\text{addr}}$ = (DX)	
a)	OUT [DX], AL	(PORT) ← (AL)	The content of 8-bit accumulator (AL) is transferred to the (8-bit) port whose address is specified by DX-register.
b)	OUT [DX], AX	$\text{PORT}_{\text{addr}}$ = (DX) (PORT) ← (AX)	The content of 16-bit accumulator (AX) is transferred to the (16-bit) port, whose address is specified by DX-register.
22.	OUT addr8, A		
a)	OUT addr8, AL	(addr8) ← (AL)	The content of 8-bit accumulator (AL) is transferred to the (8-bit) port whose address is given in the instruction.
b)	OUT addr8, AX	(addr8) ← (AX)	The content of 16-bit accumulator (AX) is transferred to the (16-bit) port whose address is given in the instruction.
23.	LEA reg16, mem	(reg16) ← EA	The 16-bit register is loaded with the **E**ffective **A**ddress (EA) of the memory location specified by the instruction.
24.	LDS reg16, mem	(reg16) ← (mem) (DS) ← (mem+2)	The word from first two memory locations is moved to the 16-bit register and the word from next two memory locations is moved to DS-register.
25.	LES reg16, mem	(reg16) ← (mem) (ES) ← (mem+2)	The word from first two memory locations is transferred to the 16-bit register and the word from next two memory locations is moved to ES-register.
26.	LAHF	(AH) ← (lower byte flag register)	The content of the lower byte flag register is transferred to the higher byte register of the accumulator.
27.	SAHF	(lower byte flag register) ← (AH)	The content of the higher byte register of the accumulator is moved to lower byte flag register.

3.7 ARITHMETIC INSTRUCTIONS

The arithmetic group includes instructions for performing the following operations:

- Addition or subtraction of binary, BCD or ASCII data.
- Multiplication or division of signed or unsigned binary data.
- Increment or decrement or comparison of binary data.

The mnemonics used for arithmetic instructions are ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP, etc.

The arithmetic instructions generally involve two operands : source operand and destination operand. The source can be a register or a memory location or an immediate data. The destination can be a register or memory. The result of arithmetic operation is stored in destination register or memory except in case of comparison. (In comparison the result is used to modify the flags and then the result is discarded.)

In double operand arithmetic instructions, the source and destination cannot refer to memory locations in the same instruction. Therefore performing arithmetic operation directly on two memory data is not possible.

In double operand arithmetic instructions except division, the source and destination operand should be of same size, either both the operand size should be byte or word. In all arithmetic instructions employing immediate addressing mode, if the immediate operand/data is 8-bit and the size of register/memory is 16-bit then the 8-bit immediate operand is sign extended to 16-bit. The arithmetic operation is performed between the sign extended data and the content of register/memory.

The arithmetic instructions alter the flags of 8086. The processor use the result of arithmetic operation to alter the flag. The flags reflect the status of result (for example, the result is zero or not; result has carry or not, etc.).

The arithmetic instructions of 8086 are listed in Table-4.11, with a brief description about each instruction.

TABLE - 3.11 : ARITHMETIC INSTRUCTIONS

S.No.	Instruction	Symbolic representation	Explanation
28.	ADD reg2/mem, reg1/mem		
a)	ADD reg2, reg1	(reg2) ← (reg1) + (reg2)	The content of two registers are added and the result is stored in register2.
b)	ADD reg2, mem	(reg2) ← (reg2) + (mem)	The content of register2 and memory are added and the result is stored is register2.
c)	ADD mem, reg1	(mem) ← (mem) + (reg1)	The content of register1 and memory are added and the result is stored in memory.
29.	ADD reg/mem, data		
a)	ADD reg, data	(reg) ← (reg) + data	The data given in the instruction is added to the content of register and the result is stored in register.
b)	ADD mem, data	(mem) ← (mem) + data	The data given in the instruction is added to the content of memory and the result is stored in memory.
30.	ADD A, data		
a)	ADD AL, data8	(AL) ← (AL) + data8	The 8-bit data given in the instruction is added to the content of 8-bit accumulator and the result is stored in 8-bit accumulator (AL).
b)	ADD AX, data16	(AX) ← (AX) + data16	The 16-bit data given in the instruction is added to the content of 16-bit accumulator and the result is stored in 16-bit accumulator (AX).
31.	ADC reg2/mem, reg1/mem		
a)	ADC reg2, reg1	(reg2) ← (reg2) + (reg1) + CF	The content of registers and carry flag are added and the result is stored in register2.
b)	ADC reg2, mem	(reg2) ← (reg2) + (mem) + CF	The carry flag and the content of memory are added to register2 and the result is stored in register2.
c)	ADC mem, reg1	(mem) ← (mem) + (reg1) + CF	The carry flag and the content of register1 are added to memory and the result is stored in memory.

Table - 3.11 continued...

S.No.	Instruction	Symbolic representation	Explanation
32. a)	ADC reg/mem, data ADC reg, data	(reg) ← (reg) + data + CF	The data given in the instruction and the carry flag are added to the content of register and the result is stored in register.
b)	ADC mem, data	(mem) ← (mem) + data + CF	The data given in instruction and the carry flag are added to the content of memory and the result is stored in memory.
33. a)	ADC A, data ADC AL, data8	(AL) ← (AL) + data8 + CF	The 8-bit data given in instruction and the carry flag are added to content of 8-bit accumulator(AL) and the result is stored in 8-bit accumulator(AL).
b)	ADC AX, data16	(AX) ← (AX) + data16 + CF	The 16-bit data given in instruction and the carry flag are added to content of accumulator(AX) and the result is stored in 16-bit accumulator (AX).
34.	AAA	Adjust AL to unpacked BCD 1. (AL) ← (AL) & $0F_H$ 2. If AL > 9 or AF = 1 then (AL) ← (AL) + 6 (AH)← (AH) + 1 CF ← 1 ; AF ← 1 (AL) ← (AL) & $0F_H$	This instruction is executed after addition of two ASCII data to convert the result in AL to correct unpacked BCD.
35.	DAA	Adjust AL to packed BCD. 1. If lower nibble of AL>9 or AF=1 then (AL) ← (AL)+06 ; AF ← 1 2. If higher nibble of AL>9 or CF=1 then (AL) ← (AL) + 60 ; CF ← 1	This instruction is executed after addition of two packed BCD data to convert the result in AL to packed BCD data.
36. a)	SUB reg2/mem, reg1/mem SUB reg2, reg1	(reg2) ← (reg2) – (reg1)	The content of register1 is subtracted from the register2 and result is stored in register2.
b)	SUB reg2, mem	(reg2) ← (reg2) – (mem)	The content of memory is subtracted from the content of register2 and result is stored in register2.
c)	SUB mem, reg1	(mem) ← (mem) – (reg1)	The content of register1 is subtracted from the content of memory and the result is stored in memory.

Table - 3.11 continued...

S.No.	Instruction	Symbolic representation	Explanation
37.	SUB reg/mem, data		
a)	SUB reg, data	(reg) ← (reg) – data	The data given in the instruction is subtracted from the register and the result is stored in register.
b)	SUB mem, data	(mem) ← (mem) – data	The data given in the instruction is subtracted from the content of memory and the result is stored in memory.
38.	SUB A, data		
a)	SUB AL, data8	(AL) ← (AL) – data8	The 8-bit data given in the instruction is subtracted from AL and the result is stored in AL-register.
b)	SUB AX, data16	(AX) ← (AX) – data16	The 16-bit data given in the instruction is subtracted from accumulator(AX) and the result is stored in accumulator(AX).
39.	SBB reg2/mem, reg1/mem		
a)	SBB reg2, reg1	(reg2) ← (reg2) – (reg1) – CF	The carry flag and the content of register1 are subtracted from register2 and the result is stored in register2.
b)	SBB reg2, mem	(reg2) ← (reg2) – (mem) – CF	The carry flag and the content of memory are subtracted from register2 and the result is stored in register2.
c)	SBB mem, reg1	(mem) ← (mem) – (reg1) – CF	The carry flag and the content of register1 are subtracted from the content of memory and the result is stored in memory.
40.	SBB reg/mem, data		
a)	SBB reg, data	(reg) ← (reg) – data – CF	The carry flag and the data given in instruction are subtracted from register and the result is stored in register.
b)	SBB mem, data	(mem) ← (mem) – data – CF	The carry flag and the data given in the instruction are subtracted from the content of memory and the result is stored in memory.

Table - 3.11 continued...

S.No.	Instruction	Symbolic representation	Explanation
41. a)	SBB A, data SBB AL, data8	(AL) ← (AL) – data8 – CF	The carry flag and the 8-bit data given in the instruction are subtracted from AL-register and the result is stored in AL-register.
b)	SBB AX, data16	(AX) ← (AX) – data16 – CF	The carry flag and the 16-bit data given in the instruction are subtracted from the accumulator and the 16-bit result is stored in accumulator.
42.	AAS	Adjust AL to unpacked BCD. 1. (AL) ← (AL) & $0F_H$ 2. If (AL) > 9 or AF = 1 then, (AL) ← (AL) – 6 ; (AH) ← (AH) – 1 AF ← 1 ; CF ← 1 ; AL ← (AL) & $0F_H$	This instruction is executed after subtraction of ASCII data to convert the result in AL to correct unpacked BCD.
43.	DAS	Adjust AL to packed BCD. 1. If lower nibble of AL>9 or AF = 1 then, (AL) ← (AL) –6 ; AF ← 1. 2. If upper nibble of AL>9 or CF=1 then, (AL) ← (AL) –60 ; CF ← 1.	This instruction is executed after subtraction of packed BCD data to convert the result in AL to packed BCD data.
44. a)	MUL reg/mem MUL reg	For byte (AX) ← (AL) × (reg8) For word (DX)(AX) ← (AX) × (reg16)	It is unsigned multiplication.While using this instruction the content of accumulator and register should be unsigned binary and the result is also unsigned binary. For byte operand : The content of 8-bit accumulator(AL) is multiplied by the content of 8-bit register and the product is stored in AX-register. For word operand: The content of 16-bit accumulator(AX) is multiplied by the content of 16-bit register.The lower word of the result is stored in AX-register and the upper word in DX-register.
b)	MUL mem	For byte (AX) ← (AL) × (mem8) For word (DX)(AX) ← (AX) × (mem16)	This instruction is same as MUL reg, except that one of the source operand is in memory instead of register.

Table - 3.11 continued...

S.No.	Instruction	Symbolic representation	Explanation
45.	IMUL reg/mem		
a)	IMUL reg	For byte (AX) ← (AL) × (reg8) For word (DX) (AX) ← (AX) × (reg16)	It is signed multiplication.While using this instruction the content of accumulator and register should be sign extended binary in 2's complement form and the result is also sign extended binary. For byte operand : The content of AL is multiplied by the content of 8-bit register and the sign extended result is stored in AX. For word operand : The content of AX is multiplied by the content of 16-bit register. The lower word of sign extended result is stored in AX-register and the upper word in DX-register.
b)	IMUL mem	For byte (AX) ← (AX) × (mem8) For word (DX) (AX) ← (AX) × (mem16)	This instruction is same as IMUL reg, except that one of the source operand is in memory instead of register.
46.	AAM	Adjust AH to unpacked BCD data. $(AH) = (AL) \div 0A_H$ $(AL) = (AL)$ MOD $0A_H$ *Note :* $\boxed{0A_H = 10_{10}}$	After multiplication of two 8-bit unpacked BCD data the result in AX will be in binary. This instruction can be executed after multiplication to convert the result in AX to unpacked BCD.

Table - 3.11 continued...

S.No.	Instruction	Symbolic representation	Explanation
47. a)	DIV reg/mem DIV reg	For 16-bit ÷ 8-bit (AL) ← (AX) ÷ (reg8) Quotient (AH) ← (AX) MOD (reg8) Remainder For 32-bit ÷ 16-bit (AX) ← (DX)(AX) ÷ (reg16) Quotient (DX) ← (DX)(AX) MOD (reg16) Remainder	It is unsigned division. While using this instruction the content of accumulator and register should be an unsigned binary. The result is also an unsigned binary. This instruction divides the content of accumulator by the content of register. Division by zero will generate a type-0 interrupt. For 16-bit ÷ 8-bit : The quotient is stored in AL-register and the remainder is stored in AH-register. For 32-bit ÷ 16-bit : The quotient is stored in AX (accumulator) while the remainder is stored in DX-register.
b)	DIV mem	For 16-bit÷8-bit (AL) ← (AX) ÷ (mem8) Quotient (AH) ← (AX) MOD (mem8) Remainder For 32-bit ÷ 16-bit (AX) ← (DX)(AX) ÷ (mem16) Quotient (DX) ← (DX)(AX) MOD (mem16) Remainder	This instruction is same as DIV reg except that the divisor is stored in memory instead of register.

Table - 3.11 continued...

S.No.	Instruction	Symbolic representation	Explanation
48.	IDIV reg/mem		
a)	IDIV reg	For 16-bit ÷ 8-bit (AL) ← (AX) ÷ (reg8) (AH) ← (AX) MOD (reg8) For 32-bit ÷ 16-bit (AX) ← (DX)(AX) ÷ (reg16) (DX) ← (DX)(AX) MOD (reg16)	It is signed division.While using this instruction the content of accumulator and register should be sign extended binary. The sign of quotient depends on the sign of the dividend and the divisor. The sign of the remainder will be same as that of the dividend. Division by zero generates a type-0 interrupt. For 16-bit ÷ 8-bit : The quotient is stored in AL-register and the remainder is stored in AH-register. For 32-bit ÷ 16-bit : The quotient is stored in AX (accumulator) while the remainder is stored in DX-register.
b)	IDIV mem	For 16-bit ÷ 8-bit (AL) ← (AX) ÷ (mem8) (AH) ← (AX) MOD (mem8) For 32-bit ÷ 16-bit (AX) ← (DX)(AX) ÷ (mem16) (DX) ← (DX)(AX) MOD (mem16)	This instruction is same as IDIV reg, except that the signed divisor is stored in memory instead of register.
49.	AAD	Adjust AL to packed BCD. (AL) ← (AH) × 16_{10} + (AL) (AH) ← 00_H	The unpacked BCD digit in AH and AL registers are converted to equivalent packed BCD data and stored in AL-register. This instruction should be used before the use of division instruction.
50.	NEG mem/reg		
a)	NEG reg	(reg) ← 0 − (reg)	Changes the sign of the register content. (The register content will be replaced by its 2's complement value.)
b)	NEG mem	(mem) ← 0 − (mem)	Changes the sign of the memory content. (The memory content is replaced by its 2's complement value.)

Table - 3.11 continued...

S.No.	Instruction	Symbolic representation	Explanation
51.	INC reg8/mem		
a)	INC reg8	(reg8) ← (reg8) + 1	The content of the 8-bit register is incremented by 1.
b)	INC mem	(mem) ← (mem) + 1	The content of the memory is incremented by 1.
52.	INC reg16	(reg16) ← (reg16) + 1	The content of the 16-bit register is incremented by 1.
53.	DEC reg8/mem		
a)	DEC reg8	(reg8) ← (reg8) −1	The content of the 8-bit register is decremented by 1.
b)	DEC mem	(mem) ← (mem) −1	The content of memory is decremented by 1.
54.	DEC reg16	(reg16) ← (reg16) − 1	The content of the 16-bit register is decremented by 1.
55.	CBW	Bit-7 of AL is moved to all the bits of AH-register. 1. If AL = 1xxx xxxx (ie., ≥ 80_H) then AH ← 1111 1111(FF_H) 2. If AL = 0xxx xxxx (ie., < 80_H) then AH ← 0000 0000 (00_H)	Sign extends the content of AL to AH-register by copying the sign bit of AL to all the bits in AH-register.
56.	CWD	Bit-15 of AX is moved to all the bits of DX-register. 1. If AX = 1xxx xxxx xxxx xxxx (ie., ≥ 8000_H) then DX ← 1111 1111 1111 1111 ($FFFF_H$) 2. If AX = 0xxx xxxx xxxx xxxx (ie., < 8000_H) then DX ← 0000 0000 0000 0000 (0000_H)	Sign extends the content of AX to DX-register by copying the sign bit of AX to all the bits in DX-register.

Table - 3.11 continued...

S.No.	Instruction	Symbolic representation	Explanation
57.	CMP reg2/mem, reg1/mem	Modify flags ← (reg2) − (reg1)	
a)	CMP reg2, reg1	If (reg2) > (reg1) then CF=0 ; ZF=0 ; SF=0 If (reg2) < (reg1) then CF=1 ; ZF=0 ; SF=1 If (reg2) = (reg1) then CF=0 ; ZF=1 ; SF=0	The content of two registers are compared by subtraction and the result is used to modify the flags. The content of the two registers are not altered.
b)	CMP reg2, mem	Modify flags ← (reg2) − (mem) If (reg2) > (mem) then CF=0 ; ZF=0 ; SF=0 If (reg2) < (mem) then CF=1 ; ZF=0 ; SF=1 If (reg2) = (mem) then CF=0 ; ZF=1 ; SF=0	The content of memory and register2 are compared by subtraction and the result is used to modify the flags. The content of memory and register are not altered.
c)	CMP mem, reg1	Modify flags ← (mem) − (reg1) If (mem) > (reg1) then CF=0 ; ZF=0 ; SF=0 If (mem) < (reg1) then CF=1 ; ZF=0 ; SF=1 If (mem) = (reg1) then CF=0 ; ZF=1 ; SF=0	The content of memory and register1 are compared by subtraction and the result is used to modify flags. The content of memory and register are not altered.
58.	CMP reg/mem, data	Modify flags ← (reg) − data	The content of reg/mem is compared with data given in the instruction by subtraction and the result is used to modify the flags. The content of reg/mem is not altered.
a)	CMP reg, data	If (reg) >data then, CF=0 ; ZF=0 ; SF=0 If (reg) <data then, CF=1 ; ZF=0 ; SF=1 If (reg) = data then, CF=0 ; ZF=1 ; SF=0	
b)	CMP mem, data	Modify flags ← (mem) − data If (mem) >data then, CF=0 ; ZF=0 ; SF=0 If (mem) <data then, CF=1 ; ZF=0 ; SF=1 If (mem) = data then, CF=0 ; ZF=1 ; SF=0	

Table - 3.11 continued...

S.No.	Instruction	Symbolic representation	Explanation
59.	CMP A, data	Modify flags ← (AL) – data 8	The content of accumulator is compared with data given in the instruction by subtraction and the result is used to modify the flags. The content of accumulator is not altered.
a)	CMP AL, data8	If (AL) > data8 then, CF=0 ; ZF=0 ; SF=0 If (AL) < data8 then, CF=1 ; ZF=0 ; SF=1 If (AL) = data8 then, CF=0 ; ZF=1 ; SF=0	
b)	CMP AX, data16	Modify flags ← (AX) – data16 If (AX) > data16 then, CF=0 ; ZF=0 ; SF=0 If (AX) < data16 then, CF=1 ; ZF=0 ; SF=1 If (AX) = data16 then, CF=0 ; ZF=1 ; SF=0	

3.8 LOGICAL INSTRUCTIONS

The logical group includes instructions for performing AND, OR, Exclusive-OR, complement, shift and rotate operations on binary data. The mnemonics used for logical instructions are AND, OR, XOR, TEST, SHR, SHL, RCR, RCL, etc.

The logical instructions except shift and rotate involves two operands: source operand and destination operand. The source operand can be a register or memory location or immediate data. The destination can be a register or memory. The result of logical operation is stored in destination register or memory except in case of TEST. (In TEST operation the result is used to modify the flags and then the result is discarded.)

In double operand logical instructions the source and destination cannot refer to memory locations in the same instruction. Therefore performing logical operation directly on two memory data is not possible.

In double operand logical instructions, the source and destination operand should be of same size, either both the operand size should be byte or word.

The logical instructions alter the flags of 8086. The processor uses the result of logical operation to alter the flag. The flags reflect the status of the result (for example, whether the result is zero or not).

The logical instructions of 8086 are listed in Table-3.12, with a brief description about each instruction.

TABLE - 3.12 : LOGICAL INSTRUCTIONS

S.No.	Instruction	Symbolic representation	Explanation
60.	AND reg2/mem, reg1/mem		
a)	AND reg2, reg1	(reg2) ← (reg2) and (reg1)	The content of registers are logically ANDed bit by bit and the result is stored in register2.
b)	AND reg2, mem	(reg2) ← (reg2) and (mem)	The content of register2 and memory are logically ANDed bit by bit and the result is stored in register2.
c)	AND mem, reg1	(mem) ← (mem) and (reg1)	The content of the memory and register1 are logically ANDed bit by bit and result is stored in memory.
61.	AND reg/mem, data		
a)	AND reg, data	(reg) ← (reg) and data	The content of register and the data given in the instruction are bit by bit logically ANDed and result is stored in register.
b)	AND mem, data	(mem) ← (mem) and data	The content of memory and the data given in the instruction are bit by bit logically ANDed and result is stored in memory.
62.	AND A, data		
a)	AND AL, data8	(AL) ← (AL) and data8	The content of 8-bit accumulator(AL) and 8-bit data given in the instruction are bit by bit logically ANDed and result is stored in AL.
b)	AND AX, data16	(AX) ← (AX) and data16	The content of accumulator(AX) and 16-bit data given in the instruction are bit by bit logically ANDed and the result is stored in AX.

Table - 3.12 continued...

S.No.	Instruction	Symbolic representation	Explanation
63.	OR reg2/mem, reg1/mem		
a)	OR reg2, reg1	(reg2) ← (reg2) | (reg1)	The content of registers are bit by bit logically ORed and result is stored in register2.
b)	OR reg2, mem	(reg2) ← (reg2) | (mem)	The content of register2 and memory are bit by bit logically ORed and result is stored in register2.
c)	OR mem, reg1	(mem) ← (mem) | (reg1)	The content of memory and register1 are bit by bit logically ORed and result is stored in memory.
64.	OR reg/mem, data		
a)	OR reg, data	(reg) ← (reg) | data	The content of register and the data given in the instruction are bit by bit logically ORed and the result is stored in register.
b)	OR mem, data	(mem) ← (mem) | data	The content of memory and the data given in the instruction are bit by bit logically ORed and the result is stored in memory.
65.	OR A, data		
a)	OR AL, data8	(AL) ←(AL) | data8	The content of 8-bit accumulator (AL) and 8-bit data given in the instruction are bit by bit logically ORed and the result is stored in AL.
b)	OR AX, data16	(AX) ← (AX) | data16	The content of 16-bit accumulator (AX) and 16-bit data given in the instruction are bit by bit logically ORed and the result is stored in AX.

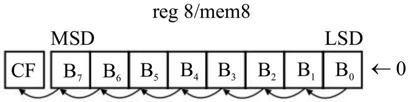
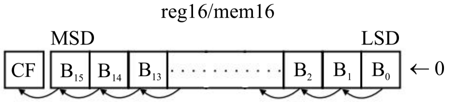
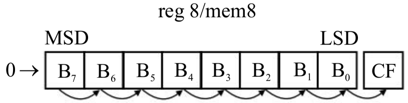
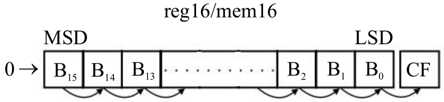
Table - 3.12 continued...

S.No.	Instruction	Symbolic representation	Explanation
66.	XOR reg2/mem, reg1/mem		
a)	XOR reg2, reg1	(reg2) ← (reg2) ^ (reg1)	The content of registers are bit by bit Exclusive-ORed and the result is stored in register2.
b)	XOR reg2, mem	(reg2) ← (reg2) ^ (mem)	The content of register and memory are bit by bit Exclusive-ORed and result is stored in register2.
c)	XOR mem, reg1	(mem) ←(mem) ^ (reg1)	The content of memory and register1 are bit by bit Exclusive-ORed and the result is stored in memory.
67.	XOR reg/mem, data		
a)	XOR reg, data	(reg) ← (reg) ^ data	The content of register and the data given in the instruction are bit by bit Exclusive-ORed and the result is stored in register.
b)	XOR mem, data	(mem) ← (mem) ^ data	The content of memory and the data given in the instruction are bit by bit Exclusive-ORed and result is stored in memory.
68.	XOR A, data		
a)	XOR AL, data8	(AL) ← (AL) ^ data8	The content of 8-bit accumulator (AL) and 8-bit data given in the instruction are bit by bit Exclusive-ORed and result is stored in AL.
b)	XOR AX, data16	(AX) ← (AX) ^ data16	The content of 16-bit accumulator (AX) and 16-bit data given in the instruction are bit by bit Exclusive-ORed and result is stored in AX.

Table - 3.12 continued...

S.No.	Instruction	Symbolic representation	Explanation
69.	TEST reg2/mem, reg1/mem		
a)	TEST reg2, reg1	Modify flags ← (reg2) and (reg1)	The content of registers are ANDed and the result is used to modify flags. The content of registers are not altered.
b)	TEST reg2, mem	Modify flags ← (reg2) and (mem)	The content of register and memory are ANDed and the result is used to modify the flags. The content of register/memory are not altered.
c)	TEST mem, reg1	Modify flags ← (mem) and (reg1)	
70.	TEST reg/mem, data		
a)	TEST reg, data	Modify flags ← (reg) and data	The content of register and the data given in the instruction are ANDed and the result is used to modify flags. The content of register is not altered.
b)	TEST mem, data	Modify flags ← (mem) and data	The content of memory and the data given in the instruction are ANDed and the result is used to modify flags.The content of memory is not altered.
71.	TEST A, data		
a)	TEST AL, data8	Modify flags ← (AL) and data8	The content of accumulator and the data given in the instruction are logically ANDed and the result is used to modify flags. The content of accumulator is not altered.
b)	TEST AX, data16	Modify flags ← (AX) and data16	
72.	NOT reg/mem		
a)	NOT reg	(reg) ← ~ (reg)	The content of the register is complemented.
b)	NOT mem	(mem) ← ~ (mem)	The content of memory is complemented.

Table - 3.12 continued...

S.No.	Instruction	Symbolic representation	Explanation
73. a) b)	SHL reg/mem or SAL reg/mem SHL reg or SAL reg i) SHL reg, 1 or SAL reg, 1 ii) SHL reg, CL or SAL reg, CL SHL mem or SAL mem i) SHL mem, 1 or SAL mem, 1 ii) SHL mem, CL or SAL mem, CL	$CF \leftarrow B_{MSD}$; $B_{n+1} \leftarrow B_n$; $B_{LSD} \leftarrow 0$  reg 8/mem8 MSD LSD CF B7 B6 B5 B4 B3 B2 B1 B0 ← 0  reg16/mem16 MSD LSD CF B15 B14 B13 B2 B1 B0 ← 0	The content of register/memory is shifted left, the MSD is shifted to carry flag while the LSD is filled with zero. For **SHL reg/mem, 1** the content of the register/memory is shifted left once. For **SHL reg/mem, CL**, the number of times the content of register/memory has to be shifted left is specified by a count value (1 to 255_{10}) stored in CL-register.
74. a) b)	SHR reg/mem SHR reg i) SHR reg, 1 ii) SHR reg, CL SHR mem i) SHR mem, 1 ii) SHR mem, CL	$CF \leftarrow B_{LSD}$; $B_n \leftarrow B_{n+1}$; $B_{MSD} \leftarrow 0$  reg 8/mem8 MSD LSD 0 → B7 B6 B5 B4 B3 B2 B1 B0 CF  reg16/mem16 MSD LSD 0 → B15 B14 B13 B2 B1 B0 CF	The content of register/memory is shifted right, the LSD is shifted to carry flag while the MSD is filled with zero. For **SHR reg/mem, 1** the content of register/memory is shifted right once. For **SHR reg/mem, CL**, the number of times the content of register/memory has to be shifted right is specified by a count value (1 to 255_{10}) stored in CL-register.

Note : MSD - Most Significant Digit ; LSD - Least Significant Digit.

Table - 3.12 continued...

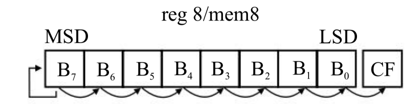
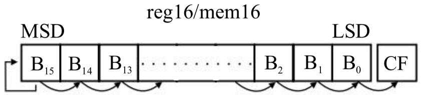
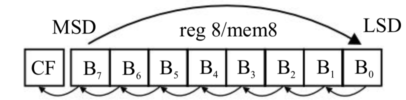
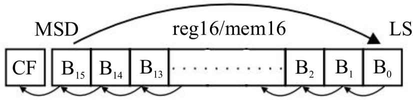
S.No.	Instruction	Symbolic representation	Explanation
75. a)	SAR reg/mem SAR reg i) SAR reg, 1 ii) SAR reg, CL	C← B_{LSD} ; B_n← B_{n+1} ; B_{MSD} ← B_{MSD}  	The content of register/memory is shifted right, the LSD is shifted to carry flag while the MSD is retained. For **SAR reg/mem, 1**, the content of register/ memory is shifted right once. For **SAR reg/mem, CL**, the number of times the content of register/memory has to be shifted right is specified by a count value (0 to 255_{10}) stored in CL-register.
b)	SAR mem i) SAR mem, 1 ii) SAR mem, CL	 	
76. a)	ROL reg/mem ROL reg i) ROL reg, 1 ii) ROL reg, CL	B_{n+1} ← B_n ; CF ← B_{MSD} ; B_{LSD} ← B_{MSD}  	The content of register/memory is rotated left, while the MSD is moved to both LSD and carry flag. For **ROL reg/mem, 1**, the content of register/ memory is rotated left once. For **ROL reg/mem, CL**, the number of times the content of register/memory has to be rotated left is specified by a count value (0 to 255_{10}) stored in CL-register.
b)	ROL mem i) ROL mem, 1 ii) ROL mem, CL	 	

Table - 3.12 continued...

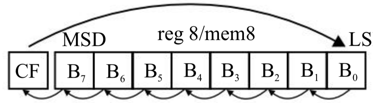
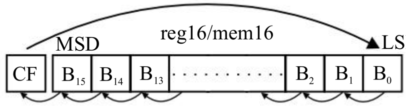
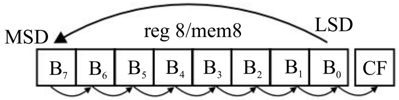
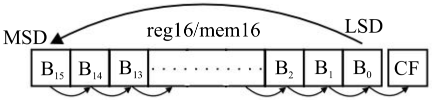
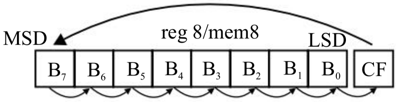
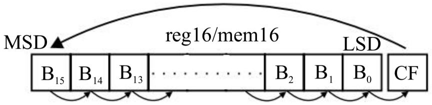
S.No.	Instruction	Symbolic representation	Explanation
77.	RCL reg/mem	$B_{n+1} \leftarrow B_n$; $B_{LSD} \leftarrow CF$; $CF \leftarrow B_{MSD}$	
a)	RCL reg i) RCL reg, 1 ii) RCL reg, CL	 MSD reg 8/mem8 LSD CF B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0	The content of register/memory is rotated left, the carry flag is moved to the LSD, while the MSD is moved to carry flag. For **RCL reg/mem, 1**, the content of register/memory is rotated left once. For **RCL reg/mem, CL**, the number of times the content of register/memory has to be rotated left is specified by a count value (0 to 255_{10}) stored in CL-register.
b)	RCL mem i) RCL mem, 1 ii) RCL mem, CL	 MSD reg16/mem16 LSD CF B_{15} B_{14} B_{13} B_2 B_1 B_0	
78.	ROR reg/mem	$B_n \leftarrow B_{n+1}$; $B_{MSD} \leftarrow CF$; $CF \leftarrow B_{LSD}$	
a)	ROR reg i) ROR reg, 1 ii) ROR reg, CL	 MSD reg 8/mem8 LSD B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0 CF	The content of register/memory is rotated right, the LSD is moved both to MSD and carry flag. For **ROR reg/mem, 1**, the content of register/memory is rotated right once. For **ROR reg/mem, CL**, the number of times the content of register/memory has to be rotated right is specified by a count value (0 to 255_{10}) stored in CL-register.
b)	ROR mem i) ROR mem,1 ii) ROR mem, CL	 MSD reg16/mem16 LSD B_{15} B_{14} B_{13} B_2 B_1 B_0 CF	

Table - 3.12 continued...

S.No.	Instruction	Symbolic representation	Explanation
79.	RCR reg/mem	$B_n \leftarrow B_{n+1}$; $B_{MSD} \leftarrow CF$; $CF \leftarrow B_{LSD}$	
a)	RCR reg i) RCR reg, 1 ii) RCR reg, CL		The content of register/memory is rotated right, the carry flag is moved to MSD while the LSD is moved to carry flag. For **RCR reg/mem, 1**, the content of register/memory is rotated right once. For **RCR reg/mem, CL**, the number of times the content of register/memory has to be rotated right is specified by a count value (0 to 255_{10}) stored in CL-register.
b)	RCR mem i) RCR mem, 1 ii) RCR mem, CL		

3.9 STRING MANIPULATION INSTRUCTIONS

A string is a sequence of bytes or words. The 8086 instruction set includes instructions for string movement, comparison, scan, load and store. It also consists of the REP instruction prefix which is used to repeat the execution of string instructions.

The string instructions end with "S" or "SB" or "SW". Here, "S" represents string, "SB" represents string byte and "SW" represents string word.

All string instructions have an implied source and destination operand (i.e., the operands are not specified as a part of the instruction). The string instructions MOVS and CMPS assume that the source operand is in data segment memory, and the destination is in extra segment memory. The string instruction STOS and SCANS assumes that the source operand is in accumulator, and destination is in extra segment memory. The string instruction LODS assumes that the source operand is in data segment memory and destination is accumulator.

For string operations, the offset or effective address of the source operand is stored in SI-register and that of destination operand is stored in DI-register. On execution of a string instruction depending on **D**irection **F**lag (DF), SI and DI registers are automatically updated to point to the next byte/word of the source and destination. If DF = 0, then SI and DI are incremented by one for byte and incremented by two for word. If DF = 1, then SI and DI are decremented by one for byte and decremented by two for word.

The string instructions of 8086 are listed in Table-3.13, with a brief description about each instruction.

TABLE - 3.13 : STRING MANIPULATION INSTRUCTIONS

S.No.	Instruction	Symbolic representation	Explanation
80.	REP		
a)	REPZ/REPE	While CX ≠ 0 and ZF = 1, repeat execution of string instruction and (CX) ← (CX) – 1	It is a prefix used for compare or scan string instruction. When a string instruction is prefixed with **REPZ/REPE**, the instruction execution is repeated if CX ≠ 0 and ZF=1. After each execution of string instruction, the content of CX is decremented by 1. The repeat operation is terminated if CX = 0 or ZF = 0.
b)	REPNZ/REPNE	While CX ≠ 0 and ZF = 0, repeat execution of string instruction and (CX) ← (CX) – 1	It is a prefix used for compare or scan string instructions. When a string instruction is prefixed with **REPNZ/REPNE**, the instruction execution is repeated if CX ≠ 0 and ZF=0. After each execution of string instruction the content of CX is decremented by 1.The repeat operation is terminated if CX = 0 or ZF = 1.

Table - 3.13 continued...

S.No.	Instruction	Symbolic representation	Explanation
81. a)	MOVS MOVSB	$MA = (DS) \times 16_{10} + (SI)$ $MA_E = (ES) \times 16_{10} + (DI)$ $(MA_E) \leftarrow (MA)$ If DF=0, then $(DI) \leftarrow (DI)+1$; $(SI) \leftarrow (SI)+1$ If DF=1, then $(DI) \leftarrow (DI)-1$; $(SI) \leftarrow (SI)-1$	One byte of a string data stored in data segment is is copied into extra segment, and SI and DI are automatically incremented/decremented by 1 depending on **D**irection **F**lag (DF).
b)	MOVSW	$MA = (DS) \times 16_{10} + (SI)$ $MA_E = (ES) \times 16_{10} + (DI)$ $(MA_E ; MA_E + 1) \leftarrow (MA ; MA + 1)$ If DF = 0, then $(DI) \leftarrow (DI)+2$; $(SI) \leftarrow (SI) + 2$ If DF = 1, then $(DI) \leftarrow (DI)-2$; $(SI) \leftarrow (SI) + 2$	One word of a string data stored in data segment is copied into extra segment, and SI and DI are automatically incremented/decremented by 2 depending on **D**irection **F**lag (DF).
82. a) b)	CMPS CMPSB CMPSW	$MA = (DS) \times 16_{10} + (SI)$ $MA_E = (ES) \times 16_{10} + (DI)$ Modify flags $\leftarrow (MA) - (MA_E)$ If $(MA) > (MA_E)$ then CF = 0 ; ZF = 0 ; SF = 0 If $(MA) < (MA_E)$ then CF = 1 ; ZF = 0 ; SF = 1 If $(MA) = (MA_E)$ then CF = 0 ; ZF = 1 ; SF = 0 For byte operation : If DF=0, then $(DI) \leftarrow (DI)+1$; $(SI) \leftarrow (SI)+1$ If DF=1, then $(DI) \leftarrow (DI)-1$; $(SI) \leftarrow (SI)-1$ For word operation : If DF=0, then $(DI) \leftarrow (DI)+2$; $(SI) \leftarrow (SI)+2$ If DF=1, then $(DI) \leftarrow (DI)-2$; $(SI) \leftarrow (SI)-2$	One byte/word of a string data in extra segment is subtracted from one byte/word of a string data in the data segment and the result is used to modify flags.The contents of DI and SI are automatically incremented/decremented depending on **D**irection **F**lag (DF). For byte operation the contents of DI and SI are incremented/decremented by 1.For word operation, the contents of DI and SI are incremented/ decremented by 2.

Table - 3.13 continued...

S.No.	Instruction	Symbolic representation	Explanation
83. a)	SCAS SCASB	$MA_E = (ES) \times 16_{10} + (DI)$ Modify flags $\leftarrow$ $(AL) - (MA_E)$ If $(AL) > (MA_E)$ then CF = 0 ; ZF = 0 ; SF = 0 If $(AL) < (MA_E)$ then CF = 1 ; ZF = 0 ; SF = 1 If $(AL) = (MA_E)$ then CF = 0 ; ZF = 1 ; SF = 0 If DF = 0 then, $(DI) \leftarrow (DI) + 1$ If DF = 1 then, $(DI) \leftarrow (DI) - 1$	One byte of string data in extra segment is subtracted from the content of AL and the result is used to modify flags.The content of DI and SI are automatically incremented/decremented by 1 depending on **D**irection **F**lag (DF).
b)	SCASW	$MA_E = (ES) \times 16_{10} + (DI)$ Modify flags $\leftarrow$ $(AX) - (MA_E ; MA_E + 1)$ If $(AX) > (MA_E ; MA_E + 1)$ then CF = 0 ; ZF = 0 ; SF = 0 If $(AX) < (MA_E ; MA_E + 1)$ then CF = 1 ; ZF = 0 ; SF = 1 If $(AX) = (MA_E ; MA_E + 1)$ then CF = 0 ; ZF = 1 ; SF = 0 If DF = 0 then, $(DI) \leftarrow (DI) + 2$ If DF = 1 then, $(DI) \leftarrow (DI) - 2$	One word of string data in extra segment is subtracted from the content of AX and the result is used to modify flags.The content of DI and SI are automatically incremented/decremented by 2 depending on **D**irection **F**lag (DF).
84. a)	LODS LODSB	$MA = (DS) \times 16_{10} + (SI)$ $(AL) \leftarrow (MA)$ If DF = 0 then, $(SI) \leftarrow (SI) + 1$ If DF = 1 then, $(SI) \leftarrow (SI) - 1$	One byte of a string data stored in data segment is copied into the AL-register and SI is automatically incremented/decremented by 1, depending on **D**irection **F**lag (DF).
b)	LODSW	$MA = (DS) \times 16_{10} + (SI)$ $(AX) \leftarrow (MA ; MA + 1)$ If DF = 0 then, $(SI) \leftarrow (SI) + 2$ If DF = 1 then, $(SI) \leftarrow (SI) - 2$	One word of a string data stored in data segment is copied into the accumulator. SI is automatically incremented/decremented by 2, depending on **D**irection **F**lag (DF).

Table - 3.13 continued...

S.No.	Instruction	Symbolic representation	Explanation
85. a)	STOS STOSB	$MA_E = (ES) \times 16_{10} + (DI)$ $(MA_E) \leftarrow (AL)$ If DF=0 then, $(DI) \leftarrow (DI) + 1$ If DF=1 then, $(DI) \leftarrow (DI) - 1$	The content of AL-register is stored as one-byte of string data in the extra segment. DI is automatically incremented/decremented by 1 depending on **D**irection **F**lag (DF).
b)	STOSW	$MA_E = (ES) \times 16_{10} + (DI)$ $(MA_E ; MA_E + 1) \leftarrow (AX)$ If DF=0 then, $(DI) \leftarrow (DI) + 2$ If DF=1 then, $(DI) \leftarrow (DI) - 2$	The content of AX-register is stored as one word of string data in the extra segment. DI is automatically incremented/decremented by 2 depending on **D**irection **F**lag (DF).

3.10 CONTROL TRANSFER INSTRUCTIONS

The control transfer group consists of call, jump, loop and software interrupt instructions. Normally, a program is executed sequentially (i.e., the program instructions are executed one after the other). When a branch instruction is encountered, the program execution control is transferred to the specified destination or target instruction. The transfer of program execution control is done either by changing the content of IP or by changing the content of IP and CS. When the content of IP alone is modified, the program control branches to a new memory location in the same segment. When the content of both IP and CS are modified, the program control branches to a new memory location in another memory segment.

The control transfer instructions do not affect the flags of 8086. The jump and loop instructions can be classified into conditional and unconditional instructions. In conditional instructions, the status of one or more flags are checked and control transfer takes place only if the specified condition is satisfied.

The control transfer instructions are listed in Table-3.14 to Table-3.19, with a brief description about each instruction.

3.10.1 CALL and RET Instructions

The CALL instructions transfer control to a subprogram or subroutine or a procedure after saving return address in the stack memory. There are two types of call instructions : Intrasegment or near call and Intersegment or far call. A near call refers to calling a procedure stored in the same code segment memory in which the main program (or calling program) resides. A far call refers to calling a procedure stored in a different code segment memory than that of main program.

While executing near call, the content of IP alone is pushed to stack. While executing far call, the content of CS and IP are pushed to stack. Every procedure or subroutine ends with an RET instruction. The execution of RET instruction at the end of subroutine or procedure, will pop the content of top of stack to IP in case of near call or to IP and CS in case of far call. Thus, the program control return back to main program.

The call and return instructions are listed in Table-3.14, with a brief description about each instruction.

TABLE - 3.14 : CALL AND RET INSTRUCTIONS

S.No.	Instruction	Symbolic representation	Explanation
86.	CALL disp16 (Call near - direct within segment)	(SP) ← (SP) – 2 $MA_S = (SS) \times 16_{10} + (SP)$ $(MA_S) \leftarrow (IP)$ (IP) ← disp16	This instruction is near-direct call in which the program control is transferred within the same segment. The stack pointer is decremented by 2, the **I**nstruction **P**ointer (IP) is pushed into stack and the effective address (disp16) of the subroutine/procedure to be executed is loaded in IP.

Table - 3.14 continued...

S.No.	Instruction	Symbolic representation	Explanation
87.	CALL reg/mem (Call near - indirect within segment)		
a)	CALL reg	$(SP) \leftarrow (SP) - 2$ $MA_S = (SS) \times 16_{10} + (SP)$ $(MA_S) \leftarrow (IP)$ $(IP) \leftarrow (reg)$	This instruction is near-indirect call in which the control transfer is within same segment and the effective address of subroutine/procedure to be called is stored in register/memory. The stack pointer is decremented by 2, the **I**nstruction **P**ointer (IP) is pushed into the stack and the effective address of the subroutine/procedure to be executed is loaded in IP from the register/memory.
b)	CALL mem	$(SP) \leftarrow (SP) - 2$ $MA_S = (SS) \times 16_{10} + (SP)$ $(MA_S) \leftarrow (IP)$ $(IP) \leftarrow (mem)$	
88.	CALL $addr_{offset}$, $addr_{base}$ (Call far - direct intersegment)	$(SP) \leftarrow (SP) - 2$ $MA_S = (SS) \times 16_{10} + (SP)$ $(MA_S) \leftarrow (IP)$ $(IP) \leftarrow addr_{offset}$ $(SP) \leftarrow (SP) - 2$ $MA_S = (SS) \times 16_{10} + (SP)$ $(MA_S) \leftarrow (CS)$ $(CS) \leftarrow addr_{base}$	This instruction is far-direct call in which the program control is transferred to another segment. The offset and segment base address of the procedure to be executed are directly given in the instruction. The stack pointer is decremented by 2, the **I**nstruction **P**ointer (IP) is pushed into stack. The offset address of the procedure to be executed is loaded in IP. The stack pointer is again decremented by 2 and CS is pushed into stack and the base address of the procedure to be executed is loaded in CS.

Table - 3.14 continued...

S.No.	Instruction	Symbolic representation	Explanation
89.	CALL mem (Call far - indirect intersegment)	$(SP) \leftarrow (SP) - 2$ $MA_S = (SS) \times 16_{10} + (SP)$ $(MA_S) \leftarrow (IP)$ $(IP) \leftarrow (mem)_{(\text{offset address})}$ $(SP) \leftarrow (SP) - 2$ $MA_S = (SS) \times 16_{10} + (SP)$ $(MA_S) \leftarrow (CS)$ $(CS) \leftarrow (mem+2)_{(\text{base address})}$	This instruction is far-indirect call in which the program control is transferred to different segment, and the offset and segment base address of procedure to be executed are stored in memory. The stack pointer is decremented by 2 and IP is pushed to stack. The offset address available in memory is moved to IP. The stack pointer is again decremented by 2, and CS is pushed to stack. The base address available in memory is moved to CS.
90.	RET (Return from call within segment)	$MA_S = (SS) \times 16_{10} + (SP)$ $(IP) \leftarrow (MA_S)$ $(SP) \leftarrow (SP) + 2$	Return the control back to calling procedure from the called procedure within the segment. The content of top of stack is transferred to IP. The stack pointer is incremented by 2.
91.	RET data16 (Return from call within segment adding immediate value to SP).	$MA_S = (SS) \times 16_{10} + (SP)$ $(SP) \leftarrow (SP) + data16$	Return the control back to calling procedure from the called procedure within the segment. The content of top of stack is transferred to IP and the stack pointer is incremented by a value (data16) specified in the instruction.
92.	RET (Return from intersegment call)	$MA_S = (SS) \times 16_{10} + (SP)$ $(IP) \leftarrow (MA_S)$ $(SP) \leftarrow (SP) + 2$ $MA_S = (SS) \times 16_{10} + (SP)$ $(CS) \leftarrow (MA_S)$ $(SP) \leftarrow (SP) + 2$	Return the control back to calling procedure from the called procedure which is in different segment. The content of top of the stack is moved to IP, and stack pointer is incremented by 2. Next the content of current top of stack is moved to CS and SP is incremented by 2.

Table - 3.14 continued...

S.No.	Instruction	Symbolic representation	Explanation
93.	RET data16 (Return from intersegment call adding immediate data to SP)	$MA_S = (SS) \times 16_{10} + (SP)$ $(IP) \leftarrow (MA_S)$ $(SP) \leftarrow (SP) + 2$ $MA_S = (SS) \times 16_{10} + (SP)$ $(CS) \leftarrow (MA_S)$ $(SP) \leftarrow (SP) + data16$	Return the control back to calling procedure from the called procedure which is in different segment. The content of top of stack is moved to IP and stack pointer is incremented by 2. Next the content of current top of stack is moved to CS and the stack pointer is incremented by a value (data16) specified in the instruction.

3.10.2 Unconditional Jump Instructions

The unconditional jump instructions does not check for any flag condition. When the unconditional jump instruction is executed the program control is transferred to new memory location either in same segment or in another segment. In near jump instruction the program control is transferred to new memory location in the same segment by modifying the content of **I**nstruction **P**ointer (IP). In far jump instruction the program control is transferred to new memory location in another segment by modifying the content of **I**nstruction **P**ointer (IP) and **C**ode **S**egment (CS) register.

TABLE - 3.15 : NEAR JUMP INSTRUCTIONS

S.No.	Instruction	Symbolic representation	Explanation
94.	JMP disp16	$(IP) \leftarrow (IP) + disp16$	The 16-bit value (disp16) given in the instruction is added to **I**nstruction **P**ointer (IP).
95.	JMP disp8	$disp16 \xleftarrow[\text{extend}]{\text{Sign}} disp8$ $(IP) \leftarrow (IP) + disp16$	The 8-bit value (disp8) given in the instruction is sign extended to 16-bit and added to **I**nstruction **P**ointer (IP).

Table - 3.15 continued...

S.No.	Instruction	Symbolic representation	Explanation
96.	JMP reg/mem		
a)	JMP reg	(IP) ← (IP) + (reg)	The 16-bit value stored in the register/memory is added to **I**nstruction **P**ointer (IP).
b)	JMP mem	(IP) ← (IP) + (mem)	

TABLE - 3.16 : FAR JUMP INSTRUCTIONS

S.No.	Instruction	Symbolic representation	Explanation
97.	JMP $\text{addr}_{\text{offset}}$, $\text{addr}_{\text{base}}$	(IP) ← $\text{addr}_{\text{offset}}$ (CS) ← $\text{addr}_{\text{base}}$	The offset address given in the instruction is loaded in IP and the base address given in the instruction is loaded in CS-register.
98.	JMP mem	(IP) ← (mem) (CS) ← (mem+2)	The content of (16-bit) memory is moved to IP and the next word in the memory is moved to CS-register.

3.10.3 Conditional Jump Instructions

In a conditional jump instruction, one or more flag conditions are checked. If the conditions are TRUE, then the program control is transferred to new memory location in the same segment by modifying the content of the IP. All conditional instructions are only near jump (or short jump), hence the content of CS is not altered.

In all conditional jump instructions, an 8-bit value(disp8) will be directly specified in the instruction which is sign extended to 16-bit and added to IP. The new value in IP is the effective address of the instruction where the program control is transferred, if the condition is TRUE.

Instruction Format

J <condition> disp8

If <condition> is TRUE then, $\text{disp16} \xleftarrow{\text{sign extend}} \text{disp8}$; $(\text{IP}) \leftarrow (\text{IP}) + \text{disp16}$

Note : *If the condition specified by the instruction is FALSE then the content of IP is not altered.*

TABLE - 3.17 : CONDITIONAL JUMP INSTRUCTIONS

S.No.	Instruction	Explanation
99.	JE disp8 (JZ disp8)	Jump if ZF=1
100.	JL disp8 (JNGE disp8)	Jump if SF≠OF
101.	JLE disp8 (JNG disp8)	Jump if SF≠OF or ZF = 1
102.	JB disp8 (JNAE/JC disp8)	Jump if CF=1
103.	JBE disp8 (JNA disp8)	Jump if CF=1 or ZF=1
104.	JP disp8 (JPE disp8)	Jump if PF=1
105.	JNB disp8 (JAE/JNC disp8)	Jump if CF=0
106.	JNBE disp8 (JA disp8)	Jump if CF=0 and ZF=0
107.	JNP disp8 (JPO disp8)	Jump if PF=0
108.	JNO disp8	Jump if OF=0
109.	JNS disp8	Jump if SF=0
110.	JO disp8	Jump if OF=1
111.	JS disp8	Jump if SF=1
112.	JNE disp8 (JNZ disp8)	Jump if ZF=0
113.	JNL disp8 (JGE disp8)	Jump if SF=OF
114.	JNLE disp8 (JG disp8)	Jump if CF=OF and ZF=0
115.	JCXZ disp8	Jump if (CX) = 0

Note : *The execution of instruction JCXZ is similar to that of any conditional jump instruction except that the content of CX-register is checked to make a decision instead of a flag.*

3.10.4 LOOP Instructions

LOOP instructions are used to execute a group of instructions, a number of times, as specified by a count value stored in CX-register. The number of instructions to be looped will be specified directly in the instruction as a signed eight bit number (displacement or disp8). For positive displacement the instructions below the LOOP instruction are executed and for negative displacement the instructions above the LOOP instructions are executed. The content of CX-register is decremented by one after each execution of looped instructions.The effective address of the first instruction of the loop is obtained by sign extending the disp8 to 16-bit and adding it to IP.

TABLE - 3.18 : LOOP INSTRUCTIONS

S.No.	Instruction	Symbolic representation	Explanation
116.	LOOP disp8	Loop if (CX) ≠ 0 (CX) ← (CX) – 1	Repeat execution of the group of instructions until the content of CX is zero. After each execution CX is decremented by one.
117.	LOOPZ disp8 (LOOPE disp8)	Loop if (CX)≠0 and ZF=1 (CX) ← (CX) – 1	Repeat execution of the group of instructions, if the content of CX is not zero and the ZF=1. After each execution CX is decremented by one.
118.	LOOPNZ disp8 (LOOPNE disp8)	Loop if (CX)≠0 and ZF=0 (CX) ← (CX) – 1	Repeat execution of the group of instructions, if the content of CX is not zero and the ZF=0. After each execution, the CX is decremented by one.

3.10.5 Software Interrupts

The INT instructions are called software interrupts. The INT instruction is used to call a procedure or subroutine on interrupt basis. Hence, the procedure executed on interrupt basis is called **I**nterrupt **S**ervice **R**outine (ISR).

The INT instruction is accompanied by a type number, which can be in the range of 0 to 255. Therefore, in an 8086 processor, 256 types of software interrupts can be implemented. These software interrupts are used to implement the system call service of the operating system.

In order to execute an ISR, a 16-bit effective address for IP and a 16-bit base address for CS are needed. Therefore, for each INT instruction four memory locations are reserved in the first 1k address space of memory. In the reserved locations, the first two locations are used to store the effective address (to be loaded in IP), and the next two locations are used to store the base address, (to be loaded in CS-register).

The address of the reserved memory location is called Vector address. The vector address of an interrupt is obtained by multiplying the type number by 4.

Before executing ISR the content of IP, CS and flag register are pushed to stack. Each ISR is terminated by IRET (**I**nterrupt **ret**urn) instruction. On executing IRET instruction the top of stack are poped to IP, CS and flag register. Thus, the program control return back to main program after executing ISR.

TABLE - 3.19 : SOFTWARE INTERRUPT

S.No.	Instruction	Symbolic representation	Explanation
119.	INT type	$(SP) \leftarrow (SP) - 2\ ;\ (MA_S) \leftarrow$ Flags $IF \leftarrow 0\ ;\ TF \leftarrow 0$ $(SP) \leftarrow (SP) - 2\ ;\ (MA_S) \leftarrow (CS)$ $(SP) \leftarrow (SP) - 2\ ;\ (MA_S) \leftarrow (IP)$ $(IP) \leftarrow (0000 : (type \times 4))$ $(CS) \leftarrow (0000 : (type \times 4) + 2)$ For each push operation the stack memory address is calculated as shown below. $MA_S = (SS) \times 16_{10} + (SP)$	This instruction is a software interrupt and used to call a service procedure(or subroutine) on interrupt basis. The type number is from 0 to 255. On execution of this instruction, the content of flag register, CS-register and IP are pushed to stack one by one after decrementing SP by 2 before each push operation. The flags TF and IF are also cleared. The effective vector address is calculated by multiplying the type number by 4. The memory location pointed by vector address contain the address of interrupt service routine. The first word pointed by the calculated vector address is moved to IP and the next word is moved to CS-register.

Table - 3.19 continued ....

S.No.	Instruction	Symbolic representation	Explanation
120.	INT 3	(SP) ← (SP) – 2 ; (MA_S) ← Flags IF ← 0 ; TF ← 0 (SP)← (SP) – 2 ; (MA_S) ← (CS) (SP) ← (SP) – 2 ; (MA_S) ← (IP) (IP) ← ($0000C_H$) ; (CS) ← ($0000E_H$) ***Note :*** $3 \times 4 = 12_{10} = 0C_H ; 12 + 2 = 14_{10} = 0E_H$ For each push operation the MA_S is given by $MA_S = (SS) \times 16_{10} + (SP)$	This instruction is a special type of software interrupt which has the single byte code of CC_H. Many systems use this as a break point instruction. The operations performed by this instruction is same as that of a type-3 interrupt.
121.	INTO	If OF=1, then following operations are performed. (SP) ← (SP) – 2 ; (MA_S) ← Flags IF← 0 ; TF ← 0 (SP) ← (SP) – 2 ; (MA_S) ← (CS) (SP) ← (SP) – 2 ; (MA_S) ← (IP) (IP) ←(00010_H) ; (CS) ← (00012_H) ***Note :*** $4 \times 4 = 16_{10} = 10_H ; 16 + 2 = 18_{10} = 12_H$ For each push operation the MA_S is given by $MA_S = (SS) \times 16_{10} + (SP)$	If **O**verflow **F**lag (OF) is 1, then a type-4 interrupt is performed.
122.	IRET	(IP) ← (MA_S) ; (SP) ← (SP) + 2 (CS) ← (MA_S) ; (SP) ← (SP) + 2 Flag ← (MA_S) ; (SP)← (SP) + 2 For each pop operation the stack memory address is calculated as shown below: $MA_S = (SS) \times 16_{10} + (SP)$	This instruction is used to terminate an interrupt service procedure and transfer the program control back to the main program. On execution of this instruction, the contents of top of stack (pointed by SP) are moved (poped) to IP, CS and flag registers one by one. After every pop operation, the SP is incremented by 2.

3.11 PROCESSOR CONTROL INSTRUCTIONS

The processor control group includes instructions to set or clear carry flag, direction flag and interrupt flag. It also includes HLT, NOP, LOCK and ESC instructions which controls the processor operation.

The processor control instructions are listed in Table-3.20, with a brief description about each instruction.

TABLE - 3.20 : PROCESSOR CONTROL INSTRUCTIONS

S.No.	Instruction	Symbolic representation	Explanation
123.	CLC	CF ← 0	The carry flag is reset to zero.
124.	CMC	CF ← ~ CF	The carry flag is complemented.
125.	STC	CF ← 1	The carry flag is set to one.
126.	CLD	DF ← 0	The direction flag is reset to zero.
127.	STD	DF ← 1	The direction flag is set to one.
128.	CLI	IF ← 0	The interrupt flag is reset to zero.
129.	STI	IF ← 1	The interrupt flag is set to one.
130.	HLT	Halt program execution	This instruction is used to terminate a program. On execution of this instruction the processor enter into an idle state and performs no-operation until an interrupt occurs.
131.	WAIT	Wait for test line active.	This instruction causes the processor to enter into an idle state or a wait state and continue to remain in that state until a signal is asserted on TEST input pin or until a valid interrupt signal is received on the INTR or NMI interrupt input pin.

Table - 3.20 : continued ....

S.No.	Instruction	Explanation
132. a) b)	ESC opcode, mem/reg ESC opcode, mem ESC opcode, reg	This instruction is used to pass instructions to a coprocessor which shares the address and data bus with the 8086. The lower 6 bits of the opcode is the opcode of 8087 instruction and the upper two bits are zeros. For **ESC opcode, mem** the data is accessed by 8087 from memory and for **ESC opcode, reg** the data is accessed by 8087 from the 8086 register specified in the instruction.
133.	LOCK	The **LOCK** is used as a prefix to a critical instruction which has to be executed without any disturbance from other bus masters. When **LOCK** prefix is used in an instruction then during execution of this instruction the **LOCK** prefix ensures that the shared system resources are not taken over by other bus masters in the middle of instruction execution.
134.	NOP	**N**o **op**eration is performed for 3 clock periods, when this instruction is executed. The processor waits for 3 clock periods and then the next instruction is executed.

Symbols/Abbreviations used in Instruction Set

reg, reg1, reg2	- 8-bit or 16-bit register	Flags	- Flag register	OF	- Overflow Flag
reg8	- 8-bit register	&	- logical AND	TF	- Trace Flag
reg16	- 16-bit register	|	- logical-OR	IF	- Interrupt Flag
mem	- 8-bit or 16-bit memory	^	- logical Exclusive-OR	DF	- Direction Flag
mem8	- 8-bit memory	~	- logical NOT	SF	- Sign Flag
mem16	- 16-bit memory	disp	- 8-bit or 16-bit displacement	ZF	- Zero Flag
data	- 8-bit or 16-bit immediate data	disp8	- 8-bit displacement	PF	- Parity Flag
data8	- 8-bit immediate data	disp16	- 16-bit displacement	addr8	- 8-bit port address
segreg	- segment register (Excluding CS)	CF	- Carry Flag	$addr_{offset}$	- 16-bit offset/effective address
data16	- 16-bit immediate data	AF	- Auxiliary Carry Flag	$addr_{base}$	- 16-bit base address

3.12 EXAMPLES OF 8086 INSTRUCTIONS

Note : 1. *The register or register + constant enclosed by square brackets in the operand field of instructions refer to the method of effective address calculation of memory. The 16-bit constant enclosed by square brackets in the operand field of instructions refer to the effective address of memory data. The 8-bit/16-bit constants which are not enclosed by brackets in the operand field refer to immediate data.*

2. *The term MA used in the symbolic description of instructions refer to physical memory address of data segment memory and MA_s refer to physical memory address of stack segment memory and MA_E refer to physical memory address of extra segment memory.*

3. *The register/memory enclosed by brackets in symbolic description refers to the content of register/memory.*

4. *For hexa-decimal constant (data/address) the letter H is included at the end of 8-bit/16-bit constants (data/address). When a hexa-decimal constant start with A, B, C, D, E or F, a zero should be placed before the constant, otherwise the assembler will treat the constant as a variable.*

TABLE - 3.21

Instruction	Symbolic description	Explanation
MOV AX, SI	(AX) ← (SI)	The content of SI-register is moved to AX-register.
MOV CH, CL	(CH) ← (CL)	The content of CL-register is moved to CH-register.
MOV [BX + 08H], AX	$0008_H \xleftarrow{\text{Sign extend}} 08_H$ $MA = (DS) \times 16_{10} + (BX) + 0008_H$ $(MA) \leftarrow (AL)$; $(MA+1) \leftarrow (AH)$	The contents of AX-register is moved to two consecutive memory locations.
MOV AX, [BP + SI + 07H]	$0007_H \xleftarrow{\text{Sign extend}} 07_H$ $MA_S = (SS) \times 16_{10} + (BP) + (SI) + 0007_H$ $(AL) \leftarrow (MA_S)$; $(AH) \leftarrow (MA_S+1)$	The contents of two consecutive memory locations from stack memory are moved to AX-register.
MOV CX, 150AH	$(CX) \leftarrow 150A_H$	The 16-bit data ($150A_H$) given in the instruction is moved to CX-register.
MOV CX, [150AH]	$MA = (DS) \times 16_{10} + 150A_H$ $(CL) \leftarrow (MA)$; $(CH) \leftarrow (MA + 1)$	The contents of two consecutive memory locations addressed by the instruction are moved to CX-register.

Table - 3.21 : continued ....

Instruction	Symbolic description	Explanation
MOV DL, 3CH	$(DL) \leftarrow 3C_H$	The 8-bit data ($3C_H$) given in the instruction is moved to DL-register.
MOV [BX], 9AH	$MA = (DS) \times 16_{10} + (BX)$ $(MA) \leftarrow 9A_H$	The 8-bit data ($9A_H$) given in the instruction is moved to memory.
MOV [SI+0A0H], 0C001H	$FFA0_H \xleftarrow{\text{Sign extend}} A0_H$ $MA = (DS) \times 16_{10} + (SI) + FFA0_H$ $(MA) \leftarrow 01_H\ ;\ (MA+1) \leftarrow C0_H$	The 16-bit data ($C001_H$) given in the instruction is moved to two consecutive memory locations.
MOV ES, CX	$(ES) \leftarrow (CX)$	The content of CX-register is moved to ES-register.
MOV ES, [SI + 0008H]	$MA = (DS) \times 16_{10} + (SI) + 0008_H$ $(ES) \leftarrow (MA\ ;\ MA+1)$	The contents of two consecutive memory locations are moved to ES-register.
MOV DX, SS	$(DX) \leftarrow (SS)$	The content of SS-register is moved to DX-register.
MOV [BX + 0C0H], SS	$FFC0_H \xleftarrow{\text{Sign extend}} C0_H$ $MA = (DS) \times 16_{10} + (BX) + FFC0_H$ $(MA\ ;\ MA+1) \leftarrow (SS)$	The content of SS-register is moved to two consecutive memory locations.
PUSH CX	$(SP) \leftarrow (SP) - 2$ $MA_S = (SS) \times 16_{10} + (SP)$ $(MA_S\ ;\ MA_S+1) \leftarrow (CX)$	The content of CX-register is pushed to top of stack. (The content of CX-register is moved to two consecutive locations in stack memory, whose effective address is obtained by decrementing SP by two.)

Table - 3.21 : continued ....

Instruction	Symbolic description	Explanation
PUSH [BX + 05H]	$0005_H \xleftarrow{\text{Sign extend}} 05_H$ $MA = (DS) \times 16_{10} + (BX) + 0005_H$ $(SP) \leftarrow (SP) - 2$ $MA_S = (SS) \times 16_{10} + (SP)$ $(MA_S) \leftarrow (MA)$; $(MA_S+1) \leftarrow (MA+1)$	The content of data memory specified by the instruction is pushed to top of stack memory. (The content of two consecutive memory locations in data memory are moved to two consecutive locations in stack memory. The effective address of stack memory is obtained by decrementing SP by two.)
POP BX	$MA_S = (SS) \times 16_{10} + (SP)$ $(BX) \leftarrow (MA_S ; MA_S + 1)$ $(SP) \leftarrow (SP) + 2$	The content of top of stack is moved to BX-register. (The content of two consecutive locations in stack memory are moved to BX-register.) After this move operation SP is incremented by two.
POP [SI + 05H]	$0005_H \xleftarrow{\text{Sign extend}} 05_H$ $MA = (DS) \times 16_{10} + (SI) + 0005_H$ $MA_S = (SS) \times 16_{10} + (SP)$ $(MA) \leftarrow (MA_S)$; $(MA+1) \leftarrow (MA_S+1)$ $(SP) \leftarrow (SP) + 2$	The content of top of stack memory is moved to data memory specified by the instruction. (The content of two consecutive locations in stack memory are moved to two consecutive locations in data memory.) Then SP is incremented by two.
XCHG CX, SI	$(CX) \leftrightarrow (SI)$	The content of SI-register is exchanged with the content of CX-register.
XCHG DH, CL	$(DH) \leftrightarrow (CL)$	The content of CL-register is exchanged with the content of DH-register.
XCHG [DI + 07H], DX	$0007_H \xleftarrow{\text{Sign extend}} 07_H$ $MA = (DS) \times 16_{10} + (DI) + 0007_H$ $(MA ; MA + 1) \leftrightarrow (DX)$	The content of DX-register is exchanged with the content of memory.

Table - 3.21 : continued ....

Instruction	Symbolic description	Explanation
IN AX, [DX]	$PORT_{addr} = (DX)$ $(AX) \leftarrow (PORT)$	The content of port, (whose address is in DX-register) is moved to AX-register.
IN AX, 0C0H	$PORT_{addr} = C0_H$ $(AX) \leftarrow (PORT)$	The content of port, (whose address is specified in the instruction) is moved to AX-register.
OUT [DX], AL	$PORT_{addr} = (DX)$ $(PORT) \leftarrow (AL)$	The content of AL-register is moved to port addressed by DX-register.
OUT 0F2H, AX	$PORT_{addr} = F2_H$ $(PORT) \leftarrow (AX)$	The content of AX-register is moved to port whose address is specified in the instruction.
LEA CX, [BX + DI]	$EA = (BX) + (DI)$ $(CX) \leftarrow EA$ *Note : EA - Effective Address*	The instruction LEA determines the **E**ffective **A**ddress (EA) of source operand in memory and load the effective address in CX-register.
LDS BX, [420AH]	$MA = (DS) \times 16_{10} + 420A_H$ $(BX) \leftarrow (MA\,;MA+1)$ $(DS) \leftarrow (MA+2\,;MA+3)$	This instruction copies a word from memory to BX-register and copies the next word in memory to DS-register.
LES BP, [0C00FH]	$MA = (DS) \times 16_{10} + C00F_H$ $(BP) \leftarrow (MA\,;MA+1)$ $(ES) \leftarrow (MA+2\,;MA+3)$	This instruction copies a word from memory to BP-register and copies the next word in memory to ES-register.
ADD CX, DX	$(CX) \leftarrow (CX) + (DX)$	The contents of CX and DX registers are added and the result is stored in CX-register.

Table - 3.21 : continued ....

Instruction	Symbolic description	Explanation
ADC BH, AL	$(BH) \leftarrow (BH) + (AL) + CF$	The content of BH-register, AL-register and the carry flag are added. The result is stored in BH-register.
ADD CX, [BX + 05H]	$0005_H \xleftarrow{\text{Sign extend}} 05_H$ $MA = (DS) \times 16_{10} + (BX) + 0005_H$ $(CX) \leftarrow (CX) + (MA\,;MA+1)$	The content of CX-register and a word from memory are added. The result is stored in CX-register.
ADC [DI], C2H	$MA = (DS) \times 16_{10} + (DI)$ $(MA) \leftarrow (MA) + C2_H + CF$	The 8-bit data ($C2_H$) given in the instruction and the carry flag are added to memory. The result is stored in memory.
ADD DX, 0FA5H	$(DX) \leftarrow (DX) + 0FA5_H$	The 16-bit data ($0FA5_H$) given in the instruction is added to the content of DX-register. The result is stored in DX-register.
SUB DI, SI	$(DI) \leftarrow (DI) - (SI)$	The content of SI-register is subtracted from DI-register. The result is stored in DI-register.
SUB [BP+DI], AH	$MA_S = (SS) \times 16_{10} + (BP) + (DI)$ $(MA_S) \leftarrow (MA_S) - (AH)$	The content of AH-register is subtracted from the content of a location in stack memory. The result is stored in stack memory location.
SUB SP, 0500H	$(SP) \leftarrow (SP) - 0500_H$	The 16-bit data (0500_H) given in the instruction is subtracted from SP-register. The result is stored in SP-register.
SBB AX, DI	$(AX) \leftarrow (AX) - (DI) - CF$	The content of DI-register and carry flag are subtracted from the content of AX-register. The result is stored in AX-register.

Table - 3.21 : continued ....

Instruction	Symbolic description	Explanation
SBB [BX + 08H], DL	$0008_H \xleftarrow{\text{Sign extend}} 08_H$ $MA = (DS) \times 16_{10} + (BX) + 0008_H$ $(MA) \leftarrow (MA) - (DL) - CF$	The content of DL-register and carry flag are subtracted from memory. The result is stored in memory.
MUL BX	$(DX)\ (AX) \leftarrow (AX) \times (BX)$	The content of AX and BX registers are multiplied. The lower word of the result is stored in AX-register and the upper word of the result is stored in DX-register.
MUL DL	$(AX) \leftarrow (AL) \times (DL)$	The content of AL and DL registers are multiplied. The 16-bit result is stored in AX-register.
MUL [BX + 08H]	$0008_H \xleftarrow{\text{Sign extend}} 08_H$ $MA = (DS) \times 16_{10} + (BX) + 0008_H$ $(AX) \leftarrow (AL) \times (MA)$ (or) $(DX)\ (AX) \leftarrow (AX) \times (MA\ ;\ MA + 1)$	In 8-bit multiplication the content of AL and 8-bit memory are multiplied. The result is stored in AX-register. In 16-bit multiplication the content of AX and 16-bit memory are multiplied. The result is stored in AX and DX registers. The 8-bit or 16-bit multiplication is defined by w-bit in the instruction template.
DIV CH	$\underset{\text{Quotient}}{(AL)} \leftarrow (AX) \div (CH)$ $\underset{\text{Remainder}}{(AH)} \leftarrow (AX)\ MOD\ (CH)$	The content of AX-register is divided by the content of CH-register. The quotient is stored in AL-register and the remainder in AH-register.
DIV BX	$\underset{\text{Quotient}}{(AX)} \leftarrow (DX)\ (AX) \div (BX)$ $\underset{\text{Remainder}}{(DX)} \leftarrow (DX)\ (AX)\ MOD\ (BX)$	The content of AX and DX registers are divided by the content of BX-register. The quotient is stored in AX-register and the remainder in DX-register.

Table - 3.21 : continued ....

Instruction	Symbolic description	Explanation
DIV [SI + 0C002H]	MA = (DS) × 16_{10} + (SI) + $C002_H$ 16-bit ÷ 8-bit (AL) ← (AX) ÷ (MA) Quotient AH ← (AX) MOD (MA) Remainder 32-bit ÷16-bit AX ← (DX) (AX) ÷ (MA ; MA + 1) Quotient DX ← (DX) (AX) MOD (MA ; MA + 1) Remainder	In 16-bit by 8-bit division the content of AX-register is divided by 8-bit memory. The quotient is stored in AL-register and the remainder is stored in AH-register. In 32-bit by 16-bit division, the contents of AX and DX register are divided by 16-bit memory. The quotient is stored in AX-register and the remainder is stored in DX-register. The 8-bit or 16-bit divisor is defined by w-bit in the instruction template.
INC DL	(DL) ← (DL) + 1	The content of DL-register is incremented by one.
INC DX	(DX) ← (DX) + 1	The content of DX-register is incremented by one.
INC [BP + SI + 0F5H]	$FFF5_H \xleftarrow{\text{Sign extend}} F5_H$ MA_S = (SS) × 16_{10} + (BP) + (SI) + $FFF5_H$ (MA_S) ← (MA_S) + 1 (or) (MA_S ; MA_S + 1) ← (MA_S ; MA_S + 1) + 1	In 8-bit operation the content of 8-bit stack memory is incremented by one. In 16-bit operation the content of 16-bit stack memory is incremented by one. The 8-bit or 16-bit operation is defined by w-bit in the instruction template.
DEC CH	(CH) ← (CH) – 1	The content of CH-register is decremented by one.
DEC BP	(BP) ← (BP) – 1	The content of BP-register is decremented by one.
DEC [DI + 0007H]	MA = (DS) × 16_{10} + (DI) + 0007_H (MA) ← (MA) – 1 (or) (MA ; MA + 1) ← (MA ; MA + 1) – 1	In 8-bit operation, the content of 8-bit memory is decremented by one. In 16-bit operation the content of 16-bit memory is decremented by one. The 8-bit or 16-bit operation is defined by w-bit in the instruction template.

Table - 3.21 : continued ....

Instruction	Symbolic description	Explanation
CMP DL, CH	Modify flags ← (DL) – (CH) If (DL) = (CH) ; then CF = 0, SF = 0, ZF = 1 If (DL) < (CH) ; then CF = 1, SF = 1, ZF = 0 If (DL) > (CH) ; then CF = 0, SF = 0, ZF = 0	The comparison is performed by subtracting the content of CH-register from the content of DL-register. The result is used to modify flags. The contents of DL and CH registers are not altered.
CMP CX, SI	Modify flags ← (CX) – (SI) If (CX) = (SI) ; then CF = 0, SF = 0, ZF = 1 If (CX) < (SI) ; then CF = 1, SF = 1, ZF = 0 If (CX) > (SI) ; then CF = 0, SF = 0, ZF = 0	The comparison is performed by subtracting the content of SI-register from the content of CX-register. The result is used to modify flags. The content of SI and CX registers are preserved.
CMP [BX], CL	MA = (DS) × 16_{10} + (BX) Modify flags ← (MA) – (CL) If (MA) = (CL) ; then CF = 0, SF = 0, ZF = 1 If (MA) < (CL) ; then CF = 1, SF = 1, ZF = 0 If (MA) > (CL) ; then CF = 0, SF = 0, ZF = 0	The comparison is performed by subtracting the content of CL-register from 8-bit memory. The result is used to modify flags. The content of CL-register and memory are preserved.
CMP [DI], 00FFH	MA = (DS) × 16_{10} + (DI) Modify flags ← (MA ; MA + 1) – $00FF_H$ If (MA ; MA + 1) = $00FF_H$; then CF = 0, SF = 0, ZF = 1 If (MA ; MA + 1) < $00FF_H$; then CF = 1, SF = 1, ZF = 0 If (MA ; MA + 1) > $00FF_H$; then CF = 0, SF = 0, ZF = 0	The comparison is performed by subtracting the 16-bit data ($00FF_H$) given in the instruction from 16-bit memory. The result is used to modify flags. The content of memory is preserved.
AND CX, DX	(CX) ← (CX) and (DX)	The content of CX and DX registers are ANDed and the result is stored in CX-register.
AND [BX + SI], AX	MA = (DS) × 16_{10} + (BX) + (SI) (MA ; MA + 1) ← (MA ; MA + 1) & (AX)	The content of 16-bit memory and AX-register are ANDed and the result is stored in memory.
AND CL, 0FH	(CL) ← (CL) and $0F_H$	The content of CL-register and the 8-bit data ($0F_H$) given in the instruction are ANDed. The result is stored in CL-register.

Table - 3.21 : continued ....

Instruction	Symbolic description	Explanation
OR AH, DL	$(AH) \leftarrow (AH) \vert (DL)$	The content of AH and DL registers are ORed and the result is stored in AH-register.
OR CX, [BP + DI + 05H]	$0005_H \xleftarrow{\text{Sign extend}} 05_H$ $MA_S = (SS) \times 16_{10} + (BP) + (DI) + 0005_H$ $(CX) \leftarrow (CX) \vert (MA_S ; MA_S + 1)$	The content of 16-bit stack memory and CX-register are ORed and the result is stored in CX-register.
OR DI, 0F0F0H	$(DI) \leftarrow (DI) \vert F0F0_H$	The content of DI-register is ORed with 16-bit data ($F0F0_H$) given in the instruction. The result is stored in DI-register.
XOR BX, DX	$(BX) \leftarrow (BX) \wedge (DX)$	The content of BX and DX registers are Exclusive-ORed. The result is stored in BX-register.
XOR SP, [SI + 0A00AH]	$MA = (DS) \times 16_{10} + (SI) + A00A_H$ $(SP) \leftarrow (SP) \wedge (MA ; MA + 1)$	The content of SP-register and 16-bit memory are Exclusive-ORed. The result is stored in SP-register.
XOR [DI + 05H], 0F0FH	$0005_H \xleftarrow{\text{Sign extend}} 05_H$ $MA = (DS) \times 16_{10} + (DI) + 0005_H$ $(MA ; MA + 1) \leftarrow (MA ; MA + 1) \wedge 0F0F_H$	The content of 16-bit memory is Exclusive-ORed with 16-bit data ($0F0F_H$) given in the instruction. The result is stored in memory.
TEST CX, DI	Modify flags $\leftarrow$ (CX) and (DI)	The TEST operation is performed by logically ANDing the content of CX and DI registers. The result is used to modify flags. The content of CX and DI registers are not altered.
TEST [SI + 0F0H], 0C000H	$FFF0_H \xleftarrow{\text{Sign extend}} F0_H$ $MA = (DS) \times 16_{10} + (SI) + FFF0_H$ Modify flags $\leftarrow$ (MA ; MA + 1) and $C000_H$	The TEST operation is performed by logically ANDing the content of 16-bit memory and 16-bit data given in the instruction. The result is used to modify flags. The content of memory is not altered.

3.13 SHORT QUESTIONS AND ANSWERS

3.1 *What is the size of 8086 instructions?*

The size of 8086 instruction is one to six bytes. The first byte consists of opcode and special bit indicators. The second byte will specify the addressing mode of the operands. The subsequent bytes will specify immediate data or address.

3.2 *Write the general format of 8086 instructions?*

The general format of 8086 instruction is shown in Fig. Q3.2.

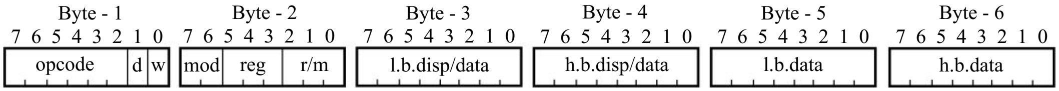


Fig. Q3.2 : General format of 8086 instruction.

3.3 *What is addressing?*

The method of specifying the data to be operated (operand) by the instruction is called addressing.

3.4 *What are the addressing modes available in 8086?*

The 8086 has the following 12 addressing modes.

i) Register addressing
ii) Immediate addressing
iii) Direct addressing
iv) Register indirect addressing
v) Based addressing
vi) Indexed addressing
vii) Based index addressing
viii) String addressing
ix) Direct IO port addressing
x) Indirect IO port addressing
xi) Relative addressing
xii) Implied addressing

3.5 *What is register addressing? Give example.*

In register addressing the instruction will specify the name of the register which holds the data to be operated by the instruction.

Example : MOV CX, DX - *The content of DX-register is moved to CX-register.*

3.6 *What is immediate addressing? Give example.*

In immediate addressing mode an 8-bit or 16-bit data is specified as part of the instruction.

Example : MOV BX, 0CA5H - *The 16-bit data (0CA5$_H$) given in the instruction is moved to BX-register.*

3.7 *Explain the direct addressing in 8086.*

In direct addressing an unsigned 16-bit displacement or signed 8-bit displacement will be specified in the instruction. The displacement is the effective address of the data. The 20-bit physical address of the data is computed by multiplying the content of DS-register by 16$_{10}$ and adding to effective address.

Example : MOV CL, [0F2AH] - *The memory address is computed by multiplying the content of DS-register by 16$_{10}$ and adding the 16-bit displacement (0F2A$_H$) given in the instruction. Then the content of memory is moved to CL-register.*

3.8 *Explain the register indirect addressing in 8086.*

In register indirect addressing the name of the register which holds the effective address of data will be specified in the instruction. The register used to hold the effective address are BX, SI or DI. The 20-bit physical address of data is obtained by multiplying the content of DS-register by 16$_{10}$ and adding to effective address.

Example : MOV DX, [DI] - *The memory address of the data is obtained by multiplying the content of DS-register by 16$_{10}$ and adding the content of DI-register. The content of memory is moved to DX-register.*

3.9 *Explain the based addressing in 8086.*

In based addressing the effective address of data is specified as a sum of base value and displacement. The register BX or BP is used to hold the base value. When BX holds the base value, the 20-bit physical address of data is calculated by multiplying the content of DS-register by 16$_{10}$ and adding to effective address. When BP holds the base value, the 20-bit physical address of data is calculated by multiplying the content of SS register by 16$_{10}$ and adding to effective address.

Example : MOV CX, [BP + 00A2H] - *The effective address is computed by adding the 16-bit displacement (00A2$_H$) given in the instruction to the content of BP-register. The 20-bit physical address is obtained by multiplying the content of SS-register by 16$_{10}$ and adding to effective address. The content memory is moved to CX-register.*

3.10 *Explain the indexed addressing in 8086.*

In indexed addressing the effective address of data is specified as a sum of index value and displacement. The register SI or DI is used to hold the index value. The 20-bit physical address of data is computed by multiplying the content of DS-register by 16$_{10}$ and adding to effective address.

Example : MOV AX, [DI + 04H] - *The effective address is computed by sign extending the 8-bit displacement given in the instruction to 16-bit and adding to the content of DI. The 20-bit physical address of memory is computed by multiplying the content of DS-register by 16_{10} and adding to effective address. The content of memory is moved to AX-register.*

3.11 What is based indexed addressing? Give an example.

In based indexed addressing, the effective address is specified as a sum of base value, index value and displacement. The base value is stored in BX or BP and the index value is stored in SI or DI-register. When BX holds the base value of effective address, the content of DS-register is considered as segment base address and when BP holds the base value, the content of SS-register is considered as segment base address.

The 20-bit physical address is computed by multiplying the segment base address by 16_{10} and adding to effective address.

Example : *MOV CX, [BP + DI + 01A0H] - The effective address is computed by adding the contents of BP-register, DI-register and the 16-bit displacement ($01A0_H$) given in the instruction. The 20-bit physical address of memory is computed by multiplying the content of SS-register by 16_{10} and adding to effective address. The content of memory is moved to CX-register.*

3.12 Explain string addressing in 8086.

In 8086, string addressing is used by string instructions to address the source and destination operand/data. In this mode the SI-register is used to hold the effective address of source data and DI-register is used to hold the effective address of destination. The memory address of source is obtained by multiplying the content of DS-register by 16_{10} and adding to effective address (content of SI-register). The memory address of destination is obtained by multiplying the content of ES-register by 16_{10} and adding to effective address (content of DI-register). After execution of string instruction the content of SI and DI are incremented or decremented depending on direction flag.

3.13 How are IO ports addressed in 8086?

The IO ports in 8086-based system can be addressed either by direct addressing or by indirect addressing. In direct addressing an 8-bit port address is directly specified in the instruction. In indirect addressing a 16-bit port address is stored in DX-register and the name of the register (DX) is specified in the instruction.

3.14 What is relative addressing?

In relative addressing, the effective address of a program instruction is specified relative to instruction pointer by an 8-bit signed displacement.

Example : JC 0F2H - *If carry flag is one, then a new effective address is calculated and loaded in instruction pointer. The new effective address is obtained by sign extending the 8-bit displacement ($F2_H$) given in the instruction and adding to the content of instruction pointer.*

3.15 What is implied addressing?

In implied addressing mode, the instruction itself will specify the data to be operated by the instruction.

Example : CLD - ***Cl****ear* ***D****irection* ***F****lag.*

3.16 List the data transfer instructions that affects the flags in 8086.

In 8086, the data transfer instructions affecting the flags are POPF and SAHF. The POPF instruction is used to restore the previously stored status of the flag. The instruction SAHF is used to modify the content of flag register.

3.17 List the instructions of 8086 that affect only carry flag.

The instructions that affect only carry flag are CLC, CMC and STC.

3.18 List the instructions of 8086 that affect direction flag.

The instructions that affect direction flag are CLD, POPF and STD.

3.19 List the instructions of 8086 that affects interrupt flag.

The instructions that affects interrupt flag are CLI, INT, INTO, IRET, POPF and STI.

3.20 What are the operations performed by data transfer instructions?

The operations performed by the data transfer instructions are:

i) Copy the content of a register to another register.

ii) Copy the content of a register/segment register to memory or vice versa.

iii) Copy the content of accumulator to port or vice versa.
iv) Exchange the content of two registers or register and memory.
v) Load an immediate operand to register/memory.
vi) Load effective address in segment registers.

3.21 *What are the operations performed by arithmetic instructions?*

The operations performed by arithmetic instructions are:

i) Addition or subtraction of binary, BCD or ASCII data.
ii) Multiplication or division of signed or unsigned binary data.
iii) Increment or decrement or comparison of binary data.

3.22 *What are the operations performed by logical instructions?*

The operations performed by logical instructions are AND, OR, Exclusive-OR, complement, arithmetic shift and logical shift.

3.23 *What are the operations performed by string instructions?*

The operations performed by string instructions are:

i) Copy a byte/word of a string data from data segment to extra segment.
ii) Compare the content of two memory locations or accumulator and a memory location.
iii) Load a byte/word of a string data from memory to accumulator or vice versa.

3.24 *List the string instructions of 8086.*

The string instructions of 8086 are REPZ/REPE, REPNZ/REPNE, MOVSB, MOVSW, CMPSB, CMPSW, SCASB, SCASW, LODSB, LODSW, STOSB and STOSW.

3.25 *What will be the content of stack pointer (SP) after a PUSH operation and after a POP operation?*

The PUSH operation will decrement the content of SP by two and so after a PUSH operation the content of SP will be less by two than earlier value.

The POP operation will increment the content of SP by two and so after a POP operation the content of SP will be greater by two than earlier.

3.26 *List the IO instructions of 8086.*

The IO instructions of 8086 are:

i) IN A, addr8	iii) OUT addr8, A
ii) IN A, [DX]	iv) OUT [DX], A

The IN instruction is used to load a byte/word from IO port to accumlator and the OUT instruction is used to send a byte/word from accumlator to IO port.

3.27 *Explain the instruction LEA reg16, mem.*

The instruction LEA is used to load the effective address of the memory operand to the register specified in the instruction (i.e., this instruction will not load the content of memory in register but the calculated effective address of memory data is loaded in the register).

3.28 *How can the low byte flag register be modified in 8086?*

The low byte of flag register can be modified by moving an 8-bit data to AH-register and then moving the content of AH to low byte flag register using SAHF instruction.

3.29 *How can the 16-bit flag register be modified in 8086?*

The steps involved in modifying the 16-bit flag register are given below:

i) First move a 16-bit data to a 16-bit register.
ii) Second save the content of register in stack using PUSH instruction.
iii) Finally move the top of stack to flag register using POPF instruction.

3.30 *How is subtraction performed in 8086 and how can the result be interpreted?*

The 8086 processor performs 2's complement subtraction and after subtraction the carry flag is complemented. Therefore the result of subtraction can be interpreted as follows:

i) After subtraction if carry flag is set (i.e., CF = 1) then the result is negative and the result will be in 2's complement form.
ii) After subtraction if carry flag is cleared/reset (i.e., CF = 0) then the result is positive.

3.31 What is the similarity and difference between subtract and compare instructions?

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 operation, the result is stored in destination register/memory, but after compare operation the result is discarded.

3.32 What will be the status of flags after division and multiplication operations?

The division and multiplication operation will modify all the six arithmetic flags (CF, AF, PF. ZF, SF and OF flags) but all these flags will be in undefined state after the division and multiplication operations.

3.33 What is the difference between Compare and Test operations in 8086?

In Compare operation the content of register or memory is subtracted from the content of another register and the result is used to modify the flags.

In Test operation the content of register or memory is bit by bit ANDed with the content another register and the result is used to modify the flags.

In both Compare and Test operations the content of source and destination are not altered.

3.34 What is the difference between arithmetic shift and logical shift?

In logical shift operation zero is inserted in the shifted location (i.e., zero is inserted in LSD position for left shift and for right shift zero is inserted in MSD position).

The arithmetic left shift operation is same as logical left shift, whereas in arithemetic right shift operation, the sign bit is copied into shifted location i.e., after every right shift the old value of MSD (**M**ost **S**ignificant **D**igit) is copied into the current MSD location.

3.35 What is the difference between shift and rotate operation?

In shift operation either zero or one is inserted in the shifted location, whereas in rotate operation ony the content of register/memory with or without carry are rotated (i.e., in rotate operation there is no insertion of extra bit in the shifted position).

3.36 What is near call and far call?

Near call refers to calling a procedure stored in the same code segment memory in which main program (or calling program)resides. Far call refers to calling a procedure stored in different code segment memory than that of main program.

While executing near call instructions the content of IP alone is pushed to stack. While executing far call the content of CS and IP are pushed to stack.

3.37 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.

3.38 What is the difference between conditional and unconditional branch instructions?

In unconditional branch instruction, 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 next instruction is executed.

3.39 What is near jump and far jump?

In near jump, the program control is transferred to new memory location in the same segment by modifying the content of **I**nstruction **P**ointer (IP). In far jump the program control is transferred to new memory location in another segment by modifying the content of **I**nstruction **P**ointer (IP) and **C**ode **S**egment (CS) register.

3.40 What is the difference between CALL and INT instruction?

While executing CALL instruction only IP and CS are saved in stack. But, while executing INT instruction IP, CS and flag register are saved in stack.

CHAPTER 4

MEMORY AND IO INTERFACING

4.1 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:

- Processor memory
- Primary or main memory
- Secondary memory

The processor memory refers to registers inside the microprocessor. These registers are used to hold the data and results temporarily when a 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 processor. However, the cost involved in this approach forces a manufacturer to include only a few registers in the microprocessor.

The 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 the primary memory prior to execution. In primary memories 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 comprising slow devices such as magnetic tapes and disks [hard disk, floppy disc and **C**ompact **D**isc (CD)]. They are known as auxiliary or backup storage. 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 the secondary memory to the main memory and work directly with the main memory only.

4.2 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 so they 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 $\mathbf{2^n \times m}$ bits. Figure 4.1 shows a simplified functional block diagram of semiconductor memory. The functional blocks of semiconductor memory are Row address decoder, Column address decoder, Memory array, Input buffer and Output buffer.

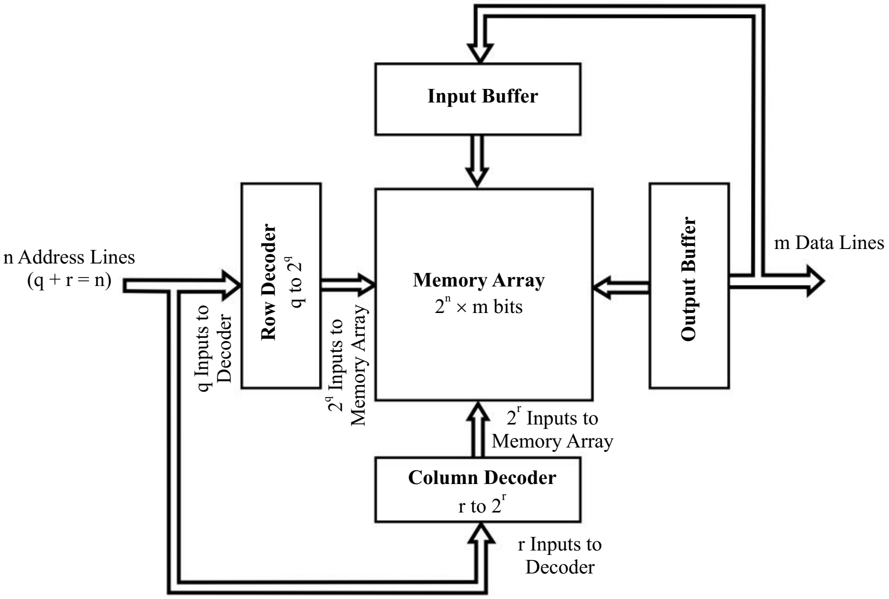


Fig. 4.1 : A simplified functional block diagram of a typical semiconductor memory.

The input and output buffers are used to hold the data until the valid time and also takes care of the signal current level matching (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 the column decoder.

The output lines of the row and column decoder are used to form a matrix array of size, $\mathbf{2^q \times 2^r}$ consisting of $\mathbf{2^n}$ crossing points as shown in Fig. 4.2. 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 that of Fig. 4.2 and all of them are wired in parallel. When an

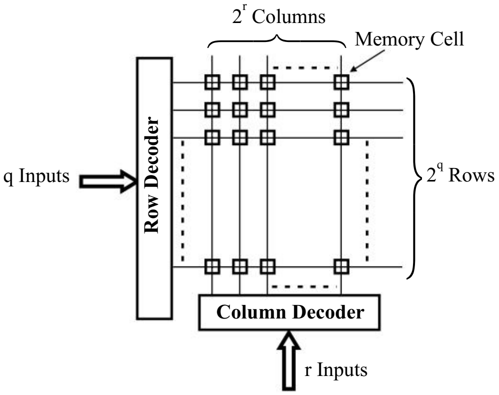


Fig. 4.2 : One layer of memory array.

address is send to memory IC, the row and column decoder will select one line each, which in turn select one memory cell in each layer. Thus, **m** memory cells are selected by an address. Then using read or write control signals, the data can be read or stored in the selected memory cells.

In the first version of semiconductor memory, the memory cells are 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 offer very low power and high speed operation.

The different types of semiconductor memory are ROM, PROM, EPROM, static RAM, DRAM and NVRAM. These semiconductor memories can be classified into volatile and nonvolatile memory. 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 if the power supply is switched OFF, then the memory is called nonvolatile. The ROM, PROM, EPROM and NVRAM are nonvolatile memories. The static RAM and DRAM are volatile memories.

The semiconductor memories can also be classified into read only memory and read/write memory. 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 information from these memories and cannot write into it. But in read/write memory, the processor can store (write) the 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.

The other features of semiconductor memories are random access and nondestructive readout. 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 first or 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 (**N**on**d**estructive **R**ead-**O**ut memory).

4.3 ROM AND PROM

ROM is a semiconductor memory which permits only a read access. 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 **R**ead-**O**nly **S**tore (ROS). In ROM memory, the memory cell (storage unit) will have a MOS transistor either with open gate or closed gate. The transistors with closed gate represent **1's** and with open gate represent **0's**. Since the configuration is fixed, they permanently store **1's** and **0's**.

ROM is a nonvolatile memory, i.e., loss of power or system malfunction does not change the contents of the memory. Also, the ROM memories have 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:

1. Custom programmed or Mask programmed ROM (ROM).
2. Programmable or Field programmable ROM (PROM).
3. Reprogrammable or Erasable-Programmable ROM (EPROM).

The custom programmed ROMs are programmed by the manufacturer as specified by the user during fabrication and the contents cannot be changed after packaging. The programmable ROMs are one-time programmable by the user. The reprogrammable ROMs have facilities for programming as well as for erasing its content and reprogramming the memory. The reprogrammable ROMs are erased either by passing electrical current or Ultraviolet light.

The programming of ROMs can be carried out using ROM (EPROM) programmer. Usually the ROM programmer is a digital system interfaced to a **P**ersonal **C**omputer (PC). The information to be programmed are first stored as a file in the PC and converted to the required binary format using a conversion software. Then the information is transferred from the PC to ROM programmer.

4.4 EPROM

The **R**ead **O**nly **M**emory (ROM) which has reprogrammable features is called EPROM (**E**rasable-**P**rogrammable **R**ead **O**nly **M**emory). The EPROM memory is nonvolatile and also has the feature of random access. In an EPROM, the binary informations are entered using electrical impulses and the stored information is erased using ultraviolet rays. 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 isolated gate. The isolated gate is located between the normal control gate and the source/drain region of a 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 is 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 **P**ersonal **C**omputer (PC) and the information to be programmed is downloaded from PC.

EPROMs are manufactured by many semiconductor industries like INTEL, Hitachi, Toshiba, Cypress, etc. The manufacturers have a common industry standard, so that a product from different industry will be pin-to-pin compatible and slightly differ in electrical and switching characteristics. The various features of 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. 4.3 and the pin configuration during read mode is shown in Fig. 4.4. [The pin configuration of CY27C64 will be different to that of Fig. 4.4 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 the read operation.

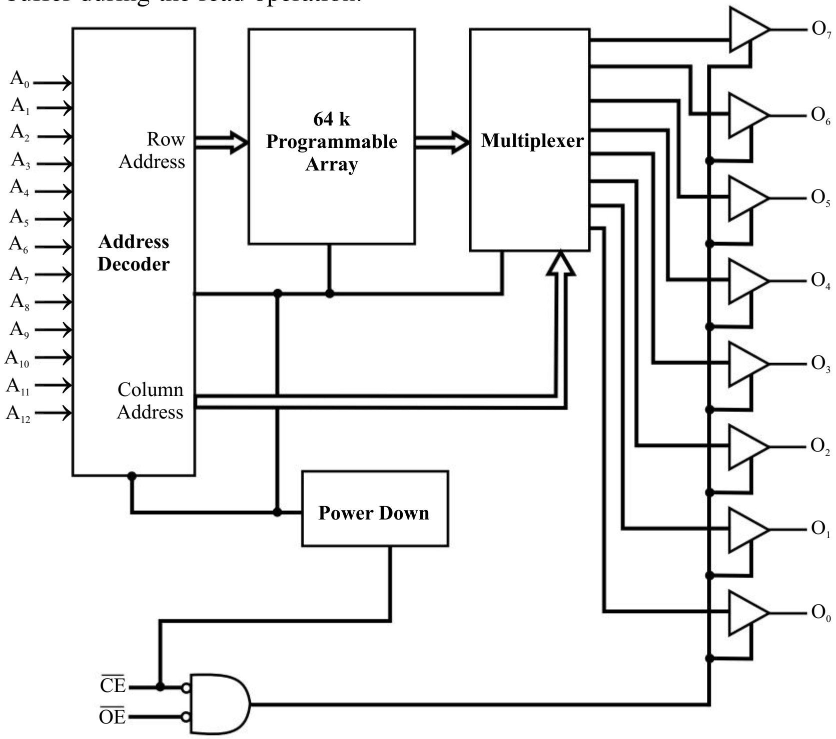
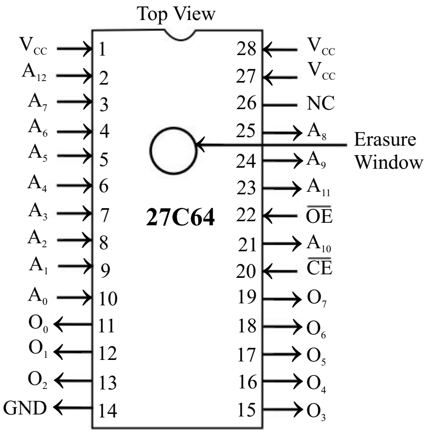


Fig. 4.3 : Logic block diagram of a CY27C64.



Pin	Description
A_0 - A_{12}	Address
O_0 - O_7	Output/Data
$\overline{CE}$	Chip Enable
$\overline{OE}$	Output Enable
V_{CC}	Power supply, +5-V
GND	Ground (0-Volt)
NC	No Connection

Fig. 4.4 : Pin configuration of a CY27C64 in read mode.

The CY27C64 EPROM is available with maximum access time of 70, 90, 120, 150 or 200 ns (nanosecond). The electrical characteristics or ratings of the EPROM are listed in Table-4.1.

TABLE - 4.1 : 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. 4.5. Only four important timings are shown in this diagram. For detailed discussions on timing diagram refer to manufacturer's data sheet. The switching timings of various signals of CY27C64 are listed in Table-4.2.

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**.

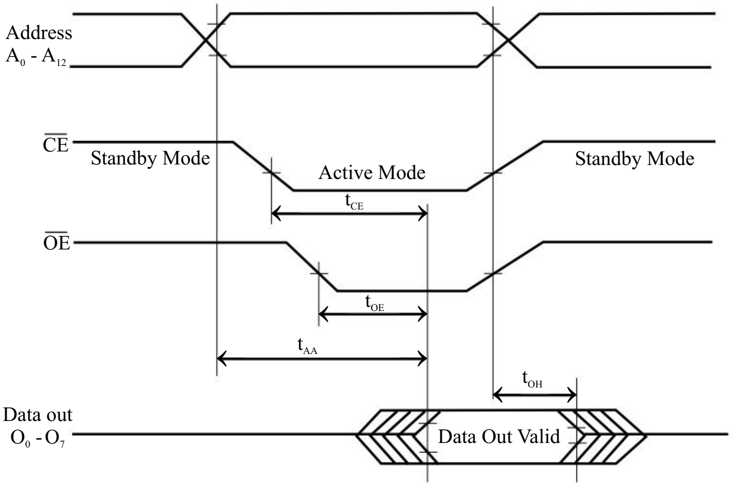


Fig. 4.5 : Timing diagram of a CY27C64 for read operation.

TABLE - 4.2 : 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 4000A° (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 a very long time.

The recommended dose of UV light for erasure is a wavelength of 2537 A° for a minimum dose (UV intensity multiplied by exposure time) of 25 W-sec/cm^2. For an UV lamp with a 12 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 a very long time. (Maximum dosage is 7258 W-sec/cm^2).

4.5 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. [Even though ROM memories are also technically random access memory, 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) consist of a flip-flop or 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 companies like Motorola, Hitachi, Toshiba, Cypress, etc. The manufacturers have a common industry standard, so that a product from different companies will be pin-to-pin compatible and olny slightly differ in electrical and switching characteristics. The various features of 6264 (8 kb RAM) manufactured by Cypress semiconductor corporation are discussed in this section.

CY6264 (Cypress Make CMOS 6264)

The CY6264 is a high performance CMOS static RAM organized as 8192 bytes (8 kb). 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 Figs. 4.6 and 4.7. 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-IO_4.

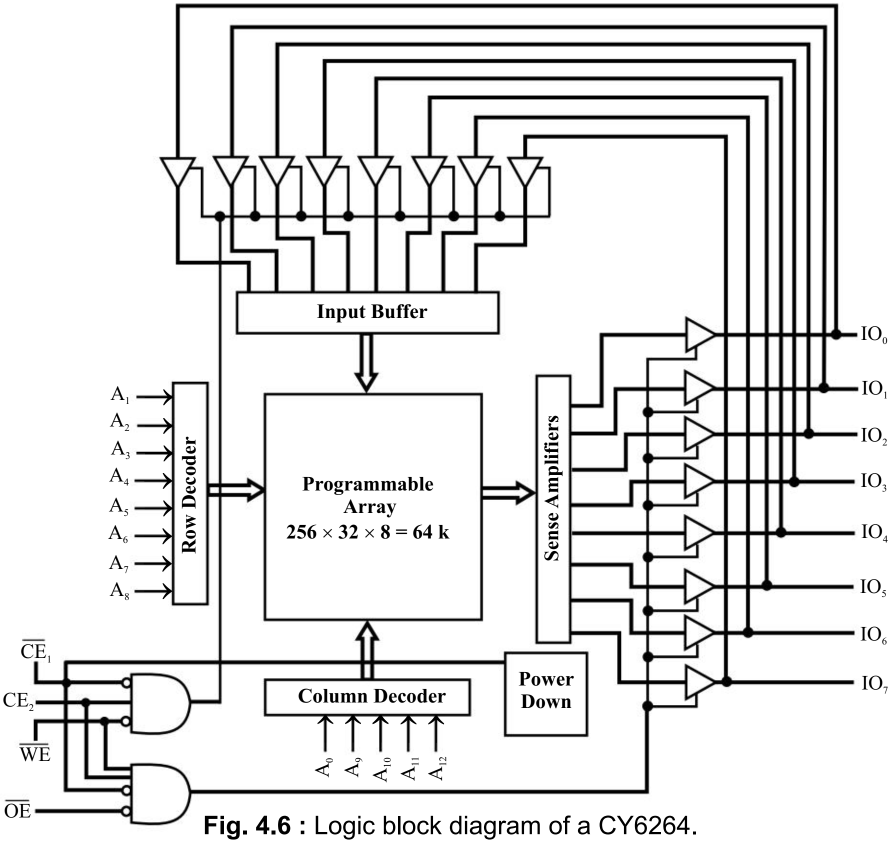
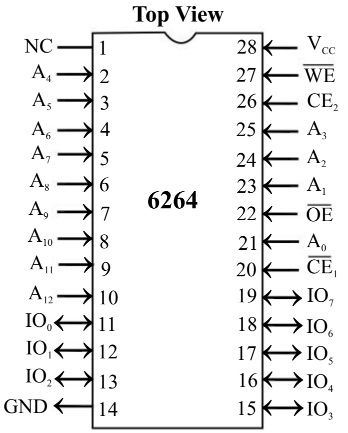


Fig. 4.6 : Logic block diagram of a CY6264.



Pin	Description
A_0 - A_{12}	Address
IO_0 - IO_7	Input/Output Data
$\overline{CE_1}$	Active **low** Chip Enable
CE_2	Active **high** Chip Enable
$\overline{WE}$	Active **low** Write Enable
$\overline{OE}$	Active **low** Output Enable
V_{CC}	Power supply, +5-V
GND	Ground (0-Volt)
NC	No Connection

Fig. 4.7 : 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 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_4.

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-4.3.

TABLE - 4.3 : 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

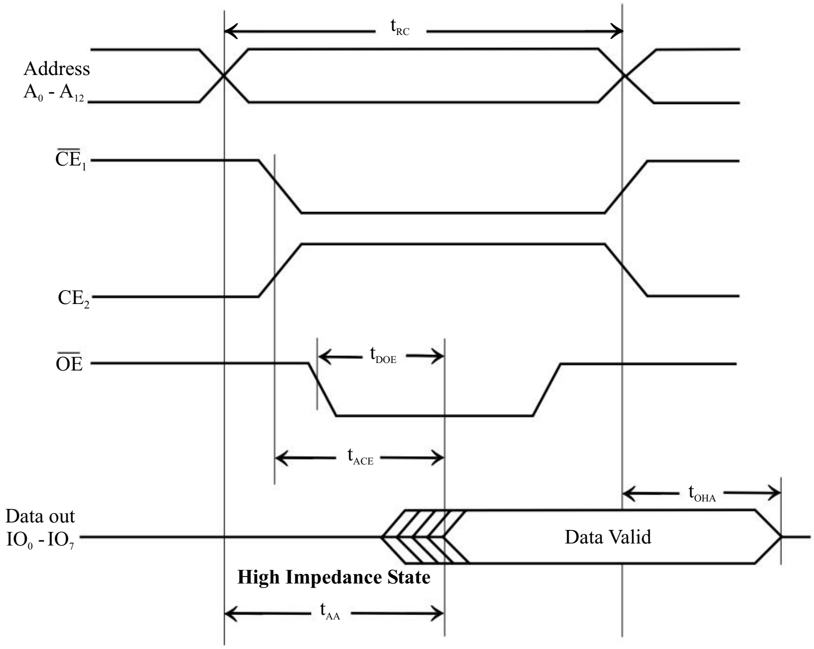


Fig. 4.8 : Read cycle timings of a CY6264.

TABLE - 4.4 : 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 **low/high** to data valid		70	ns
t_{DOE}	$\overline{OE}$ **low** to data valid		35	ns

The timing diagram (or switching waveforms) of CY6264 for read operation is shown in Fig. 4.8. Only five important timings are shown in this diagram. For detailed discussions on timing diagram, please refer to manufacturer's data sheet. The timings of various signals of CY6264 are listed in Table-4.4.

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 **C**hip **E**nable-1 ($\overline{CE}_1$) as **low** and **C**hip **E**nable-2 (CE_2) as **high**.
3. Assert the **O**utput **E**nable ($\overline{OE}$) signal **low**.
4. When $\overline{OE}$ signal is asserted **low**, the data can be read from the input/output lines (IO_0-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. 4.9. The diagram shows some important timings of write cycle. For detailed discussions on timing diagram, refer to the manufacturer's data sheet. The timings of various signals are listed in Table-4.5.

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-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).

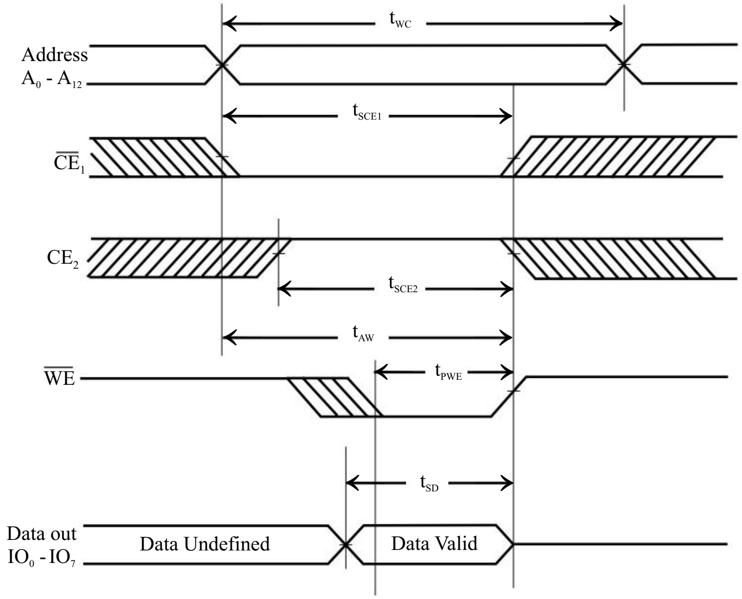


Fig. 4.9 : Write cycle timings of CY6264.

TABLE - 4.5 : 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

4.6 DRAM AND NVRAM

DRAM

DRAM (**D**ynamic **RAM**) is a read/write memory in which the 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 needs extra circuitry and so the interfacing of DRAMs to the microprocessor are more complex than the interfacing of static RAMs. The recent versions of DRAMs have internal refreshing circuit. The manufacturing of DRAMs will be cheaper only for very large capacity memories. Therefore, smaller memories are generally static elements (up to 256 kb) and large memories (> 1Mb) are typically dynamic.

NVRAM

NVRAMs are nonvolatile read/write memories. They are also called flash memory. These memory devices are electrically erasable in the system, but require more time to erase than a static RAM. Therefore, they are also called EEPROM (**E**lectrically **E**rasable **P**rogrammable **ROM**). The drawback in EEPROMs is that it takes longer 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.

INTEL and XYCOR have released their versions of nonvolatile RAMs, 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 has a special pin called nonvolatile enable ($\overline{\text{NE}}$). Normally, this pin is **high** and read or write operation is performed with static RAM. When $\overline{\text{NE}}$ is asserted **low**, the data in the static RAM cells are written into the corresponding EEPROM cells.

4.7 INTERFACING STATIC RAM AND EPROM

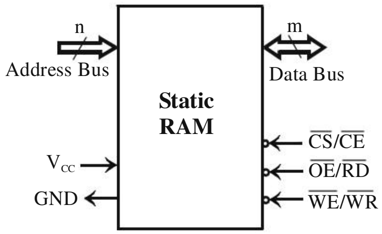
The primary function of memory interfacing is that the microprocessor should be able to read from and write into a set of semiconductor memory IC chips. Generally, EPROM is interfaced for read operations and RAM is interfaced for read and write operations. The procedure for interfacing SRAM for read/write operation and EPROM for read operation are similar and hence, they are commonly dealt in this section.

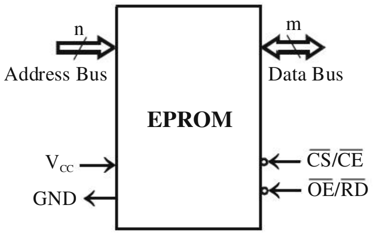
In order to perform the read/write operation the memory access time should be less than the read/write time of processor, chip select signals should be generated for selecting a particular memory IC, suitable control signals should be generated for read/write operation and a specific address should be 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). Typical static RAM and EPROM are shown in Figs. 4.10 and 4.11 respectively.


n
Address Bus
m
Data Bus
Static RAM
V_{CC}
GND
$\overline{CS}/\overline{CE}$
$\overline{OE}/\overline{RD}$
$\overline{WE}/\overline{WR}$



$\overline{CS}/\overline{CE}$ - Chip Select (or Chip Enable) ; $\overline{OE}/\overline{RD}$ - Output Enable (or Read Control)

$\overline{WE}/\overline{WR}$ - Write Enable (or Write Control)

Fig. 4.10 : A typical static RAM IC.

Fig. 4.11 : A typical EPROM IC in read mode.

> *Note : The pins of EPROM are redefined for write operation. It also requires a different hardware setup and higher 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 the data stored in each memory location is called memory word size. In INTEL 8086-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-4.4.

TABLE - 4.6 : 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	$2^{10} = 1024$	1 k	400	000 to 3FF
11	$2^{11} = 2 \times 2^{10} = 2048$	2 k	800	000 to 7FF
12	$2^{12} = 2^2 \times 2^{10} = 4 \times 2^{10} = 4096$	4 k	1000	000 to FFF
13	$2^{13} = 2^3 \times 2^{10} = 8 \times 2^{10} = 8192$	8 k	2000	0000 to 1FFF
14	$2^{14} = 2^4 \times 2^{10} = 16 \times 2^{10} = 16384$	16 k	4000	0000 to 3FFF
15	$2^{15} = 2^5 \times 2^{10} = 32 \times 2^{10} = 32768$	32 k	8000	0000 to 7FFF
16	$2^{16} = 2^6 \times 2^{10} = 64 \times 2^{10} = 65536$	64 k	10000	0000 to FFFF
17	$2^{17} = 2^7 \times 2^{10} = 128 \times 2^{10} = 131072$	128 k	20000	00000 to 1FFFF
18	$2^{18} = 2^8 \times 2^{10} = 256 \times 2^{10} = 262144$	256 k	40000	00000 to 3FFFF
19	$2^{19} = 2^9 \times 2^{10} = 512 \times 2^{10} = 524288$	512 k	80000	00000 to 7FFFF
20	$2^{20} = 2^{10} \times 2^{10} = 1024 \times 2^{10} = 1048576$	1024 k = 1 M	100000	00000 to FFFFF

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 program**,** other permanent programs and data. The RAM memories are used for stack operations, temporary program and data storage.

The popular EPROM and static RAM ICs used with 8086-systems and their capacity are listed here. The Table-4.7 shows the number of address pins and data pins available on these ICs.

EPROM	**Static RAM**
2708 (1 k × 8 = 8 kilo bits/1 kb)	6208 (1 k × 8 = 8 kilo bits/1kb)
2716 (2 k × 8 = 16 kilo bits/2 kb)	6216 (2 k × 8 = 16 kilo bits/2 kb)
2732 (4 k × 8 = 32 kilo bits/4 kb)	6232 (4 k × 8 = 32 kilo bits/4 kb)
2764 (8 k × 8 = 64 kilo bits/8 kb)	6264 (8 k × 8 = 64 kilo bits/8 kb)
27256 (32 k × 8 = 256 kilo bits/32 kb)	62256 (32 k × 8 = 256 kilo bits/32 kb)
27512 (64 k × 8 = 512 kilo bits/64 kb)	62512 (64 k × 8 = 512 kilo bits/64 kb)
27010 (128 k × 8 = 1 Mega-bit/128 kb)	62128 (128 k × 8 = 1 Mega-bit/128 kb)
27020 (256 k × 8 = 2 Mega bits/256 kb)	62138 (256 k × 8 = 2 Mega bits/256 kb)
27040 (512 k × 8 = 4 Mega bits/512 kb)	62148 (512 k × 8 = 4 Mega bits/512 kb)

Note : In this book kb refers to kilobytes.

TABLE - 4.7 : 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
27010/62128	128 kb	17	8
27020/62138	256 kb	18	8
27040/62148	512 kb	19	8

Note : 16 kb memory is not available as a standard product.

In 8086 system, the EPROMs are mapped at the end of memory space and RAMs are mapped at the begining of memory space (i.e., 00000_H address is alloted to RAM and $FFFFF_H$ is alloted to EPROM). This organization will facilitate automatic execution of monitor program and creation of interrupt vector table in RAM upon reset. 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, except CS-register all other internal registers, flag register and instruction pointer will be cleared. The CS-register is initialized with FFFF_H. Hence, after a reset, the processor starts fetching and executing the instruction stored at FFFF0_H. $[(\text{CS}) \times 16_{10} + (\text{IP}) = \text{FFFF0}_H$ after a reset].

The system designer will store the monitor program starting from the address FFFF0_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 at the end of memory space in 8086-based system. Monitor program is a permanent program written by the system designer to take care of system initializations. The system initializations includes the following.

1. Programming 8279 for keyboard scanning and display refreshing.
2. Programming peripheral ICs 8259,8257,8255,8251,8254,etc.,.
3. Initializing stack.
4. Display a message on display (output) device.
5. Intializing interrupt vector table.

Note :	*8279 - Programmable keyboard/display controller*	*8257 - DMA controller*
	8259 - Programmable interrupt controller	*8251 - USART*
	8255 - Programmable peripheral interface	*8254 - Programmable timer*

Generation of Chip Select Signals

The decoders are used for generating 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.

The decoder is a logic circuit that identifies each combination of the signals present at its input. The decoders will have **n** input lines and **2^n** output lines. In logic **low** decoder, at any one time one of the 2^n output 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, the low order address lines are connected to the memory ICs. The remaining unused address lines (or the high order address line) are connected to input of decoder. The outputs of decoder are connected to $\overline{\text{CS}}$ or $\overline{\text{CE}}$ 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 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 IC 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 and so they are electrically isolated from the system. The read/write operation is performed by the processor with the selected memory IC.

Decoder

The 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 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 logic **high** enable. The pin configuration of 3-to-8 decoder (74LS138) is shown in Fig. 4.12. The truth table of the decoder is given in Table-4.8.

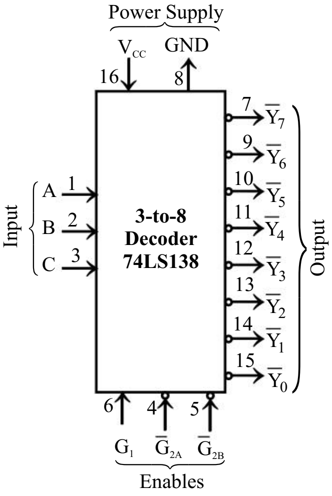


Fig. 4.12 : Signals of 74LS138.

TABLE - 4.8 : 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	1	0	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	0	1	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

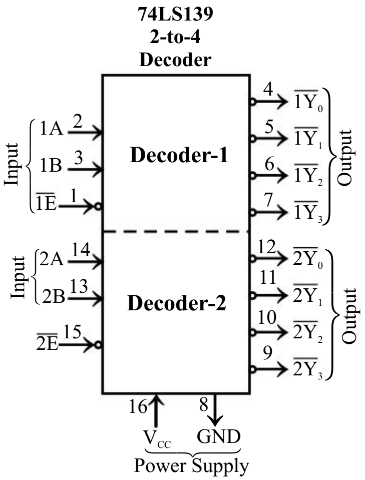


Fig. 4.13 : Signals of 74LS139.

TABLE - 4.9 : TRUTH TABLE OF 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

The 74LS139 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. 4.13. The truth table of 2-to-4 decoder is given in Table-4.9. In 74LS139 each decoder can work independently.

4.8 MEMORY ORGANIZATION IN 8086-BASED SYSTEM

In 8086, the one mega-byte (1Mb) of addressable memory space is divided into two banks : Even memory bank and Odd memory bank. Each bank will have an addressable space of 512 kilo bytes (512 kb). In 8086-based system the lower eight lines of data bus, D_0-D_7 are connected to even bank memory ICs and the upper eight lines of data bus, D_8-D_{15} are connected to odd bank memory ICs. The even bank is selected by the address line A_0 and the odd bank is selected by the control signal $\overline{BHE}$.

A microprocessor-based system requires both EPROM and RAM. Hence, the available memory space has to be divided between EPROM and RAM. This choice depends on the system designer as well as on the application for which the system is designed. For proper system functioning, the system designer should allot equal address space in odd and even bank for both EPROM and RAM.

Some systems may require large memory space and so full memory space is utilized. But in some systems, the memory requirement may be less and in this case the full memory space is not utilized. When full memory space is not utilized for memory, then 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 memory device.

The required EPROM memory capacity of the system can be implemented in two ICs (one for even and the other for odd bank) or in multiple ICs. Similarly, the RAM capacity of the system can be implemented in two ICs or in multiple ICs. This choice depends on the availability of memory IC and the system designer. Some example memory organisations for 8086 processor based-system are discussed in this section.

Consider a system in which the full memory space is utilized and the memory space is equally divided between EPROM and RAM. For this system in each memory bank 256 kilo bytes of EPROM memory space and 256 kilo bytes of RAM memory space is available. If the 256 kb memory space is implemented in single IC then we require two numbers of 256 kb RAM and two numbers of 256 kb EPROM. This memory organization is shown in Fig. 4.14.

In the memory organization shown in Fig. 4.14, the lower eight data lines D_0-D_7 are connected to even bank ICs, the upper eight data lines D_8-D_{15} are connected to the odd bank ICs and the address lines A_1-A_{18} are used to select the internal locations of memory ICs. The even bank ICs are selected when A_0 is **low** and the odd bank ICs are selected when $\overline{BHE}$ is **low**. The EPROM or RAM selection is decided by address line A_{19}. When A_{19} is **low**, RAM ICs are selected and when A_{19} is **high** EPROM ICs are selected.

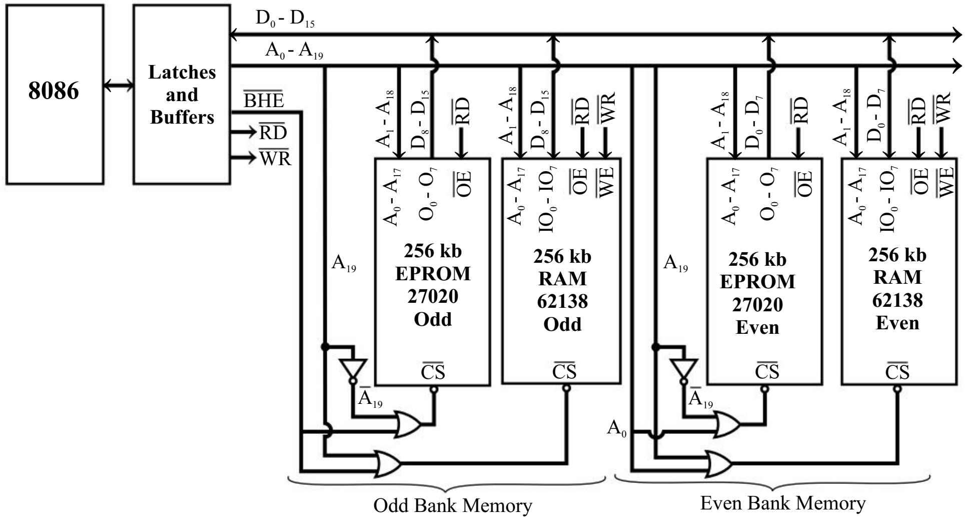


Fig. 4.14 : Example of implementing full memory space in 8086-based system.

In this organization the first half address space (i.e., first 512 kb) is implemented in RAM. The address range of RAM memory is 00000_H to $7FFFF_H$. The second half address space (i.e., second 512 kb) is implemented in EPROM. The address range of EPROM memory is 80000_H to $FFFFF_H$. The address allocation for RAM and EPROM locations are shown in Table-4.10.

Note : While alloting binary address to odd bank memory, A_0 is considered as 1.

Let us discuss another example of memory organization with unequal memory space for EPROM and RAM and utilized only half the available memory space. Consider a system with 128 kb (2 × 64 kb = 128 kb) EPROM and 384 kb (6 × 64 kb = 384 kb) RAM. For this system in each bank 64 kb of EPROM and 192 kb of RAM is available. If the memory system is designed using 64 kb memory IC then we may require two numbers of 64 kb EPROM and six numbers of 64 kb RAM. This memory organisation is shown in Fig. 4.15.

In the memory organization shown in Fig. 4.15, the lower eight data lines, D_0-D_7 are connected to even bank ICs, the upper eight data lines, D_8-D_{15} are connected to the odd bank ICs and the address lines A_1-A_{16} are used to select the internal locations of memory ICs. The address lines A_{17}, A_{18} and A_{19} are decoded to generate chip select signals. Here two numbers of 3-to-8 decoders are employed for generating separate chip select signals for even and odd bank ICs. Each 3-to-8 decoder will generate eight chip select signals and in this, four signals are used as chip select signals for four memory ICs of a bank. The remaining four signals are reserved for future expansion (or can be used for IO devices).

In the memory organization shown in Fig. 4.15, the first 384 kb of memory space is implemented in RAM and the last 128 kb of memory space is implemented in EPROM. The address range of RAM is 00000_H to $5FFFF_H$ and the address range of EPROM is $E0000_H$ to $FFFFF_H$. The address allocation for RAM and EPROM locations are shown in Table-4.11.

TABLE- 4.10 : ADDRESS ALLOCATION TABLE FOR MEMORY ORGANIZATION SHOWN IN FIG. 4.14

Memory Device	Binary address																				Hexa address	
	Input to memory address pins																					
	A_{19}	A_{18}	A_{17}	A_{16}	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		
RAM 256 kb Even	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0 0 0 0 0	RAM Address Range 00000_H to $7FFFF_H$
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0 0 0 0 2	
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0 0 0 0 4	
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0 0 0 0 6	
	.		.				.			.				.				.			.	
	.		.				.			.				.				.			.	
	.		.				.			.				.				.			.	
	.		.				.			.				.				.			.	
	.		.				.			.				.				.			.	
	.		.				.			.				.				.			.	
	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	7 F F F E	
RAM 256 kb Odd	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0 0 0 0 1	
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0 0 0 0 3	
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0 0 0 0 5	
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0 0 0 0 7	
	.		.			.				.				.				.			.	
	.		.			.				.				.				.			.	
	.		.			.				.				.				.			.	
	.		.			.				.				.				.			.	
	.		.			.				.				.				.			.	
	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	7 F F F F	
EPROM 256 kb Even	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	8 0 0 0 0	EPROM Address Range 80000_H to $FFFFF_H$
	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	8 0 0 0 2	
	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	8 0 0 0 4	
	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	8 0 0 0 6	
	.		.			.				.				.				.			.	
	.		.			.				.				.				.			.	
	.		.			.				.				.				.			.	
	.		.			.				.				.				.			.	
	.		.			.				.				.				.			.	
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	F F F F E	
EPROM 256 kb Odd	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	8 0 0 0 1	
	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	8 0 0 0 3	
	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	8 0 0 0 5	
	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	8 0 0 0 7	
	.		.			.				.				.				.			.	
	.		.			.				.				.				.			.	
	.		.			.				.				.				.			.	
	.		.			.				.				.				.			.	
	.		.			.				.				.				.			.	
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	F F F F F	

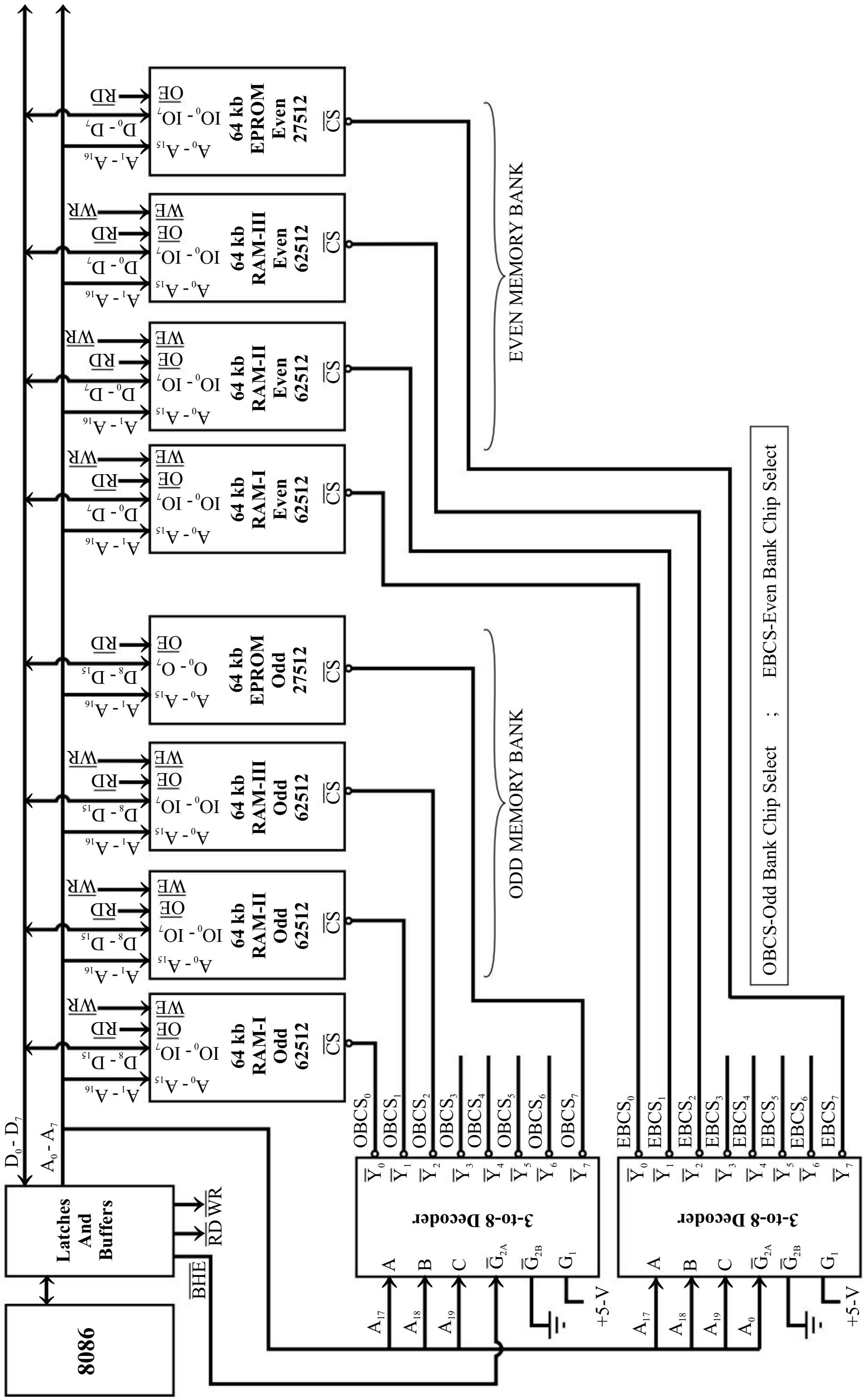


Fig. 4.15 : Example of implementing 512 kb memory space in an 8086-based system.

TABLE - 4.11 : ADDRESS ALLOCATION TABLE FOR MEMORY ORGANISATION SHOWN IN FIG 4.15

Memory Device	Binary Address																				Hexa Address		
	Decoder input			Input to memory address pins																			
	A_{19}	A_{18}	A_{17}	A_{16}	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}			
64 kb RAM-I Even	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0 0 0 0 0	RAM-I Address Range : 00000_H to $1FFFF_H$	Total RAM Address Range : 00000_H to $5FFFF_H$
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0 0 0 0 2		
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0 0 0 0 4		
	:		:	:			:			:				:				:			:		
	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1 F F F E		
64 kb RAM-I Odd	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0 0 0 0 1		
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0 0 0 0 3		
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0 0 0 0 5		
	:		:			:				:				:				:			:		
	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1 F F F F		
64 kb RAM-II Even	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2 0 0 0 0	RAM-II Address Range : 20000_H to $3FFFF_H$	
	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	2 0 0 0 2		
	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	2 0 0 0 4		
	:		:			:				:				:				:			:		
	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	3 F F F E		
64 kb RAM-II Odd	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	2 0 0 0 1		
	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	2 0 0 0 3		
	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	2 0 0 0 5		
	:		:			:				:				:				:			:		
	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	3 F F F F		
64 kb RAM-III Even	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4 0 0 0 0	RAM-III Address Range : 40000_H to $5FFFF_H$	
	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	4 0 0 0 2		
	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	4 0 0 0 4		
	:		:				:			:				:				:			:		
	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	5 F F F E		
64 kb RAM-III Odd	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	4 0 0 0 1		
	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	4 0 0 0 3		
	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	4 0 0 0 5		
	:		:			:				:				:				:			:		
	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	5 F F F F		
64 kb EPROM Even	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	E 0 0 0 0	EPROM Address Range : $E0000_H$ to $FFFFF_H$	
	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	E 0 0 0 2		
	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	E 0 0 0 4		
	:		:			:				:				:				:			:		
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	F F F F E		
64 kb EPROM Odd	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	E 0 0 0 1		
	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	E 0 0 0 3		
	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	E 0 0 0 5		
	:		:			:				:				:				:			:		
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	F F F F F		

4.9 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 **C**ompact **D**isc (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

1. Programmed IO
2. Interrrupt driven IO
3. **D**irect **M**emory **A**ccess (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 the 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. The Fig. 4.16 shows all the types of data transfer schemes in a microcomputer and it can also be called IO structure of a microcomputer.

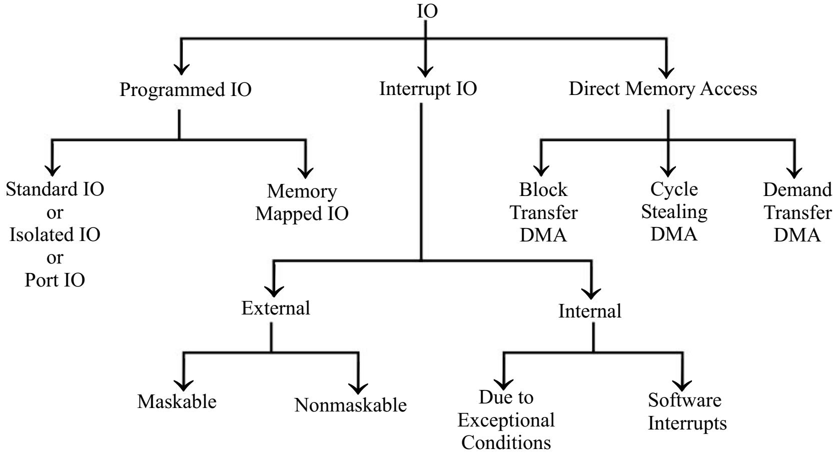


Fig. 4.16 : IO structure of a typical microcomputer.

4.10 INTERFACING IO AND PERIPHERAL DEVICES

IO devices are generally slow devices and so they are connected to the system bus through the ports. The ports are buffer ICs which are used to temporarily hold the data transmitted from microprocessor to IO device or to hold the data transmitted from IO device to microprocessor.

For data transfer from input device to processor the following operations are performed:

1. The input device will load the data to the port.
2. When the port receives a data, it sends message to the processor to read the data.
3. The processor will read the data from the port.
4. After a data have been read by the processor the input device will load the next data into the port.

For data transfer from processor to output device the following operations are performed:

1. The processor will load the data to the port.
2. The port will send a message to the output device to read the data.
3. The output device will read the data from the port.
4. After the data have 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.

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 - DI_8 and 8-output lines DO_1 - 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 the 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 $DS_2 = 1$

8155 : It has 256 × 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. The 8155 requires six internal addresses and has one logic **low** **C**hip **S**elect pin ($\overline{CS}$). The address of internal devices of 8155 are listed in Table-4.12.

TABLE - 4.12 : 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

8156 : Same as 8155, but it has logic **high** **C**hip **S**elect (CS), i.e., the chip is selected when CS = 1.

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, the port-C can be used as an IO port. The individual pins of port-C can be set or reset. The 8255 requires four internal addresses and has one logic **low** **C**hip **S**elect ($\overline{CS}$) pin. The address of internal devices of 8255 are listed in Table-4.13.

TABLE - 4.13 : 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

TABLE - 4.14 : INTERNAL ADDRESS OF 8355/8755

Internal Device	A_1	A_0
Port- A	0	0
Port- B	0	1
DDR-A	1	0
DDR-B	1	1

8355 : It has 2k × 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-6.16. The 8355 requires four internal addresses and has one logic **low** **C**hip **S**elect ($\overline{CS}$) pin.

8755 : Same as 8355 but has 2k × 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 8086 system and their functions and internal addresses are listed in Table-4.15.

TABLE - 4.15 : FUNCTIONS AND INTERNAL ADDRESSES OF PERIPHERAL DEVICES

Device	Function	Internal addresses
INTEL 8279	Keyboard/display controller. Used for keyboard scanning and display refreshing.	**Two-internal addresses** A_0 = 0 → Data register A_0 = 1 → Control register
INTEL 8257 or INTEL 8237	DMA controller. Used for supporting DMA access to IO device. It acts as a master during the DMA mode. It is a slave device during programming mode.	**Sixteen-internal addresses** 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 8085-based system and 256 interrupts in 8086-based system.	**Two-internal addresses** A_0 = 0 A_0 = 1
8253/ 8254	Used in the system to produce various timing signals. It has three independent counters and can be programmed in six operating modes.	A_1 A_0 Counter-0 0 0 Counter-1 0 1 Counter-2 1 0 Control Register 1 1
INTEL 8251 USART	Universal Synchronous/Asynchronous Receiver Transmitter. Used for serial data communication.	**Two-internal addresses** C/ $\overline{D}$ = 0 → Data register C/ $\overline{D}$ = 1 → Control register

The port and peripheral devices will have one logic **low/high** chip select pin. The processor can access the port/peripheral device by supplying the internal address and chip select signal. 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 8086-based system. *[The interfacing of IO ports and controller/peripheral ICs are commonly referred as IO device mapping.]*

The two methods are,

1. Memory-mapped IO device.
2. Standard IO-mapped IO device or Isolated IO mapping.

In memory mapping of IO devices, the ports are allotted a 20-bit address like that of 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 or 16-bit address is allotted for 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 or 16-bit address is placed on low order address lines and the high order address lines are asserted zero.
3. M/$\overline{\text{IO}}$ is asserted **low** to indicate the IO operation (for read as well as write).

The 8086 processor does not provide separate read ($\overline{\text{RD}}$) and ($\overline{\text{WR}}$)signals for memory and IO devices. But it differentiates the memory and IO device accesses by M/$\overline{\text{IO}}$ signal. The three signals $\overline{\text{RD}}$,$\overline{\text{WR}}$ and M/$\overline{\text{IO}}$ can be decoded as shown in Fig. 4.17 to provide separate read and write control signals for IO devices and memory devices.

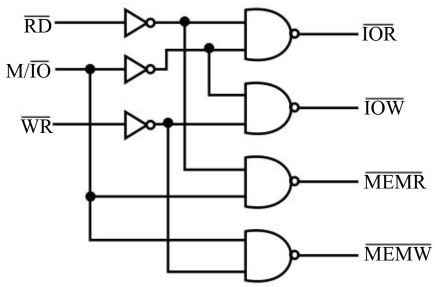


Fig. 4.17 : Circuit to generate separate read and write signals for memory and IO devices in an 8086 system.

When the devices are IO-mapped, then only IN and OUT instructions has to be used for data transfer between the device and the processor. For IO-mapped devices, a separate decoder should be used to generate the required chip select signals.

TABLE - 4.16 : COMPARISON OF MEMORY MAPPING AND IO MAPPING OF IO DEVICE

Memory mapping of IO device	IO mapping of IO device
1. 20-bit addresses are provided for IO devices.	1. 8-bit or 16-bit addresses are provided for IO devices.
2. 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 IO device and the processor.	2. Only IN and OUT instructions can be used for data transfer between IO device and the processor.
3. In memory-mapped ports, the data can be moved from any register to the ports and vice versa.	3. In IO-mapped ports, the data transfer can take place only between the accumulator and the ports.
4. 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.	4. When IO mapping is used for IO devices, then the full memory address space can be used for addressing memory. Hence it is suitable for systems which requires large memory capacity.
5. For accessing the memory-mapped devices, the processor executes memory read or write cycle. During this cycle M/$\overline{\text{IO}}$ is asserted **high**.	5. For accessing the IO-mapped devices, the processor executes IO read or write cycle. During this cycle M/$\overline{\text{IO}}$ is asserted **low**.

DESIGN EXAMPLE 1

In a microprocessor system using 8086, the memory requirement is 16 kb EPROM and 16 kb RAM. The system requires 8279 for keyboard and display interface, and 8255 for IO ports. Draw an interface diagram for memory and peripheral devices, and allot addresses for each device.

Solution

The 16 kb EPROM can be implemented in two numbers of 8 kb EPROM (2764). One of the 8 kb EPROM can be mapped as even bank and the other as odd bank. The address lines A_1-A_{13} are connected to each EPROM IC to select the internal locations of EPROM.

The 16 kb RAM can be implemented in two numbers of 8 kb RAM (6264). One of the 8 kb RAM can be mapped as even bank and the other as odd bank. The address lines A_1-A_{13} are connected to each RAM IC to select the internal locations of RAM.

Since a large amount of memory space is free, we can interface the 8279 and 8255 as memory-mapped device. In the interface diagram shown in Fig. E1.1, these devices are interfaced such that even addresses are allotted to them. The address line A_1 of 8086 is connected to address line A_0 of 8279, and the address lines A_1 and A_2 of 8086 are connected to address lines A_0 and A_1 of 8255 to provide the required internal addresses.

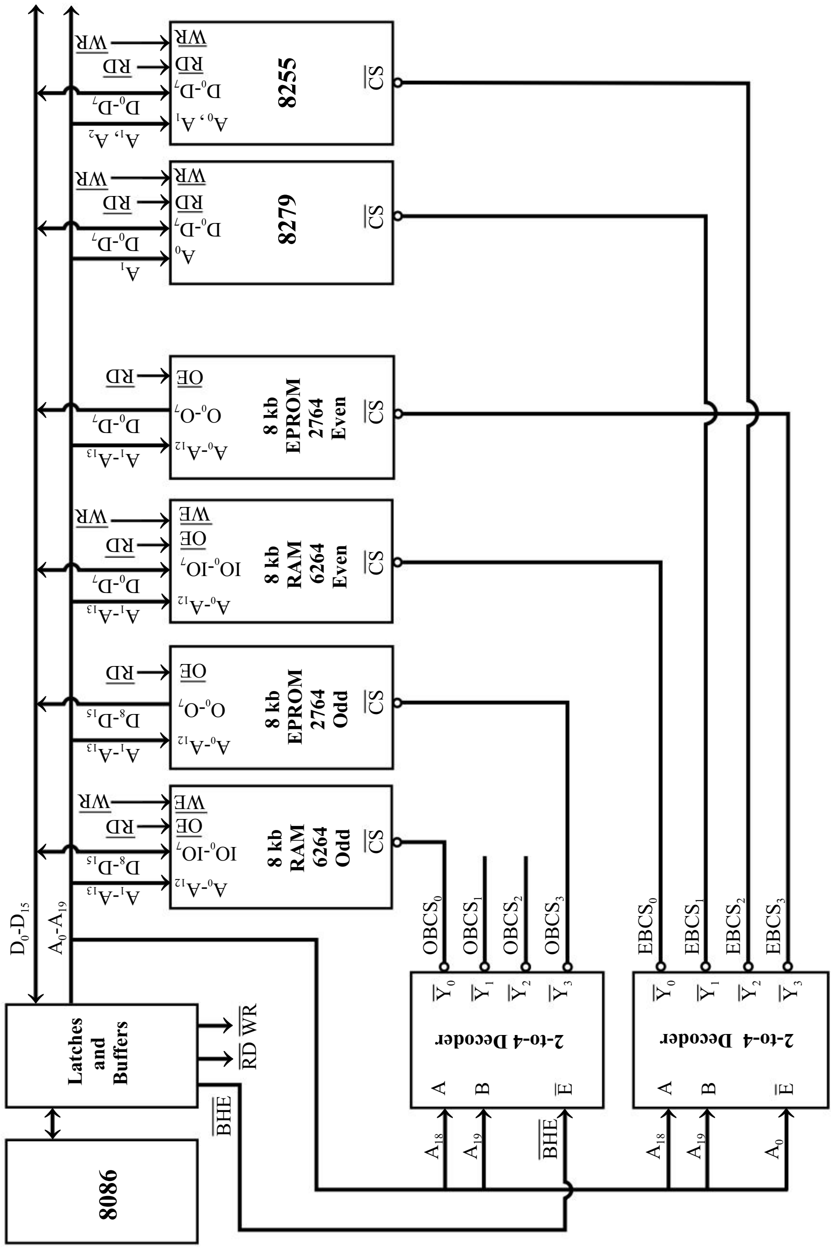


Fig. E1.1 : Memory and I/O device interface diagram for Design Example -1.

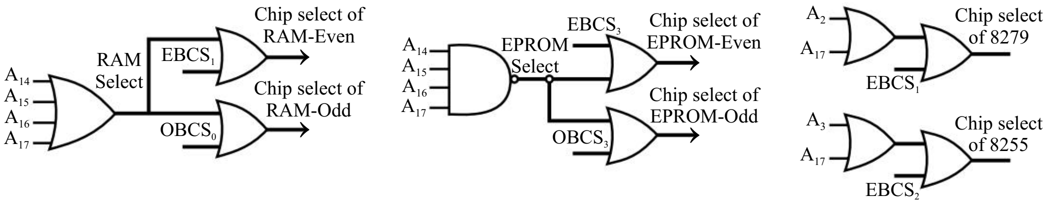


Fig. E1.2 : Additional logic circuits to avoid don't care for memory organization shown in Fig. E1.1.

TABLE - E1 : ADDRESS ALLOCATION TABLE FOR DESIGN EXAMPLE-1

Device	Binary Address																				Hexa Address	
	Decoder Input		Unused Address Lines				Input To Memory/ IO Device Address Pins															
	A_{19}	A_{18}	A_{17}	A_{16}	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}		
8 kb RAM Memory Even	0	0	x	x	x	x	0	0	0	0	0	0	0	0	0	0	0	0	0	0	00000	RAM Address Range 00000_H to $03FFF_H$
	0	0	x	x	x	x	0	0	0	0	0	0	0	0	0	0	0	0	1	0	00002	
	0	0	x	x	x	x	0	0	0	0	0	0	0	0	0	0	0	1	0	0	00004	
	.		.				.			.			.				.				.	
	.		.				.			.			.				.				.	
			.				.			.			.				.				.	
	0	0	x	x	x	x	1	1	1	1	1	1	1	1	1	1	1	1	1	0	03FFE	
8 kb RAM Memory Odd	0	0	x	x	x	x	0	0	0	0	0	0	0	0	0	0	0	0	0	1	00001	
	0	0	x	x	x	x	0	0	0	0	0	0	0	0	0	0	0	0	1	1	00003	
	0	0	x	x	x	x	0	0	0	0	0	0	0	0	0	0	0	1	0	1	00005	
	.		.		.					.			.				.				.	
	.		.		.					.			.				.				.	
	.		.		.					.			.				.				.	
	0	0	x	x	x	x	1	1	1	1	1	1	1	1	1	1	1	1	1	1	03FFF	
8 kb EPROM Memory Even	1	1	x	x	x	x	0	0	0	0	0	0	0	0	0	0	0	0	0	0	FC000	EPROM Address Range $FC000_H$ to $FFFFF_H$
	1	1	x	x	x	x	0	0	0	0	0	0	0	0	0	0	0	0	1	0	FC002	
	1	1	x	x	x	x	0	0	0	0	0	0	0	0	0	0	0	1	0	0	FC004	
	.		.		.						.		.				.				.	
	.		.		.						.		.				.				.	
	.		.		.						.		.				.				.	
	1	1	x	x	x	x	1	1	1	1	1	1	1	1	1	1	1	1	1	0	FC00E	
8 kb EPROM Memory Odd	1	1	x	x	x	x	0	0	0	0	0	0	0	0	0	0	0	0	0	1	FC001	
	1	1	x	x	x	x	0	0	0	0	0	0	0	0	0	0	0	0	1	1	FC003	
	1	1	x	x	x	x	0	0	0	0	0	0	0	0	0	0	0	1	0	1	FC005	
	.		.		.					.			.				.				.	
	.		.		.					.			.				.				.	
	.		.		.					.			.				.				.	
	1	1	x	x	x	x	1	1	1	1	1	1	1	1	1	1	1	1	1	1	FFFFF	
8279 Data Register	0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0	0	40000	
Control Register	0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	1	0	40002	
8255 Port-A	1	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0	0	0	80000	
Port-B	1	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0	1	0	80002	
Port-C	1	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	1	0	0	80004	
Control Register	1	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	1	1	0	80006	

Let us employ two numbers of 2-to-4 decoder to generate chip select signals. Both the decoder, take A_{18} and A_{19} as input, and each decoder produces four decoded output signals. One of the decoder is enabled by address line A_0 and the output of this decoder is used as chip select signals for even bank memory ICs, 8279 and 8255. The other decoder is enabled by the control signal $\overline{BHE}$ and the output of this decoder is used as chip select signals for odd bank memory ICs.

The address lines A_{14} - A_{17} are not used for memory ICs. Similarly, the address lines A_2 - A_{17} are not used for 8279 and the address lines A_3 - A_{17} are not used for 8255.

The addresses allotted to the memory and IO devices are shown in Table-E1. The unused address lines are denoted by "x"(don't care) in Table-E1. While framing hexa address, the don't cares are considered as zero for RAM and IO devices, and they are considered as one for EPROM. This will map RAM memory in the beginning of address space and EPROM at the end of address space, which is a necessary requirement for 8086 system.

The don't care can be avoided by generating additional chip select signals using unused address lines and logically ORing this additional chip select signal with output of decoder and then the combined chip select signal can be used to select the memory and IO devices as shown in Fig. E1.2.

DESIGN EXAMPLE 2

Repeat design example -1 by providing IO mapping for 8279 and 8255.

Solution

The 16 kb EPROM can be implemented in two numbers of 8 kb EPROM-2764 and 16 kb RAM can be implemented in two numbers of 8 kb RAM-6264. The odd bank will consist of 8 kb EPROM and 8 kb RAM, and the even bank will consist of 8 kb EPROM and 8 kb RAM.

The address lines A_1 - A_{13} are used to select the internal locations of memory. The address lines A_{17}, A_{18} and A_{19} are decoded to generate chip select signals for memory ICs. Two numbers of 3-to-8 decoder are employed in the system, one for even bank and the other for odd bank. The even bank decoder is enabled by A_0 and $M/\overline{IO}$. The odd bank decoder is enabled by $\overline{BHE}$ and $M/\overline{IO}$. Memory decoders are enabled when $M/\overline{IO}$ is **high**.

The 8279 and 8255 are IO-mapped in the system. The chip select signals for IO devices are generated by decoding the address lines A_5, A_6 and A_7 using a separate 3-to-8 decoder which is enabled by A_0 and $M/\overline{IO}$. The IO decoder is enabled when $M/\overline{IO}$ is l**ow**.

The address allotted to memory and IO devices are shown in Table-E2. The unused address lines are denoted by don't cares in Table-E2. While framing hexa address, the don't care are considered as zeros for RAM and IO devices, and they are considered as one for EPROM.

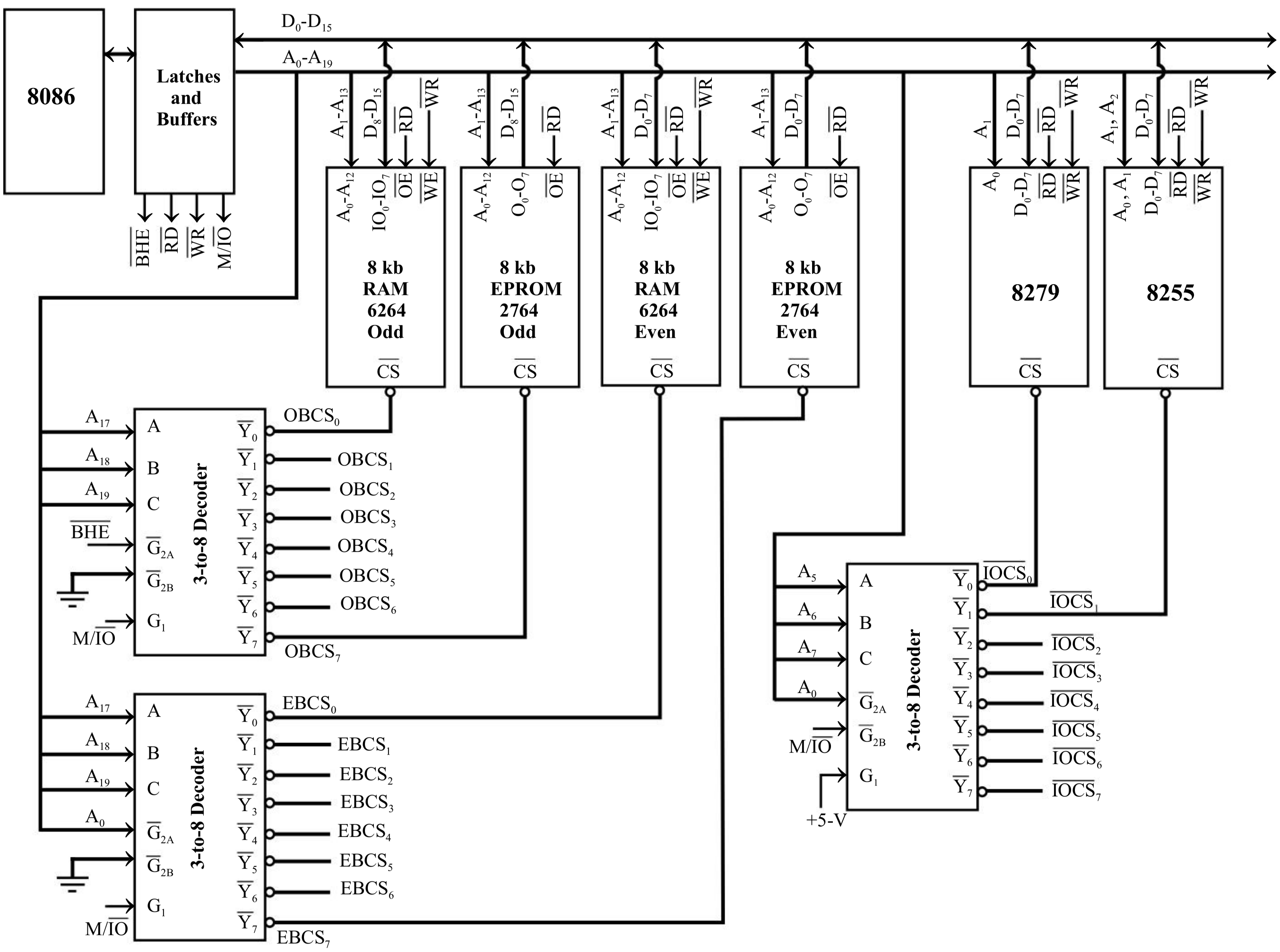


Fig. E2 : Memory and IO device interface diagram for Design Example-2.

TABLE - E2 : ADDRESS ALLOCATION TABLE FOR DESIGN EXAMPLE-2

Device	Binary Address																				Hexa Address	
	Decoder input			Unused address lines			Input address lines to memory IC															
	A_{19}	A_{18}	A_{17}	A_{16}	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		
8 kb RAM Memory Even	0	0	0	x	x	x	0	0	0	0	0	0	0	0	0	0	0	0	0	0	00000	RAM Address Range 00000_H to $03FFF_H$
	0	0	0	x	x	x	0	0	0	0	0	0	0	0	0	0	0	0	1	0	00002	
	0	0	0	x	x	x	0	0	0	0	0	0	0	0	0	0	0	1	0	0	00004	
		⋮			⋮		⋮			⋮				⋮				⋮			⋮	
	0	0	0	x	x	x	1	1	1	1	1	1	1	1	1	1	1	1	1	0	03FFE	
8 kb RAM Memory Odd	0	0	0	x	x	x	0	0	0	0	0	0	0	0	0	0	0	0	0	1	00001	
	0	0	0	x	x	x	0	0	0	0	0	0	0	0	0	0	0	0	1	1	00003	
	0	0	0	x	x	x	0	0	0	0	0	0	0	0	0	0	0	1	0	1	00005	
		⋮					⋮			⋮				⋮				⋮			⋮	
	0	0	0	x	x	x	1	1	1	1	1	1	1	1	1	1	1	1	1	1	03FFF	
8 kb EPROM Memory Even	1	1	1	x	x	x	0	0	0	0	0	0	0	0	0	0	0	0	0	0	FC000	EPROM Address Range $FC000_H$ to $FFFFF_H$
	1	1	1	x	x	x	0	0	0	0	0	0	0	0	0	0	0	0	1	0	FC002	
	1	1	1	x	x	x	0	0	0	0	0	0	0	0	0	0	0	1	0	0	FC004	
		⋮					⋮			⋮				⋮				⋮			⋮	
	1	1	1	x	x	x	1	1	1	1	1	1	1	1	1	1	1	1	1	0	FFFFE	
8 kb EPROM Memory	1	1	1	x	x	x	0	0	0	0	0	0	0	0	0	0	0	0	0	1	FC001	
	1	1	1	x	x	x	0	0	0	0	0	0	0	0	0	0	0	0	1	1	FC003	
	1	1	1	x	x	x	0	0	0	0	0	0	0	0	0	0	0	1	0	1	FC005	
		⋮					⋮			⋮				⋮				⋮			⋮	
	1	1	1	x	x	x	1	1	1	1	1	1	1	1	1	1	1	1	1	1	FFFFE	

Device	Binary Address								Hexa Address
	Decoder input			Input address lines to IO devices					
	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
8279 Data Register	0	0	0	x	x	x	0	0	00
Control Register	0	0	0	x	x	x	1	0	02
8255 Port-A	0	0	1	x	x	0	0	0	20
Port-B	0	0	1	x	x	0	1	0	22
Port-C	0	0	1	x	x	1	0	0	24
Control Register	0	0	1	x	x	1	1	0	26

4.11 SHORT QUESTIONS AND ANSWERS

4.1 ***What is memory ?***

Memory is a storage device in a microprocessor-based system and its primary function is to store programs and data.

4.2 ***Why are semiconductor memories used as the main memory in a microprocessor system ?***

The semiconductor memories alone have processor compatible access time for read and write operations. Therefore, semiconductor memories are used as main memories.

4.3 ***What are the different types of semiconductor memory?***

The different types of semiconductor memory are RAM, PROM, EPROM, static RAM, DRAM and NVRAM.

4.4 ***List the features of semiconductor memories ?***

1. The 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 semiconductor memory are compatible with the microprocessor.

4.5 ***What is meant by volatile and nonvolatile ?***

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 switching OFF the power supply, then the memory is called nonvolatile.

4.6 ***List the volatile and nonvolatile semiconductor memories.***

The volatile semiconductor memories are static RAM and DRAM. The nonvolatile semiconductor memories are ROM, PROM, EPROM and NVRAM.

4.7 ***What are the characteristics of ROM memory ?***

1. It is nonvolatile memory.
2. The contents of ROM memory can be read by the processor and it cannot write into it.
3. The ROM memory has the feature of random access.
4. The memory cell has a MOS transistor either with open gate or closed gate.

4.8 ***How are the ROM memories classified?***

The ROM memories 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.

4.9 ***List the characteristics of EPROM.***

1. EPROM is nonvolatile.
2. It has random access feature.
3. The contents of EPROM can be erased by passing UV light and then the device can be programmed.
4. The EPROM is read only memory and for writing into EPROM, a separate hardware set-up is required.

4.10 ***Write a short note on the memory cell of EPROM.***

The memory cell of EPROM contains a MOS transistor with an 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.

4.11 *What is NVRAM?*

The nonvolatile read/write memories are called NVRAM. The various types of NVRAMs are flash memory, EEPROM and Shadow RAM.

4.12 *List the features of static RAM?*

1. The 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.

4.13 *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.

4.14 *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 transistors.

4.15 *Compare Static RAM and DRAM?*

Static RAM	DRAM
1. Information is stored as voltage level in a flip-flop.	1. Information is stored as a charge in the gate to substrate capacitance.
2. Six to eight transistors are required to form one memory cell.	2. Three to four transistors are required to form one memory cell.
3. Packing density is low.	3. Packing density is high.
4. The contents of memory need not be refreshed.	4. The contents of memory have to be refreshed periodically.

4.16 *What is physical memory space?*

Memory locations that are directly addressed by the microprocessor are called physical memory space.

4.17 *What is memory word size?*

The size of data that can be stored in a memory location is called memory word size.

4.18 *What is meant by memory mapping?*

Memory mapping is the process of interfacing memories to a microprocessor and allocating addresses to each memory locations.

4.19 *What is memory access time?*

Memory access time is the time taken by the processor to read or write a memory location. During read operation, it is the time between a valid address on the bus and end of read control signal. During write operation, it is the time between, a valid address on the bus and end of write control signal.

4.20 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.

4.21 What is bus contention?

If two devices drive the data bus simultaneously, then it is called bus contention. It may lead to the 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.

4.22 Why is EPROM mapped at the end of memory space in 8086 ?

The mapping of EPROM at the end of memory space will facilitate automatic execution of monitor program upon reset. Whenever the processor is reset, the IP is cleared and CS is initialized with $FFFF_H$, and so after a reset the processor will start executing the instructions from $FFFF0_H$ after a reset. The system designer has to permanently store the monitor/boot program starting from this address, which is possible only if EPROM is mapped at the end of memory space.

4.23 What is chip select signal and how is it generated?

Chip select signal is the control signal that has to be asserted TRUE to bring an IC from **high impedance** state to normal state. Generally, chip select signals are generated in a system by decoding the unused address lines with the help of decoders.

4.24 Write the typical control signals involved in EPROM interfacing ?

The control signals needed for EPROM are chip select and output enable.

4.25 Write the typical control signals involved in RAM interfacing ?

The control signals needed for RAM interfacing are chip enable, output enable and write enable.

4.26 What is the relation between memory capacity and address and data pins of memory IC?

If a 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.

4.27 How is memory space organised in 8086?

In 8086, the one mega byte (1 Mb) of addressable memory space is divided into two banks: Even (or lower) memory bank and Odd (or upper) memory bank. Each bank will have an addressable space of 512 kb.

4.28 How are the data lines connected to memory banks in 8086?

In 8086-based system, the lower eight lines of data bus, D_0 - D_7 are connected to even bank memory ICs and the upper eight lines of data bus, D_8-D_{15} are connected to odd bank memory ICs.

4.29 What are the signals involved in memory bank selection?

In 8086-based system, the even bank is selected by the address line A_0 and the odd bank is selected by the control signal $\overline{BHE}$.

4.30 What is a 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.

4.31 What is an interrupt IO?

If the IO device initiate the data transfer through interrupt, then the IO is called interrupt driven IO.

4.32 What is DMA?

The direct data transfer between IO device and memory is called DMA.

4.33 What is the need for Port?

The IO devices are generally slow devices and their timing characteristics do not match with processor timings. Hence, the IO devices are connected to system bus through the ports.

4.34 What is a port?

The port is a buffered IC which is used to hold the data transmitted from the microprocessor to IO device or vice versa.

4.35 Give some examples of port devices used in an 8086 microprocessor-based system ?

The various INTEL IO port devices used in an 8086 microprocessor-based system are 8212, 8155, 8156, 8255, 8355 and 8755.

4.36 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.

4.37 What are the different methods of interfacing IO devices to an 8086-based system.

There are two methods of interfacing IO devices to an 8086 system. They are memory mapping of IO device and standard IO mapping.

4.38 Draw a simple circuit to decode three control signal $\overline{RD}$, $\overline{WR}$ and M/$\overline{IO}$ and to produce separate read/write control signals for memory and IO devices.

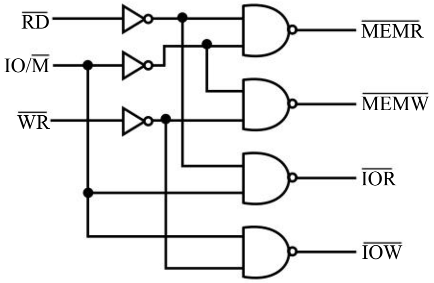


Fig. Q4.38 : Circuit to generate separate read and write signals for memory and IO devices.

4.39 Compare memory-mapped IO and standard IO-mapped IO.

Memory mapping of IO device	IO mapping of IO device
1. 20-bit addresses are provided for IO devices.	1. 8-bit or 16-bit addresses are provided for IO devices.
2. 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 IO device and the processor.	2. Only IN and OUT instructions can be used for data transfer between IO device and the processor.
3. In memory mapped ports, the data can be moved from any register to ports and vice versa.	3. In IO mapped ports, the data transfer can take place only between the accumulator and ports.
4. 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.	4. When IO mapping is used for IO devices, then the full memory address space can be used for addressing memory. Hence, it is suitable for systems which require large memory capacity.
5. For accessing the memory mapped devices, the processor executes memory read or write cycle. During this cycle $M/\overline{IO}$ is asserted **high**.	5. For accessing the IO mapped devices, the processor executes IO read or write cycle. During this cycle $M/\overline{IO}$ is asserted **low**.

4.40 What is the drawback in memory-mapped IO?

When IO devices are memory-mapped, some of the addresses are allotted to IO devices and 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.

CHAPTER 5

INTERRUPTS

5.1 INTERRUPT AND ITS NEED

The microprocessors allow normal program execution to be interrupted in order to carry out a specific task/work. The processor can be interrupted in the following ways:

1. By an external signal generated by a peripheral.
2. By an internal signal generated by a special instruction in the program.
3. 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 current normal program, saves the status (or content) of various registers (IP, CS and flag registers in case of 8086) in stack and then the processor 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 **I**nterrupt **S**ervice **S**ubroutine (ISS). 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 the 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

The 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 stack and executes an ISS to perform the data transfer between the peripheral and processor. At the end of ISS the processor status is restored from stack and 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 the polling technique or by the interrupt method. In the 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 milliseconds 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.

5.2 CLASSIFICATION OF INTERRUPTS

In general, interrupts can be classified in the following three ways:

1. Hardware and software interrupts.
2. Vectored and non-vectored interrupts.
3. Maskable and non-maskable interrupts.

The interrupts initiated by external hardware by sending an appropriate signal to the interrupt pin of the processor is called hardware interrupt. The 8086 processor has two interrupt pins: INTR and NMI. The interrupts initiated by applying appropriate signal to these pins are called hardware interrupts of 8086.

The 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 8086 processor has 256 types of software interrupts. The software interrupt instruction is INT n, where n is the type number in the range 255_{10}.

When an interrupt signal is accepted by the processor, if 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 processors. [In these vector addresses the **I**nterrupt **S**ervice **S**ubroutines(ISS) are stored.] In nonvectored interrupts the interrupting device should supply the address of the ISS to be executed in response to the interrupt. All the 8086 interrupts are vectored interrupts. The vector address for an 8086 interrupt is obtained from a vector table implemented in the first 1kb memory space.

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 8086, the **I**nterrupt **F**lag(IF) can be set to one to unmask or enable all hardware interrupts and IF is cleared to zero to mask or disable all hardware interrupts except NMI.

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 (or cannot be rejected) by the processor are called nonmaskable interrupts. Whenever a request is made by nonmaskable interrupt, the processor has to definitely accept that request and service that interrupt by suspending its current program and executing an ISS. In an 8086 processor, all the hardware interrupts initiated through INTR pin are maskable by clearing **I**nterrupt **F**lag(IF). The interrupt initiated through NMI pin and all software interrupts are nonmaskable.

5.3 SOURCES OF INTERRUPTS IN 8086

An interrupt in 8086 can come from one of the following three sources:

1. One source is from an external signal applied to NMI or INTR input pin of the processor. The interrupts initiated by applying appropriate signals to these input pins are called hardware interrupts.
2. A second source of an interrupt is execution of the interrupt instruction "INT n", where n is the type number. The interrupts initiated by "INT n" instructions are called software interrupts.
3. The third source of an interrupt is from some condition produced in the 8086 by the execution of an instruction. An example of this type of interrupt is divide by zero interrupt. Program execution will be automatically interrupted if you attempt to divide an operand by zero. Such conditional interrupts are also known as exceptions.

5.4 INTERRUPTS OF 8086

The 8086 microprocessor has 256 types of interrupts which come from any one of the three sources mentioned above. INTEL has assigned a type number to each interrupt. The type numbers are in the range of 0 to 255_{10}. The 8086 processor has dual facility of initiating these 256 interrupts. The interrupts can be initiated either by executing "INT n" instruction where n is the type number or the interrupt can be initiated by sending an appropriate signal to INTR input pin of the processor.

For the interrupts initiated by software instruction "INT n", the type number is specified by the instruction itself. When the interrupt is initiated through INTR pin, then the processor runs an interrupt acknowledge cycle to get the type number (i.e., the interrupting device should supply the type number through D_0-D_7 lines when the processor requests for the same through an interrupt acknowledge cycle).

In these 256 interrupts, INTEL has defined the functions of the first five interrupts, i.e., the interrupts type-0 to type-4 are dedicated for specific functions by INTEL and they are called INTEL predefined interrupts. The next 27 interrupts, i.e., from type-5 to type-31 are reserved by INTEL for use in future microprocessors or for system calls/services. The upper 224 interrupts, i.e., from type-32 to type-255 are available for the user as hardware or software interrupts.

5.4.1 INTEL Predefined (or Dedicated) Interrupts

The INTEL predefined interrupts are :

1. Division by zero (Type-0 interrupt).
2. Single step (Type-1 interrupt).
3. Nonmaskable interrupt, NMI (Type-2 interrupt).
4. Breakpoint interrupt (Type-3 interrupt).
5. Interrupt on overflow (Type-4 interrupt).

The predefined interrupts are only defined by INTEL and INTEL has not provided any subroutine/ procedure to be executed for these interrupts. To use the predefined interrupts the user/system designer has to write **I**nterrupt **S**ervice **S**ubroutine (ISS) for each interrupt and store them in memory. The corresponding address of the ISS should be stored in interrupt vector table. If a predefined interrupt is not used in a system then the user may assign some other functions to these interrupts.

Divide by zero interrupt (type-0 interrupt)

Type-0 interrupt is implemented by INTEL as a part of the execution of the divide instruction. The 8086 will automatically do a type-0 interrupt if the result of a division operation is too large to fit in the destination register and this interrupt is nonmaskable. Since the type-0 interrupt cannot be disabled in any way, we have to account for it in the programs using divide instructions. To account for this, we have to write an ISS which takes the desired action or indicate error condition when an invalid division occurs. The ISS should be stored in memory and the address of ISS is stored in interrupt vector table.

Single step interrupt (type-1 interrupt)

When the **T**rap/**T**race **F**lag (TF) is set to one, the 8086 processor will automatically generate a type-1 interrupt after execution of each instruction. The user can write an ISS for type-1 interrupt to halt the processor temporarily and return the control to the user so that after execution of each instruction, the processor status (content of register/memory) can be verified. If they are correct then we can proceed to execute the next instruction. Execution of one instruction by one instruction is known as single step and this feature will be useful to debug a program.

Nonmaskable interrupt, NMI (type-2 interrupt)

The 8086 processor will automatically generate a type-2 interrupt when it receives a **low-to-high** transition on its NMI input pin. This interrupt cannot be disabled or masked. Usually, the type-2 interrupt is used to save program data or processor status in case of system ac power failure.

The ac power failure is detected by an external hardware and whenever the ac power fails, the external hardware will send an interrupt signal to the NMI input pin of the processor. The rectifier which converts ac to dc usually has a large filter capacitor and so it can retain the dc power for atleast 50 milliseconds, after the ac power supply is interrupted. This 50 milliseconds time will be sufficient to run an ISS by type-2 interrupt to save program data or processor status to NVRAM or RAM with battery back-up power supply.

Breakpoint interrupt (type-3 interrupt)

Type-3 interrupt is used to implement a breakpoint function, which executes a program partly or up to the desired point and then return the control to the user.

The breakpoint interrupt is initiated by the execution of "INT 3" instructions. To implement the breakpoint function the system designer has to write an ISS for type-3, which takes care of displaying a message and return the control to the user whenever type-3 interrupt is initiated.

This interrupt will be useful to debug a program by executing the program part by part. The user can insert "INT 3" instruction at the desired location and execute the program. Whenever

"INT 3" instruction is encountered, the processor halts the program execution and return the control to the user. Now the user can verify the processor status (contents of register/memory). If they are correct then the user can proceed to execute next part of the program.

Overflow interrupt (**type-4 interrupt**)

In the 8086 processor, the **O**verflow **F**lag (OF) will be set if the signed arithmetic operation generates a result whose size is larger than the size of the destination register/memory. During such conditions, the type-4 interrupt can be used to indicate an error condition. The type-4 interrupt is initiated by "INTO" instruction.

One way of detecting the overflow error is to put the INTO instruction immediately after the arithmetic instruction in the program. After arithmetic operation if the overflow flag is not set then the processor will consider "INTO" instruction as NOP (**N**o **op**eration). However, if the overflow flag is set then the 8086 will generate a type-4 interrrupt, which executes an ISS to indicate overflow condition.

5.4.2 Software Interrupts of 8086

The "INT n" instructions are called software interrupts. The "INT n" instruction will initiate type-n interrupt, and the value of n is in the range of 0 to 255_{10}. Therefore, all the 256 type interrupts including the INTEL predefined and reserved interrupts can be initiated through "INT n" instruction. The software interrupts are nonmaskable and has higher priority than hardware interrupts.

5.4.3 Hardware Interrupts of 8086

The interrupts initiated by applying appropriate signals to INTR and NMI pins of 8086 are called hardware interrupts. All the 256 types of interrupts including INTEL predefined and reserved interrupts can be initiated by applying a **high** signal to INTR pin of 8086. When a **high** signal is applied to the INTR pin and the hardware interrupt is enabled/unmasked, then the processor runs an interrupt acknowledge cycle to get the type number of the interrupt from the device which sends the interrupt signal. The interrupting device can send a type number in the range of 0 to 255_{10}. Therefore, all the 256 types of interrupts can be initiated through INTR pin.

The hardware interrupts initiated through INTR are maskable by clearing the **I**nterrupt **F**lag, (IF), i.e., the hardware interrupts are masked/disabled when IF = 0 and they are unmasked/enabled when IF = 1. The interrupts initiated through INTR has lower priority than software interrupts.

The hardware interrupt NMI is nonmaskable and has higher priority than interrupts initiated through INTR. The NMI is initiated by a rising edge (or low-to-high transition) of the signal applied to NMI pin of the processor. The processor will execute type-2 interrupt in response to interrupt initiated through NMI pin and this type number is fixed by INTEL. The external device, interrupting the processor through NMI pin, need not supply the type number for this interrupt.

5.4.4 Priorities of Interrupts of 8086

The priorities of the interrupts of 8086 are shown in Table-5.1. The 8086 processor checks for internal interrupts before it checks for any hardware interrupt. Therefore, software interrupts has higher priority than hardware interrupts. But the processor can accept the NMI interrupt request and execute a procedure for it even in between the execution of procedure for higher priority interrupt.

For example, if the NMI is initiated by an external hardware while the processor internally generates the divide error interrupt, then the processor goes to start of divide error procedure and then suspends it to service NMI. Only after servicing the NMI, the processor will complete the divide error procedure.

TABLE - 5.1 : INTERRUPT PRIORITY

Interrupt	Priority
Divide error, INT n, INTO	Highest
NMI	
INTR	↓
SINGLE STEP	Lowest

5.5 IMPLEMENTING INTERRUPT SCHEME IN 8086

The 8086 processor has 256 types of interrupts and these interrupts can be implemented either as hardware or software interrupts. The number of interrupts to be implemented and used in a system depends on the system designer and also on the application for which the system is designed. The choice of implementing the INTEL predefined interrupts also depends on the system designer.

Except some of the INTEL predefined interrupts, for all other interrupts the system designer has to decide on the method of initiating the interrupts selected to implement on a system. The interrupts can be initiated either by external hardware or internally by software instruction "INT n". In a system some interrupt types are chosen to be initiated by the hardware, some other interrupt types are chosen to be initiated by software and some of the interrupts are left unused. The unused interrupts can be implemented by the user for user-defined functions.

5.5.1 Interrupt Vector Table

For each and every interrupt decided to be implemented in the system, the system designer has to write an **I**nterrupt **S**ervice **S**ubroutine (ISS) and store them in memory. Then the system designer has to create an interrupt vector table in the first 1kb memory space (i.e., in the memory space with address range 00000_H to $003FF_H$) of the 8086 system. In this vector table, the 16-bit offset address and 16-bit segment base address of each ISS are stored in four consecutive memory locations. The address stored in this table are called vector addresses. For storing the vector addresses of all the 256 interrupt types, the vector table requires 1 kb (256 × 4 = 1 kb) memory space.

The memory address for storing the vector address for an interrupt is given by multiplying the type number by four and sign extending it to 20-bit. The vector address for an interrupt is stored in four consecutive memory location starting from this 20-bit address. The first two locations are used to store the low byte and high byte of offset address, and next two locations are used to store low byte and high byte of segment base address of ISS to be executed for an interrupt. The organization of interrupt vector table of 8086-based system is shown in Fig. 5.1.

5.5.2 Servicing an Interrupt By 8086

The 8086 processor checks for interrupt request at the end of each instruction cycle. If an interrupt request is deducted, then the 8086 processor responds to the interrupt by performing the following operations:

1. The SP is decremented by two and the content of flag register is pushed to stack memory.
2. The interrupt system is disabled by clearing **I**nterrupt **F**lag (IF).
3. The single-step trap flag is disabled by clearing **T**rap **F**lag (TF).

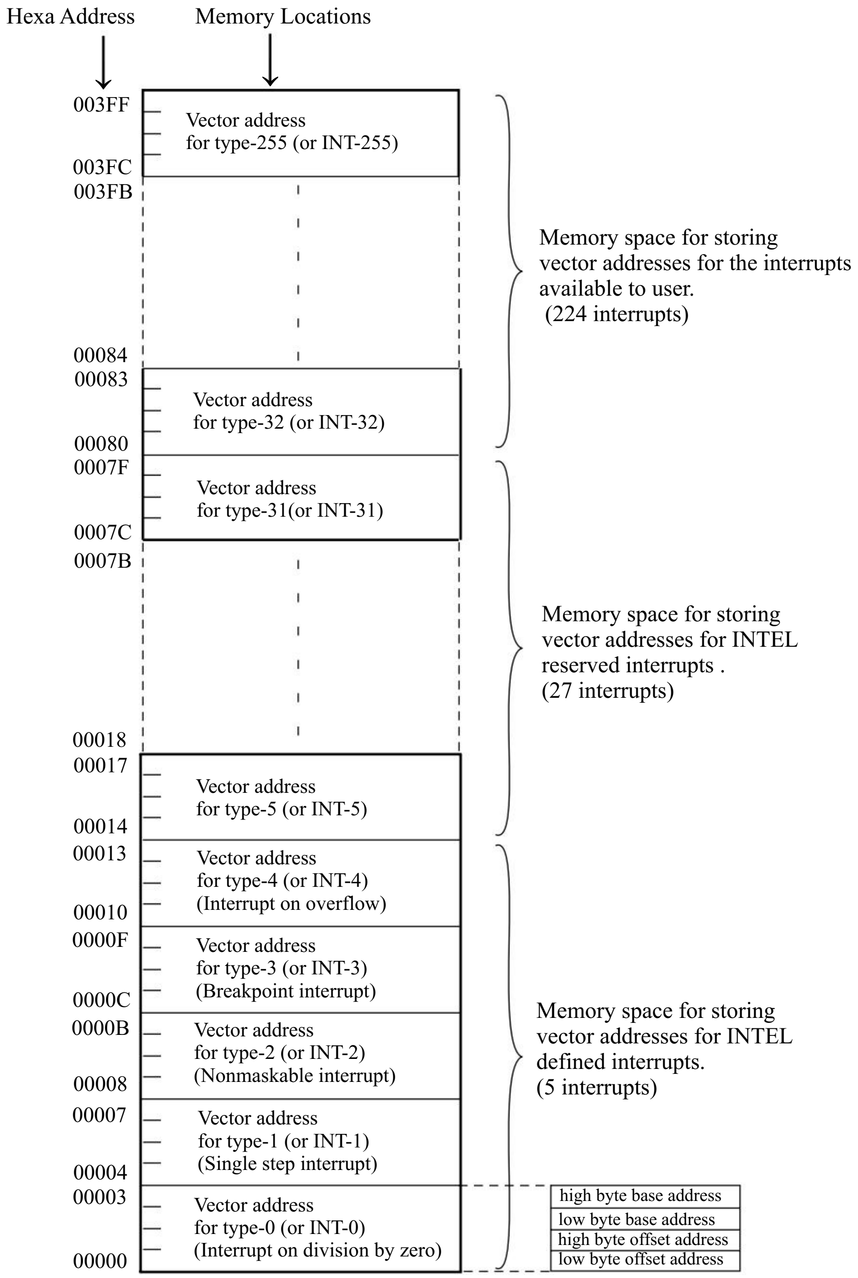

Hexa Address
Memory Locations
003FF
003FC
003FB
00084
00083
00080
0007F
0007C
0007B
00018
00017
00014
00013
00010
0000F
0000C
0000B
00008
00007
00004
00003
00000
Vector address for type-255 (or INT-255)
Vector address for type-32 (or INT-32)
Vector address for type-31(or INT-31)
Vector address for type-5 (or INT-5)
Vector address for type-4 (or INT-4) (Interrupt on overflow)
Vector address for type-3 (or INT-3) (Breakpoint interrupt)
Vector address for type-2 (or INT-2) (Nonmaskable interrupt)
Vector address for type-1 (or INT-1) (Single step interrupt)
Vector address for type-0 (or INT-0) (Interrupt on division by zero)
Memory space for storing vector addresses for the interrupts available to user. (224 interrupts)
Memory space for storing vector addresses for INTEL reserved interrupts . (27 interrupts)
Memory space for storing vector addresses for INTEL defined interrupts. (5 interrupts)
high byte base address
low byte base address
high byte offset address
low byte offset address

Fig. 5.1 : Organization of interrupt vector table in 8086.

4. The stack pointer is decremented by two and the content of CS-register is pushed to stack memory.
5. Again, the stack pointer is decremented by two and the content of IP is pushed to stack memory.
6. In case of hardware interrupt through INTR, the processor runs an interrupt acknowledge cycle to get the interrupt type number. For software interrupts, the type number is specified in the instruction itself. For NMI and exceptions the type number is defined by INTEL.
7. The processor generates a 20-bit memory address by multiplying the type number by four and sign extending it to 20-bit. This memory address is the address of the interrupt vector table, where the vector address of the **I**nterrupt **S**ervice **S**ubroutine (ISS) is stored by the user/system designer.
8. The first word pointed by vector table address is loaded in IP and the next word is loaded in CS-register. Now the content of the IP is the offset address and the content of the CS-register is the segment base address of the ISS to be executed.
9. The 20-bit physical memory address of ISS is calculated by multiplying the content of CS-register by 16_{10} and adding it to the content of IP.
10. The processor executes the ISS to service the interrupt.
11. The ISS will be terminated by the IRET instruction. When this instruction is executed, the top of stack is poped to IP, CS and flag register one word by one word. After every pop operation, the SP is incremented by two.
12. Thus, at the end of ISS, the previous status of the processor is restored and so the processor will resume the execution of normal program from the instruction where it was suspended.

5.6 INTR AND ITS EXPANSION

The hardware interrupt INTR can be used by any external device to interrupt the processor. When an interrupt request is made through the INTR and the INTR interrupt is enabled/unmasked then the processor will run an interrupt acknowledge cycle. During this cycle, the processor asserts $\overline{\text{INTA}}$ signal twice. The first $\overline{\text{INTA}}$ signal is to inform the interrupting device about the acceptance of the interrupt. The second time, $\overline{\text{INTA}}$ is asserted to request the interrupting device to supply the type number and to read the type number from the low order data bus. Therefore, the processor expects a type number on the low order data bus whenever it is interrupted through the INTR input pin.

A scheme for loading the type number on low order data bus is shown in Fig. 5.2. In this scheme the desired type number is applied to the input of the tristate octal buffer through switch settings and the buffer is enabled by the $\overline{\text{INTA}}$ signal. The output of the buffer is connected to low order data bus. Hence, whenever the buffer is enabled by $\overline{\text{INTA}}$ signal, the type number available on its input pins are transferred to its output lines. Thus, the type number is loaded on the low order data bus.

In the schematic shown in Fig. 5.2 the switches can be manually set to create a binary type number in the range $0000\ 0000_2$ to $1111\ 1111_2$ corresponding to 0_{10} to 255_{10}. When a switch is open, the voltage applied to the corresponding input pin of tristate buffer is +5-V and so it is logic-1. When a switch is closed, the voltage applied to the corresponding input pin of tristate buffer is zero volt and so it is logic-0. Thus, by closing/opening the switches an 8-bit binary number can be created which is the desired type number.

The scheme shown in Fig. 5.2 can be used to implement only one interrupt and so only one external device can interrupt the processor. But the INTR interrupt can be used to initiate all the 256 type interrupts. To initiate multiple interrupts through INTR, there should be some provision to supply different type numbers for various interrupts. Such a scheme is possible with programmable interrupt controller INTEL 8259.

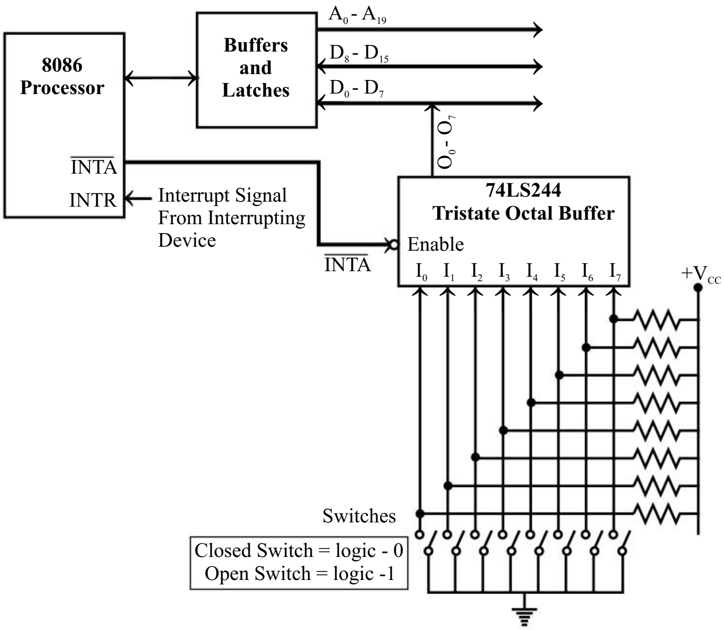


Fig. 5.2 : A schematic to load any type number on low order data bus (D_0 - D_7) in response to the $\overline{INTA}$.

The programmable interrupt controllers can be interfaced to the 8086 processor to handle multiple interrupt requests and allow one by one to the processor INTR input pin. One interrupt controller can accept up to eight interrupt requests and allow one by one to the processor. Multiple interrupt controllers can be interfaced to the processor in cascaded mode, to handle up to 64 interrupt requests. In the cascaded mode, one master interrupt controller and a maximum of eight slave interrupt controllers can be interfaced to the processor to handle 64 interrupt requests and allow one by one to the processor INTR input pin.

A detailed discussion on the programmable interrupt controller, 8259 and its interfacing with 8086 processor are presented in the following sections.

5.7 PROGRAMMABLE INTERRUPT CONTROLLER - INTEL 8259

The 8259 is a programmable interrupt controller. It is used to expand the interrupts of the 8085 or 8086 processor. One 8259 can accept eight interrupt requests and allow one by one to the processor INTR pin. The interrupt controller can be used in cascaded mode in a system to expand the interrupts up to 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. In an 8086 processor-based system, it supplies the type number of the interrupt and the type number is programmable. In an 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 decide the priorites are automatic rotation mode, specific rotation mode and fully nested mode.
5. The 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 the pending interrupts, masked interrupts and the interrupt being serviced.
8. The 8259s can be cascaded to accept a maximum of 64 interrupts.

5.7.1 Interfacing 8259 With 8086 Microprocessor

The 8259 is a 28-pin IC packed in DIP. The various pins of 8259 are shown in Fig. 5.3. It requires two internal address 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 8086 is shown in Fig. 5.4. In Fig. 5.4, 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_1 of the 8086 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_5, A_6 and A_7 are used as input to decoder. The address line A_0 and control signal $M/\overline{IO}$ are used as logic low enables for the decoder. The IO addresses of 8259 are shown in Table-5.2. The signals CAS_0-CAS_2 are used only in cascade operation of 8259s.

The $\overline{SP}/\overline{EN}$ pin can be used as input or output signal. In nonbuffered 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 the data is transferred from 8259A to the CPU.

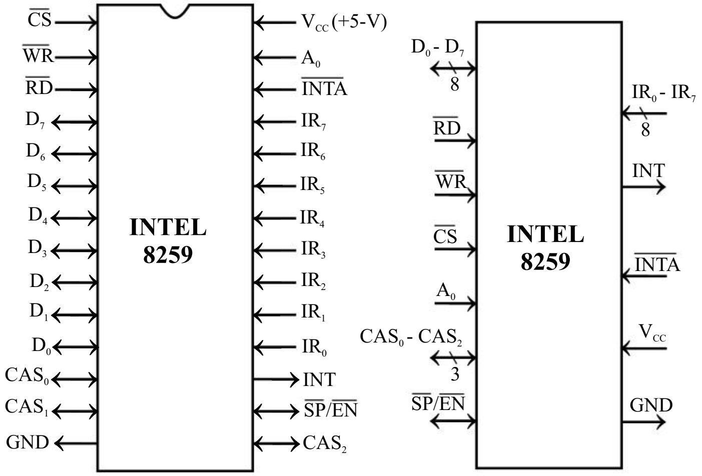
Working of 8259 With 8086 Processor

First the 8259 should be programmed by sending **I**nitialization **C**ommand **W**ord (ICW) and **O**perational **C**ommand **W**ord (OCW). These command words will inform 8259 about the following:

- Type of interrupt signal (Level triggered/Edge trigerred).
- Type of processor (8085/8086).
- Call address and its interval (4 or 8).
- Masking of interrupts.
- Priority of interrupts.
- Type of end of interrupt.

Once 8259 is programmed, it is ready for accepting an interrupt signal. When it receives an interrupt through any one of the interrupt lines IR_0-IR_7, 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 highest priority and unmasked, then it is serviced.

For servicing this interrupt the 8259 will send INT signal to INTR pin of 8086. In response it expects an acknowledge $\overline{INTA}$ from the processor. When the processor accepts the interrupt, it sends two $\overline{INTA}$ one by one. The first $\overline{INTA}$ is send to 8259, to inform the acceptance of interrupt and to prepare 8259 for supplying type number. The second $\overline{INTA}$ is send to 8259 to read the type number from the 8259. Once the processor receives the type number, it starts processing the interrupt corresponding to this type number. The 8086 processor multiplies the type number by four and sign extends to 20-bit to generate a 20-bit vector table address. From this vector table, the vector addresses of the interrupt type requested are read and loaded in IP and CS-register. Then the processor executes the ISR stored in this address.



Pins	Description
D_0-D_7	Bidirectional datalines
RD	Read control
WR	Write control
A_0	Internal address
CS	Chip select
CAS_0-CAS_2	Cascade lines
SP/EN	Slave program /Enable buffer
INT	Interrupt output
INTA	Interrupt acknowledge input
IR_0-IR_7	Interrupt request inputs

Fig. 5.3 : Pin details of 8259.

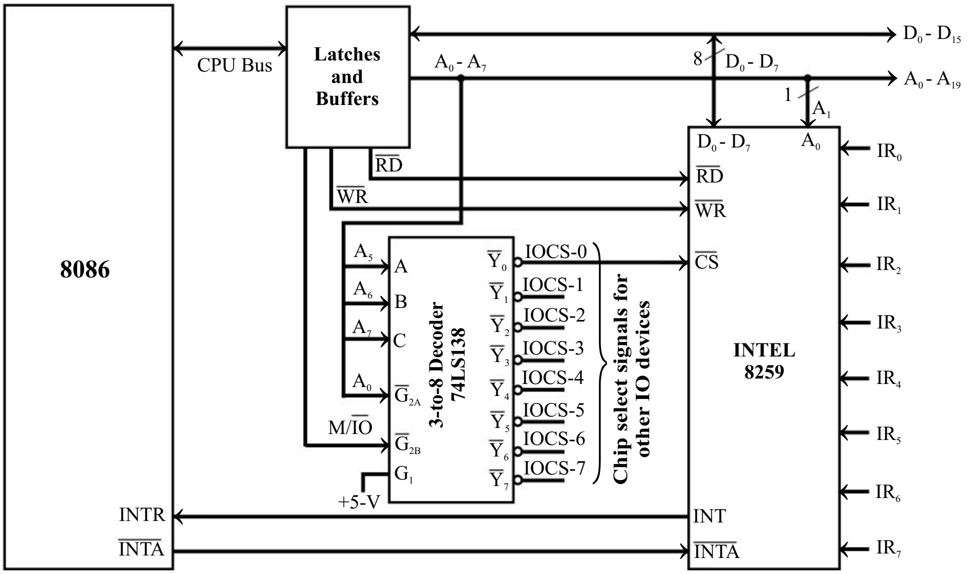


Fig. 5.4 : Interfacing 8259 to 8086 microprocessor.

TABLE - 5.2 : IO ADDRESS OF 8259

	Binary address								Hexa address
	Decoder input			Input to address pin of 8259				Decoder enable	
	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
For A_0 of 8259 to be zero	0	0	0	x	x	x	0	0	00
For A_0 of 8259 to be one	0	0	0	x	x	x	1	0	02

Note : Don't care "x" is considered as zero.

5.7.2 Functional Block Diagram of 8259

The functional block diagram of 8259 is shown in Fig. 5.5 and it shows eight functional blocks. They are Control logic, Read/Write logic, Data bus buffer, **I**nterrupt **R**equest **R**egister(IRR), **I**n-**S**ervice **R**egister(ISR), **I**nterrupt **M**ask **R**egister(IMR), **P**riority **R**esolver(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.

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 send by the processor through OCW1. The in-service register keeps a track of which interrupt input is 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 in-service registers and determines whether INT signal should be sent to the processor or not.

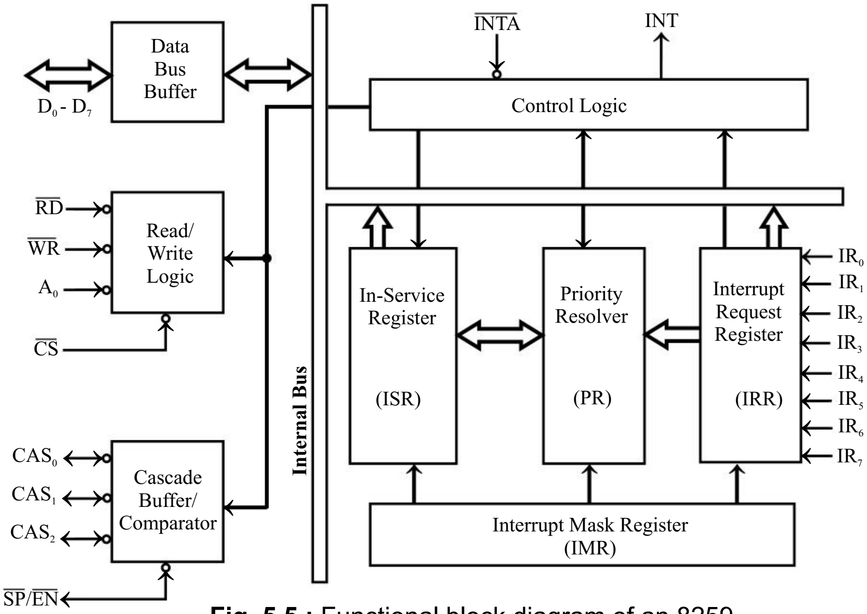


Fig. 5.5 : Functional block diagram of an 8259.

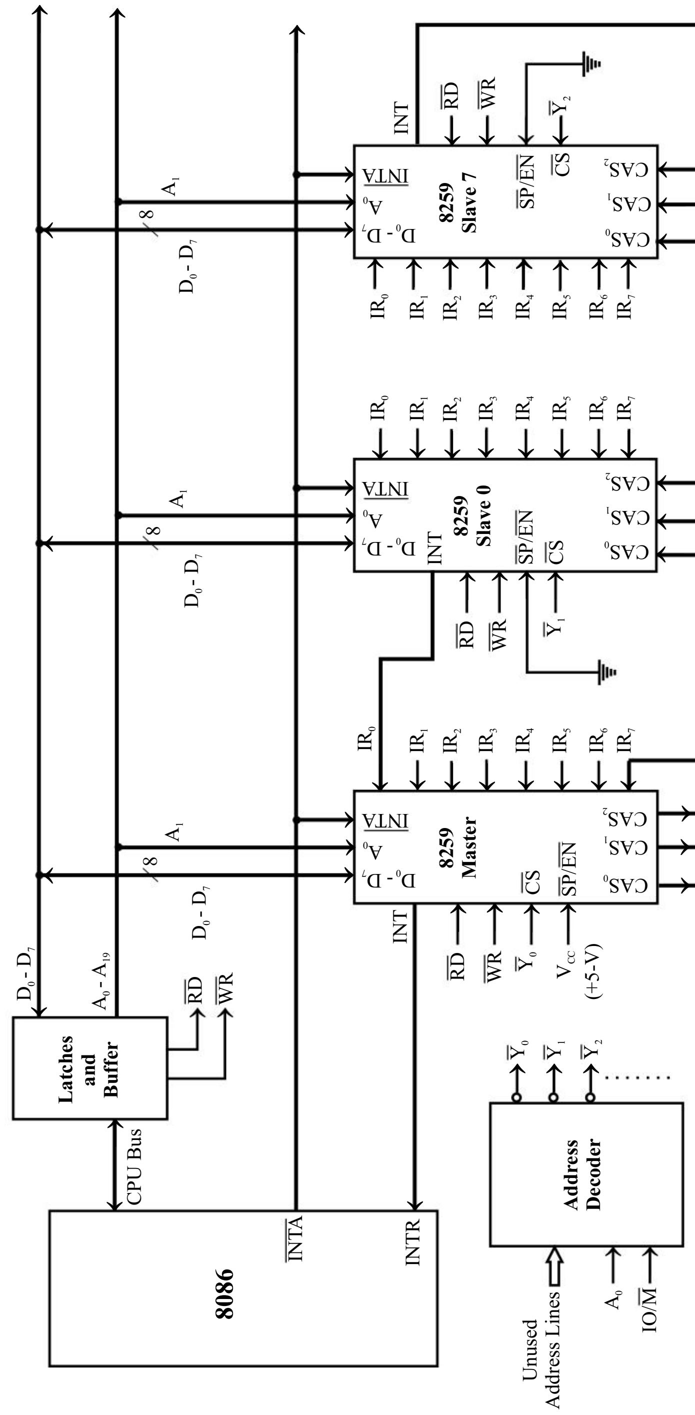


Fig. 5.6 : Example of cascade connection of programmable interrupt controllers-8259.

The cascade buffer/comparator is used to expand the interrupts of 8259. Figure 5.6 is an example of 8259s in cascade connection. In this configuration, one 8259 will be directly interrupting 8086 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 addresses so that each 8259 can be programmed independently by sending command words and independently the status 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.

5.7.3 Processing of Interrupts By 8259

To implement interrupts, the processor interrupt should be enabled and 8259 is initialized. The 8259 is initialized by sending ICWs and OCWs. The ICWs are used to set up the proper conditions and to specify interrupt type number. 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. 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 two $\overline{INTA}$ signals one by one.
4. When the first $\overline{INTA}$ 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.
5. When the 8259 receives the second $\overline{INTA}$, it places the type number on the data bus.
6. Once the processor reads the type number from 8259, the bit corresponds to the current interrupt being serviced in the ISR should be reset to allow the next interrupt. This is done automatically if the 8259 is programmed for **A**utomatic **E**nd Of **I**nterrupt (AEOI). Alternatively the processor can send command word at the end of interrupt service routine to inform 8259 about the end of interrupt.
7. The 8086 processor multiplies the type number by four to generate a vector table address and from vector table, the processor reads the vector address of the interrupt type and loads in IP and CS-register. Then the processor starts executing the ISS.

5.7.4 Programming (or Initializing) 8259

The 8259 has four numbers of **I**nitialization **C**ommand **W**ord (ICW) and three numbers of **O**perational **C**ommand **W**ord (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 the ICWs. The OCW2 can be sent at any time (either before servicing interrupt or at the end of interrupt service routine). The order of sending ICWs and OCWs are shown as flowchart in Fig. 5.7. The format of ICWs and OCWs are shown in Fig. 5.8 and Fig. 5.9.

The ICWs are used to program the following features of 8259:

- Call address interval in case of 8085.
- 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 Nonspecific end of interrupt.
- Priority modes.

A brief discussion about ICWs and OCWs are presented in the following sections.

Initialization command words (ICWs)

The 8259A has four ICWs and they are named as ICW1, ICW2, ICW3 and ICW4. When only one 8259 is used in the system then 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.

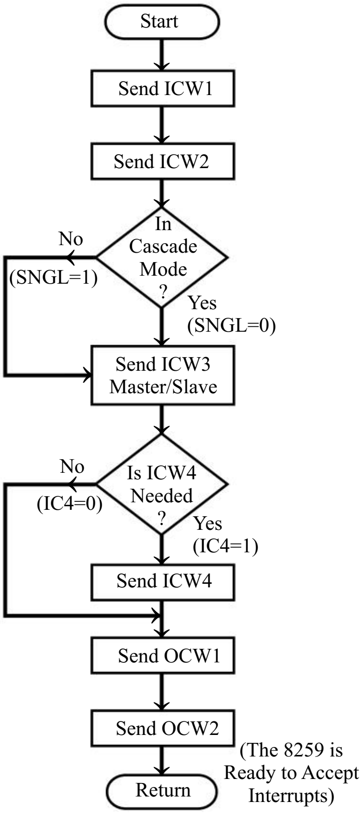


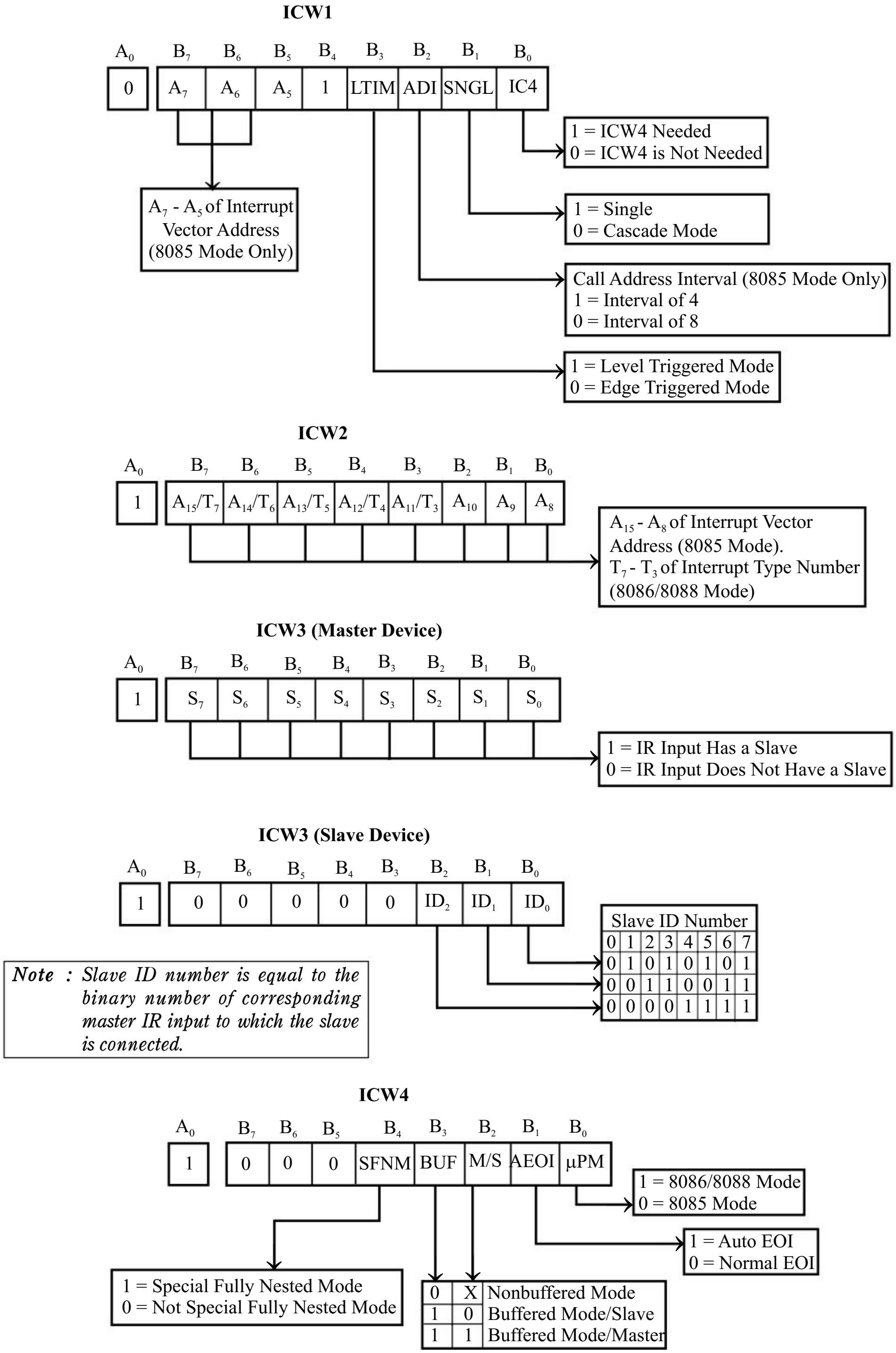
Fig. 5.7 : Sending order of ICWs and OCWs.

ICW1 : The ICW1 programs the basic operation of 8259. In the 8086 processor-based system the bits ADI, A_7, A_6 and A_5 in the format of ICW1 are don't cares.(These fields are applicable only for 8085 processor-based system.) For an 8086-based system we have to set "IC4" bit to one. The single or cascade mode of operation is selected by programming the "SNGL" bit. The LTIM bit determines whether the interrupt request input is positive edge-triggered or level-triggered.

ICW2 : In 8086, the ICW2 is used to program the interrupt type number and associate an interrupt type number to interrupt request IR_0 to IR_7. The lower three bits of type number are automatically inserted by 8259 and the upper five bits are programmable. The binary code inserted in the lower three bits for interrupt request IR_0 to IR_7 are 000 to 111.

For example, if the bits T_3 to T_7 are chosen as 10010 then the following interrupt type numbers are associated with IR_0 to IR_7. For any interrupt request input through IR_0-IR_7 lines, the associated interrupt type is executed by the processor.

IR_0 is associated with type-90_H interrupt (90_H = 1001 0000)
IR_1 is associated with type-91_H interrupt (91_H = 1001 0001)
IR_2 is associated with type-92_H interrupt (92_H = 1001 0010)
IR_3 is associated with type-93_H interrupt (93_H = 1001 0011)
IR_4 is associated with type-94_H interrupt (94_H = 1001 0100)
IR_5 is associated with type-95_H interrupt (95_H = 1001 0101)
IR_6 is associated with type-96_H interrupt (96_H = 1001 0110)
IR_7 is associated with type-97_H interrupt (97_H = 1001 0111)


ICW1
A0 B7 B6 B5 B4 B3 B2 B1 B0
0 A7 A6 A5 1 LTIM ADI SNGL IC4
1 = ICW4 Needed
0 = ICW4 is Not Needed
1 = Single
0 = Cascade Mode
Call Address Interval (8085 Mode Only)
1 = Interval of 4
0 = Interval of 8
1 = Level Triggered Mode
0 = Edge Triggered Mode
A7 - A5 of Interrupt Vector Address (8085 Mode Only)
ICW2
A0 B7 B6 B5 B4 B3 B2 B1 B0
1 A15/T7 A14/T6 A13/T5 A12/T4 A11/T3 A10 A9 A8
A15 - A8 of Interrupt Vector Address (8085 Mode).
T7 - T3 of Interrupt Type Number (8086/8088 Mode)
ICW3 (Master Device)
A0 B7 B6 B5 B4 B3 B2 B1 B0
1 S7 S6 S5 S4 S3 S2 S1 S0
1 = IR Input Has a Slave
0 = IR Input Does Not Have a Slave
ICW3 (Slave Device)
A0 B7 B6 B5 B4 B3 B2 B1 B0
1 0 0 0 0 0 ID2 ID1 ID0
Slave ID Number
Note : Slave ID number is equal to the binary number of corresponding master IR input to which the slave is connected.
ICW4
A0 B7 B6 B5 B4 B3 B2 B1 B0
1 0 0 0 SFNM BUF M/S AEOI µPM
1 = 8086/8088 Mode
0 = 8085 Mode
1 = Auto EOI
0 = Normal EOI
1 = Special Fully Nested Mode
0 = Not Special Fully Nested Mode

Slave ID Number							
0	1	2	3	4	5	6	7
0	1	0	1	0	1	0	1
0	0	1	1	0	0	1	1
0	0	0	0	1	1	1	1

0	X	Nonbuffered Mode
1	0	Buffered Mode/Slave
1	1	Buffered Mode/Master

Fig. 5.8 : Format of ICWs.

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 an 8086-based system the right most bit is set to one. The AEOI bit is used to program the method of terminating the interrupt. If AEOI is set to one, then the 8259 will automatically reset the interrupt request bit in the in-service register after supplying the type number to the processor. If AEOI 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 nonbuffered 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. Now 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 **O**peration **C**ommand **W**ords (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.

OCW2 : The OCW2 is sent to 8259A only when the AEOI mode (in ICW4) is not selected. The OCW2 is sent by the processor to decide on the type of **E**nd-**o**f-**I**nterrupt (EOI) and to program the priorities of the interrupt (i.e., IR inputs of 8259A). The different methods of EOI are discussed here :

1. ***Nonspecific End-of-Interrupt*** **:** This command is sent by the processor to 8259 to terminate the current interrupt being serviced by the 8259. This resets the corresponding bit in the in-service register of 8259 and allows the next higher priority interrupt.

2. ***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.

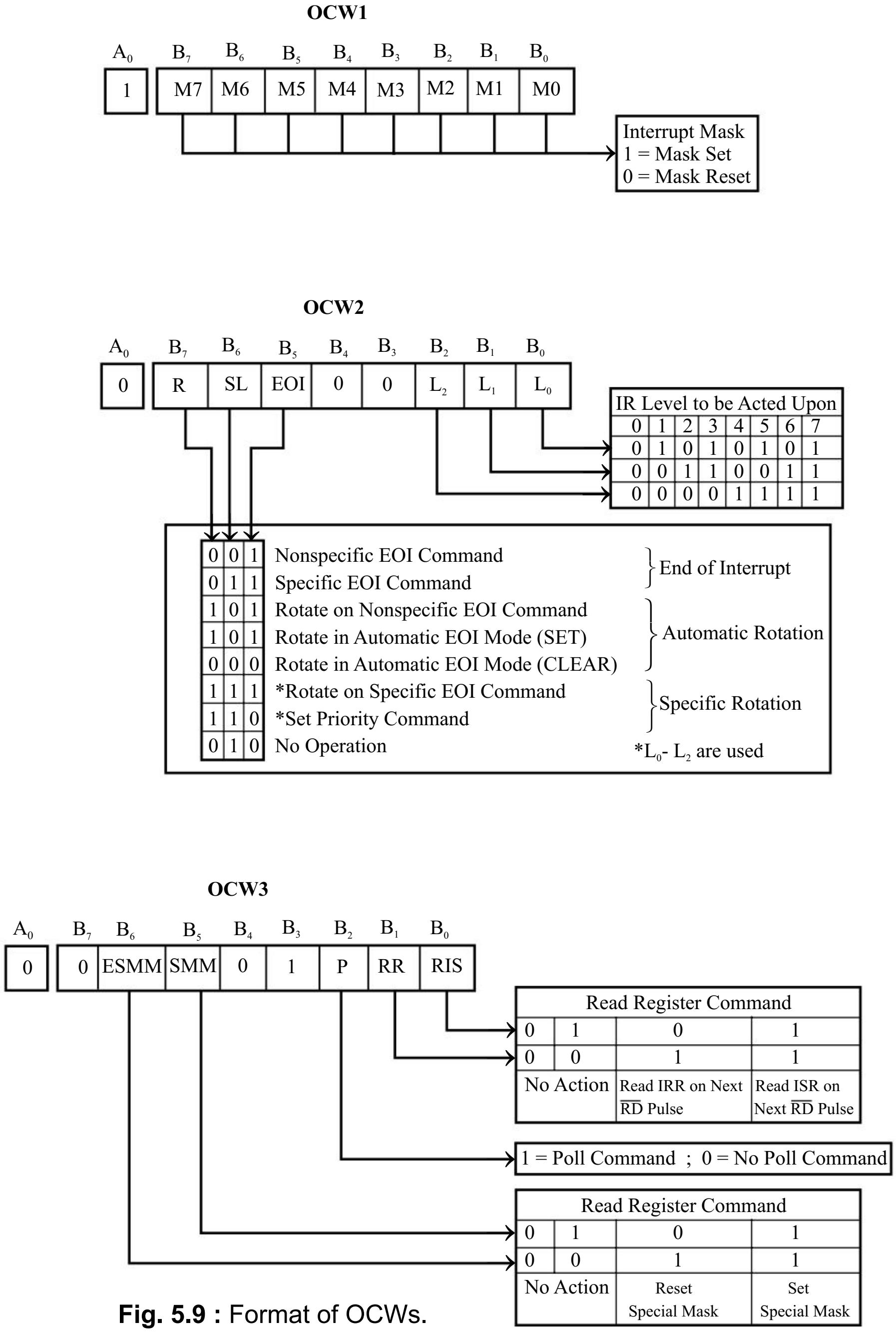
3. ***Rotate on Nonspecific EOI*** **:** This command will take action same as that of nonspecific EOI except that it rotates the priorities after resetting the bit in-service register. In this case the interrupts will have rotating priority, in which the priority of the currently serviced interrupt becomes the least.

4. ***Rotate on Automatic EOI :*** This command is sent to 8259 to select automatic EOI with rotating priority.

5. ***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.

6. ***Set priority :*** The command is sent to set the priority of the interrupt level specified by the lower three bits of OCW2 as the least.

OCW3 **:** The OCW3 is used to set special mask mode, poll the active interrupt request and to read the in-service and interrupt request registers. In special mask mode, the mask status are negated to allow the interrupts masked by the interrupt mask register.


OCW1
A0 B7 B6 B5 B4 B3 B2 B1 B0
1 M7 M6 M5 M4 M3 M2 M1 M0
Interrupt Mask
1 = Mask Set
0 = Mask Reset
OCW2
A0 B7 B6 B5 B4 B3 B2 B1 B0
0 R SL EOI 0 0 L2 L1 L0
End of Interrupt
Automatic Rotation
Specific Rotation
*L0- L2 are used
OCW3
A0 B7 B6 B5 B4 B3 B2 B1 B0
0 0 ESMM SMM 0 1 P RR RIS
1 = Poll Command ; 0 = No Poll Command

IR Level to be Acted Upon							
0	1	2	3	4	5	6	7
0	1	0	1	0	1	0	1
0	0	1	1	0	0	1	1
0	0	0	0	1	1	1	1

R	SL	EOI	
0	0	1	Nonspecific EOI Command
0	1	1	Specific EOI Command
1	0	1	Rotate on Nonspecific EOI Command
1	0	1	Rotate in Automatic EOI Mode (SET)
0	0	0	Rotate in Automatic EOI Mode (CLEAR)
1	1	1	*Rotate on Specific EOI Command
1	1	0	*Set Priority Command
0	1	0	No Operation

Read Register Command			
0	1	0	1
0	0	1	1
No Action		Read IRR on Next $\overline{RD}$ Pulse	Read ISR on Next $\overline{RD}$ Pulse

Read Register Command			
0	1	0	1
0	0	1	1
No Action		Reset Special Mask	Set Special Mask

Fig. 5.9 : Format of OCWs.

5.8 SHORT QUESTIONS AND ANSWERS

5.1 *What is an interrupt ?*

Interrupt is a signal sent by an external device to the processor to request the processor to perform a particular task or work.

5.2 *How are the 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 Nonvectored interrupt.
Method III	:	The interrupts are classified into Maskable and Nonmaskable interrupts.

5.3 *Explain how a microprocessor services an interrupt request ?*

When the processor recognizes an interrupt, it saves the processor status in a stack. Then it calls and execute an **I**nterrupt **S**ervice **S**ubroutine (ISS). At the end of ISS, it restores the processor status and the program control is transferred to the main program.

5.4 *What is the role of interrupt service subroutine?*

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.

5.5 *What are software interrupts?*

Software interrupts are program instructions. These instructions are inserted at the desired locations in a program. While running a program, if a software interrupt instruction is encountered then the processor executes an interrupt service routine.

5.6 *What is hardware interrupt?*

If an interrupt is initiated in a processor by applying an appropriate signal to interrupt pin, then the interrupt is called hardware interrupt.

5.7 *What is the difference between hardware and software interrupt?*

The software interrupt is initiated by the main program, but the hardware interrupt is initiated by an external device.

In 8086, the software interrupt cannot be disabled or masked but the hardware interrupts except NMI can be disabled or masked.

5.8 *What is vectored and nonvectored 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 nonvectored 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.

5.9 *What is masking and why is it required?*

Masking is the prevention of 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 a processor with multiple interrupts, the lower priority interrupt can be masked so as to prevent it from interrupting, the execution of interrupt service subroutine of higher priority interrupt.

5.10 *What is vectoring?*

Vectoring is the process of generating the address of an interrupt service subroutine to be loaded in program counter.

5.11 What are the sources of 8086 interrupt?

There are three sources for interrupts in 8086.

1. One source is from an external signal applied to INTR or NMI pin of the processor.
2. The second source of an interrupt is the execution of interrupt instruction "INT n".
3. The third source of an interrupt is from some condition produced in the 8086 by the execution of an instruction.

5.12 What is an exception? Give an example.

Exception is an interrupt generated due to exceptional condition (i.e., impossible situation) which occurs while executing an instruction. Example of an exception is divide by zero interrupt in 8086. While executing division instruction if the divisor is zero, then the 8086 will generate a divide by zero (type-0) interrupt.

5.13 How many interrupts are available in 8086? How are they classified?

The 8086 has 256 types of interrupts. INTEL has given a type number to the interrupts in the range of 0 to 255_{10}. Type-0 to type-4 are defined by INTEL and they are called INTEL predefined interrupts. Type-5 to type-31 are reserved by INTEL for use in future processors. Type-32 to type-255 are available for the user as hardware or software interrupts.

5.14 How can interrupts be initiated in 8086?

The 8086 processor has dual facility of initiating all the 256 interrupts. The interrupts can be initiated either by executing the "INT n" instruction where n is the type number or the interrupt can be initiated by sending an appropriate signal to the INTR pin of the processor.

5.15 List the INTEL predefined interrupts.

The INTEL predefined interrupts are:

i) Division by zero (Type-0 interrupt)
ii) Single step (Type-1 interrupt)
iii) Non-maskable interrupt, NMI (Type-2 interrupt)
iv) Breakpoint interrupt (Type-3 interrupt)
v) Interrupt on overflow (Type-4 interrupt)

5.16 What are software and hardware interrupts of 8086?

In 8086, the interrupts initiated by executing the "INT n" instruction are called software interrupts.

The interrupts initiated by applying appropriate signals to the INTR and NMI pins of 8086 are called hardware interrupt.

5.17 What are maskable and nonmaskable interrupts of 8086?

The hardware interrupts initiated by applying an appropriate signal to INTR pin of 8086 are maskable interrupts.

The software interrupts and the hardware interrupt NMI are nonmaskable.

5.18 How can the interrupts be masked/unmasked in 8086?

The maskable interrupts of 8086 can be masked by clearing the interrupt flag to zero and they can be unmasked/allowed by setting the interrupt flag to one.

5.19 What is a vector table? Where is it located?

The memory block consisting of vector addresses of all the 256 types of interrupts of 8086 is called vector table. The vector table is stored in the first 1 kb of physical memory space.

5.20 How is the interrupt address generated in 8086?

The 8086 will multiply the type number by four and sign extend to 20-bit to get a memory address of a vector table. The vector address for an interrupt will be available in four consecutive memory locations starting from this 20-bit address. The first word in the table is offset address of ISS (**I**nterrupt **S**ervice **S**ubroutine) and the next word is the segment base address of ISS.

5.21 *What is the need for an interrupt controller?*

The interrupt controller is employed to expand the interrupt input. It can handle the interrupt request from various devices and allow one by one to the processor.

5.22 *List some of the features of INTEL 8259 (Programmable Interrupt Controller).*

- It can manages eight interrupt request.
- The interrupt vector addresses are programmable.
- The priorities of interrupts are programmable.
- The interrupt can be masked or unmasked individually.

5.23 *Write the various functional blocks of INTEL 8259 ?*

The various functional blocks of 8259 are Control logic, Read/ Write logic, Data bus buffer, **I**nterrupt **R**equest **R**egister (IRR), **I**nterrupt **M**ask **R**egister (IMR) and **I**n-**S**ervice **R**egister (ISR), **P**riority **R**esolver (PR) and Cascade buffer.

5.24 *What is master and slave 8259 ?*

When 8259s are connected in cascade, one 8259 will be directly interrupting the processor and it is called master 8259. To each interrupt request input of master 8259, one slave 8259 can be connected. The 8259s interrupting the master 8259 are called slave 8259.

5.25 *How is 8259 programmed?*

The 8259 is programmed by sending **I**nitialization **C**ommand **W**ords (ICWs) and **O**perational **C**ommand **W**ords (OCWs).

5.26 *What are the features of 8259 that are programmed using ICWs?*

The ICWs are used to program the following features of 8259.

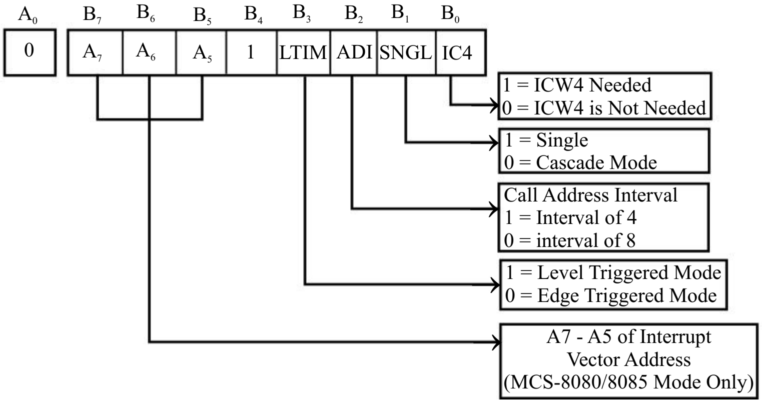
- Call address interval (in case of 8085)
- Cascade mode or single
- Level or edge triggered
- Vector address (in case of 8085) or type number (in case of 8086)
- 8085 or 8086 mode
- Auto or normal end of interrupt
- Special fully nested mode

5.27 *What are the features of 8259 that can be programmed using OCWs?*

The OCWs are used to program the following features of 8259.

- Masking of individual interrupts.
- Specific or Non-specific end of interrupt.
- Priority modes.

5.28 *Write the format of ICW1?*



5.29 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.

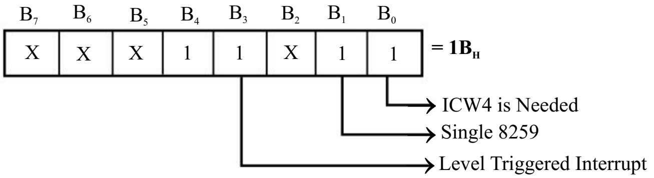
5.30 When ICW 4 is sent to 8259 ?

The ICW 4 is send 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 nonbuffered mode.

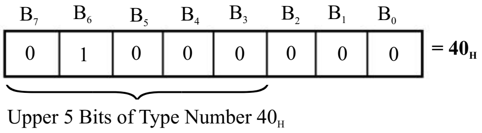
5.31 Frame the command words ICW1, ICW2, ICW4 and OCW1 for initializing single 8259 to initiate INT 40H to INT 47H . The desired features are level triggered interrupt and automatic end of interrupt.

ICW1



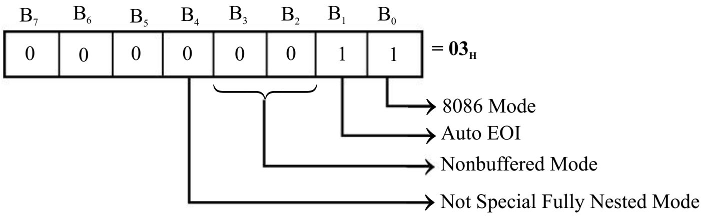
B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0	
X	X	X	1	1	X	1	1	**= $1B_H$**

ICW2



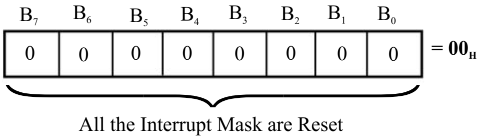
B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0	
0	1	0	0	0	0	0	0	**= 40_H**

ICW3



B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0	
0	0	0	0	0	0	1	1	**= 03_H**

OCW1



B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0	
0	0	0	0	0	0	0	0	**= 00_H**

5.32 Write a program segment to initialize a single 8259 connected to the 8086 processor.

Let us assume that 8259 is IO-mapped in the system with even address. The 8259 can be initialized by sending ICW1, ICW2, ICW4 and OCW1. Let the 8-bit address with $A_0 = 0$ be 00_H and when $A_0 = 1$ be 02_H.

```
MOV  AL,ICW1  ;Move ICW1 to AL-register
OUT  [00H]    ;Send ICW1 to 8259
MOV  AL,ICW2  ;Move ICW2 to AL-register
OUT  [02H]    ;Send ICW2 to 8259
MOV  AL,ICW4  ;Move ICW4 to AL-register
OUT  [02H]    ;Send ICW4 to 8259
MOV  AL,OCW1  ;Move OCW1 to AL-register
OUT  [02H]    ;Send OCW1 to 8259
HLT           ;Stop
```

CHAPTER 6

ASSEMBLY LANGUAGE PROGRAMMING

6.1 LEVELS OF PROGRAMMING

A program is a set of instructions or commands needed for performing 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. They are :

1. Machine level programming
2. Assembly level programming
3. High level programming

Machine Level Programming

In machine level programming, the instructions are written using binary codes which use only two symbols '0' and '1'. The manufacturer of microprocessors will give a set of instructions for each microprocessor in binary codes, i.e., a binary code will represent each 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 the 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 program in machine language.

Assembly Level Programming

In assembly level programming, the instructions are written using mnemonics. A mnemonic will have a few letters of English language which represent the operation performed by the instruction. For example, the mnemonic for the instruction which perform **addition** operation is **ADD**. The manufacturer of the microprocessors will provide a set of instructions in the form of 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 same family of microprocessors, i.e., the program developed for a lower version of microprocessor of a family can be run on a 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 systems 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 and so they are known as portable programs. A high level language program has to be converted into machine language program in order to be executed by the microprocessor. This conversion is performed by a software tool called compiler.

6.2 FLOWCHART

A flowchart is a graphical representation of the operation flow of the program. It is also the graphical form of algorithms. Flowcharts can be a valuable aid in visualizing programs. The various symbols used for drawing flowcharts are shown in Fig. 6.1. The operations represented by various symbols of flowchart are explained in Table-6.1. A sample flowchart is shown in Fig. 6.2.

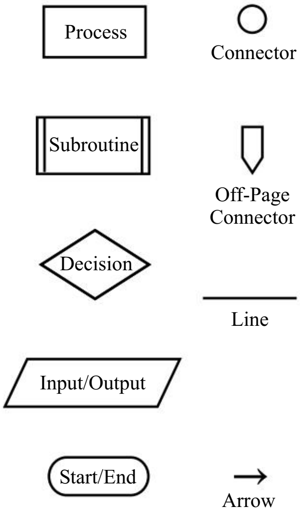


Fig. 6.1 : Symbols used in a flowchart.

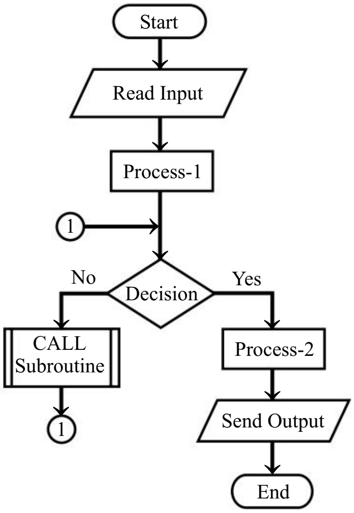
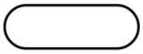
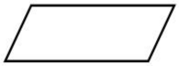
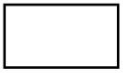
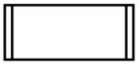
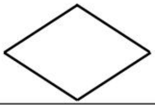
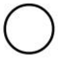
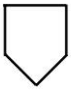


Fig. 6.2 : A sample flowchart.

TABLE - 6.1 : OPERATIONS REPRESENTED BY THE SYMBOLS USED IN FLOWCHART

Symbol	Operation
Racetrack-shaped box 	The racetrack shaped symbol is used to indicate the beginning (start) or end of a program.
Parallelogram 	The parallelogram is used to represent input or output operation.
Rectangular box 	The rectangular box is used to represent simple operations other than input and output operations.
A rectangular box with double lines on vertical sides 	The rectangular box with double lines on vertical sides is used to represent a subroutine or procedure.
Diamond-shaped box 	The diamond-shaped box is used to represent a decision point or cross-road in programs.
Small circle 	A small circle is used as a connector to show the connections between various parts of 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 off-page connector to show the connections between various sections of flowchart in different pages. Identical numbers are entered inside the boxes that represent the same connecting point.
Line ———	The lines are drawn between boxes and diamonds to indicate the program flow.
Arrow →	The arrows are placed on the lines to indicate the direction of program flow.

6.3 ASSEMBLY LANGUAGE PROGRAM DEVELOPMENT TOOLS

The 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 development system usually contain a standard PC (**P**ersonal **C**omputer), 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 the PC in order to write, assemble, debug, modify and test the assembly language programs.

Editor (Text Editor)

The 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, 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 an editor is known as source program and it is usually saved with 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 (**N**orton **E**ditor), 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 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 two-pass assemblers. In a two-pass assembler, the first pass is made through the 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 the machine code.

The input for the assembler is the source program which is saved with the file extension ".ASM". The assembler usually generates two output files called object file and list file. The object file consists 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 the 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 has 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 8086 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 collections 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 the library file and added to the program.

The input to the 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 a 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 a certain task 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

The debugger is a software tool that allows the execution of a program in single-step or breakpoint mode under the control of the 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 the register and memory contents to see whether the results are correct up to 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.

The 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.
- Breakpoint execution of a program.
- Display the contents of register/memory.

Simulators usually show the contents of registers and memory locations on the screen of the computer and allow 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 a development system with the newly designed hardware of the microprocessor system. A connector/plug at one end of the cable is plugged into the new hardware in the place of its microprocessor. The other end of the cable is connected to a parallel port of the 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.

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. 6.3.

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.

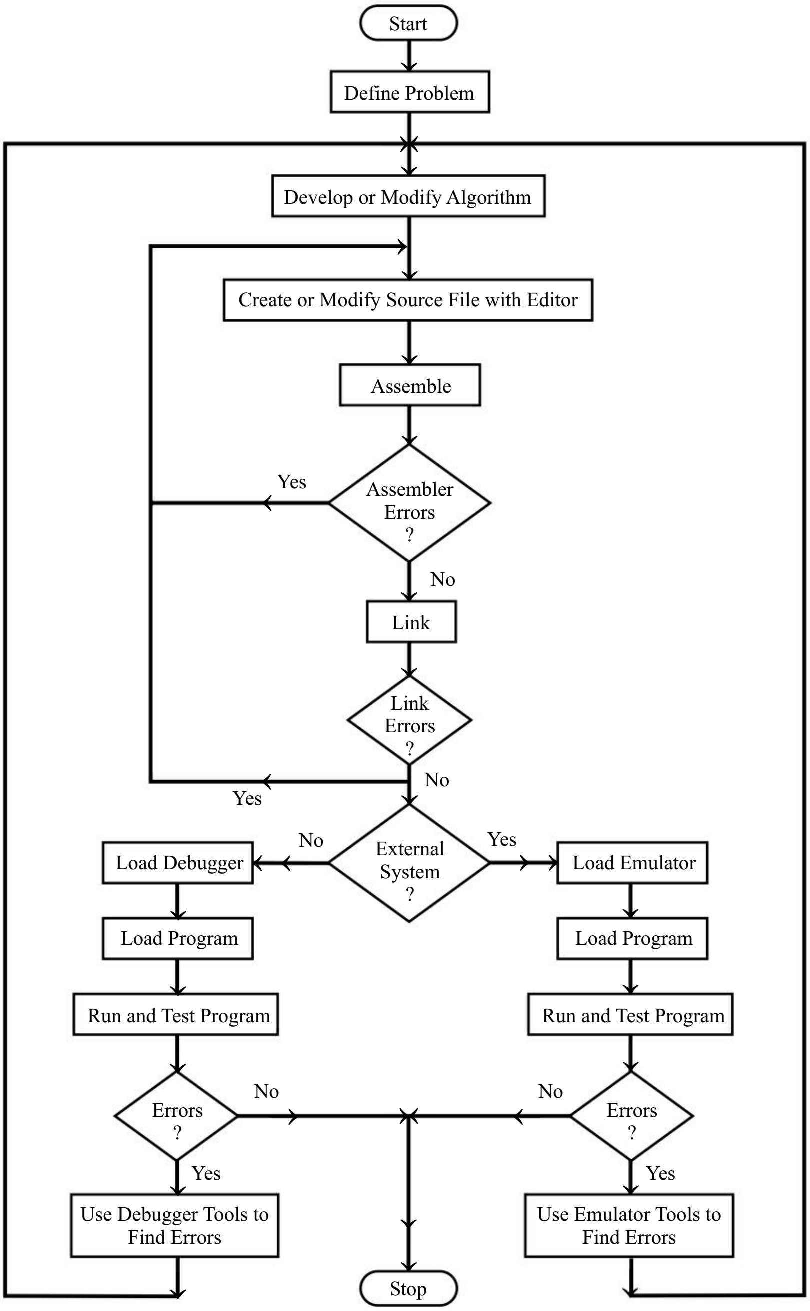


Fig. 6.3 : Development process of an assembly language program.

4. If the assembler list file indicates errors then use the editor and correct the errors.
5. Cycle through the edit-assemble loop until all errors indicated by the assembler are cleared.
6. Use the linker to link all object files of the 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 it 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.

6.4 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

The 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 construcing variable names, the numeric characters (0 to 9) should not be used as the 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 assembler and normally the maximum length of variable name is 32 characters.
4. The variable names are case insensitive. Therefore, the assembler does not distinguish between upper and lower case letters/alphabets.

Constants

The decimal, binary or hexadecimal numbers used to represent the data or address in an assembly language program statement are called constants or numerical constants. When constants are used to represent the address/data, 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 the 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 constant

1011	-	*Decimal (BCD) constant*
1060D	-	*Decimal constant*
1101B	-	*Binary constant*
92ACH	-	*Hexadecimal constant*
0E2H	-	*Hexadecimal constant*

Examples of invalid constant

1131B	-	*The character 3 should not be used in a binary constant.*
0E2	-	*The character H at the end of the 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.*

6.5 ASSEMBLER DIRECTIVES

The assembler directives are the instructions to the assembler regarding the program being assembled. They are also called pseudo-instructions. The assembler directives are used to specify start and end of a program, attach value to variables, allocate storage locations to 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 8086 assembly language program development are listed in Table-6.2. A brief discussion about some of the assembler directives is presented in the following sections.

TABLE - 6.2 : ASSEMBLER DIRECTIVES

Assembler directive	Function
ASSUME	Indicates the name of each segment to the assembler.
BYTE	Indicates a byte sized operand.
DB	Define byte. Used to define byte type variable.
DD	Define double word. Used to define 32-bit variable.
DQ	Define quad word. Used to define 64-bit variable.
DT	Define ten bytes. Used to define ten bytes of a variable.

Table-6.2 continued ...

Assembler directive	Function
DUP	Duplicate. Generate duplicates of characters or numbers.
DW	Define word. Used to define 16-bit variable.
DWORD	Double word. Indicates a double-word-sized operand.
END	Indicates the end of a program.
ENDP	End of procedure. Indicates the end of a procedure.
ENDS	End of segment. Indicates the end of a memory segment.
EQU	Equate. Used to equate numeric value or constant to a variable.
EVEN	Informs the assembler to align the data array starting from even address.
FAR	Used to declare the procedure as far which assigns a far address.
MACRO	Defines the name, parameters, and start of a macro.
NEAR	Used to declare a procedure as near which assigns a near address.
OFFSET	Specifies an offset address.
ORG	Origin. Used to assign the starting address for a program module or data segment.
PROC	Procedure. Defines the beginning of a procedure.
PTR	Pointer. It is used to indicate the type of memory access (BYTE/ WORD/ DWORD).
PUBLIC	Used to declare variables as common to various program modules.
SEGMENT	Defines the start of a memory segment.
STACK	Indicates that a segment is a stack segment.
SHORT	Used to assign one-byte displacement to jump instructions.
THIS	Used with EQU directive to set a label to a byte, word or double word.
WORD	Indicates a word sized operand.

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 variables is

variable DB value/values

Examples :	
AREA DB 45	*One memory location is reserved for the variable AREA and 45_{10} is stored as initial value in that memory location.*
LIST DB 7FH, 42H, 35H	*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.*
MARK DB 50 DUP (0)	*Fifty consecutive memory locations are reserved for the variable MARK and they are initialized with value zero.*

SCODE DB 'C'	*One memory location is reserved for variable SCODE and initialized with ASCII value of C.*
WELMSG DB 'HELLO RAM$'	*Ten consecutive memory locations are reserved for the variable WELMSG and they are initialized with ASCII value of H, E, L, L, O, space, R, A, M and $. (The symbol $ is used to denote end of a string.)*

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 store the 16-bit values specified in the statement as initial value in the allotted memory locations. The range of value that can be stored in word type variable is 0 to 65,535$_{10}$ (0000$_H$ to FFFF$_H$) for unsigned value, and -32,768 to +32,767 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 :	
WEIGHT DW 1250	*Two consecutive memory locations are reserved for the variable WEIGHT and initialized with value 1250$_{10}$.*
ALIST DW 6512H, 0F251H, 0CDE2H	*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 location.*
BCODE DW '8E'	*Two consecutive memory locations are reserved for variable BCODE and initialized with ASCII value of 8 and E .*

SEGMENT AND ENDS (END OF SEGMENT)

The directive SEGMENT is used to indicate the beginning of a code/data/stack segment. The directive ENDS is used to indicate the end of a code/data/stack segment. The directives SEGMENT and ENDS must enclose the program or data defining segment. The general form of writing a program or data defining segment is given below:

```
segnam   SEGMENT
   .   .   .  }      Program code
   .   .   .  }           or
   .   .   .  }  Data defining statements

segnam   ENDS
```

where "segnam" is the user defined name of the segment and it can be any valid assembler variable.

Examples :	
_DATA SEGMENT . . . } Data . . . } defining . . . } statements **_DATA ENDS**	*The _DATA is the name of the data segment enclosed by the directives SEGMENT and ENDS.*
_CODE SEGMENT . . . } . . . } Program codes . . . } **_CODE ENDS**	*The _CODE is the name of the program segment enclosed by the directives SEGMENT and ENDS.*

ASSUME

The directive ASSUME informs the assembler the name of the program/data segment that should be used for a specified segment. The general form of a statement using ASSUME directive is given below :

ASSUME segreg : *segnam, ... , segreg : segnam*

where, "segreg" is the segment register.

"segnam" is user defined name of the segment.

The segment register can be any of the CS, SS, DS and ES registers and segment name can be any valid assembler variable. In a single statement logical segments can be assigned to one or all the segment registers .

Examples :	
ASSUME CS : _CODE	*The directive ASSUME informs the assembler that the instruction of the program are stored in the user-defined logical segment _CODE.*
ASSUME DS : _DATA	*The directive ASSUME informs the assembler that the data of the program are stored in the user-defined logical segment _DATA.*
ASSUME CS : ACODE, DS: ADATA	*The directive ASSUME informs the assembler that the instructions of the program are stored in the segment ACODE and data are stored in the segment ADATA.*

ORG, END, EVEN AND EQU

The directive ORG (Origin) is used to assign the starting address (effective address) for a program/data segment. The directive END is used to terminate a program. The statements after the directive END will be ignored by the assembler.

The directive EVEN will inform the assembler to store the program/data segment starting from an even address. The 8086 requires one bus cycle to access a word at even address and two bus cycles to access a word at odd address. The even alignment with EVEN directive helps in accessing a series of consecutive memory words quickly.

The directive EQU (Equate) is used to attach a value to a variable.

Examples :	
ORG 1000H	*This directive informs the assembler that the statements following ORG 1000H should be stored in memory starting with effective address 1000_H.*
PORT1 EQU 0F2H	*The value of variable PORT1 is $F2_H$.*
LOOP EQU 10FEH	*The value of variable LOOP is $10FE_H$.*
SDATA SEGMENT ***ORG 1200H*** ***A DB 4CH*** ***EVEN*** ***B DW 1052H*** *** SDATA ENDS***	*In this data segment the effective address of memory location assigned to A will be 1200_H and the effective address of memory location assigned to B will be 1202_H and 1203_H.*

PROC, FAR, NEAR AND ENDP

The directives PROC, FAR, NEAR and ENDP are used to define a procedure/subroutine. The directive PROC indicates the beginning of a procedure and the directive ENDP indicates the end of a procedure. The FAR or NEAR, are type specifier which is used by the assembler to differentiate intrasegment call (call within segment/near call) and intersegment call (call from another segment/far call).

The general form of writing a procedure is given below:

```
procname    PROC [NEAR/FAR]
                      .    .    .  }
                      .    .    .  }  Program statements in the procedure
                      .    .    .  }
                  RET                 ; Last statement of the procedure
procname   ENDP
```

where "procname" is the user defined name of the procedure.

The procedure name can be any valid assembler variable. The type specifier NEAR/FAR is optional and if it is discarded then the assembler assumes the procedure as near call. Also the use of a specifier helps the assembler to decide whether to code RETas near return or far return.

Examples :	
ADD64 PROC NEAR . . . } Program . . . } statements in . . . } the procedure ***RET*** **ADD64 ENDP**	*The subroutine/procedure named ADD64 is declared as NEAR and so the assembler will code the CALL and RET instructions involved in this procedure as near call and return.*
CONVERT PROC FAR . . . } Program . . . } statements in . . . } the procedure ***RET*** **CONVERT ENDP**	*The subroutine/procedure named CONVERT is declared as FAR and so the assembler will code the CALL and RET instructions involved in this procedure as far call and return.*

SHORT

The directive SHORT is used to reserve one memory location for 8-bit signed displacement in jump instructions.

Examples :	
JMP SHORT AHEAD	*The directive will reserve one memory location for an 8-bit displacement named AHEAD.*

MACRO AND ENDM

The directive MACRO is used to indicate the beginning of a macro and the directive ENDM is used to indicate the end of a macro. The directives MACRO and ENDM must enclose the definitions, declarations and program statements which are to be substituted at the invocation of a macro.

The general form of writing a macro is given below:

```
macroname   MACRO [Arg1, Arg2, . . . . .]
                .    .    .   }
                .    .    .   }  Program statements in macro
                .    .    .   }
macroname   ENDM
```

where, "macroname" is the user defined name of the macro.
The macroname can be any valid assembler variable.

6.6 PROCEDURES AND MACROS

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.

The 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 procedure and after executing the procedure the program control is transferred back to the main program. In an 8086 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 8086 processor has two types of call instructions and they are intrasegment call or near call (call within a segment) and intersegment call or far call (call outside a segment). A procedure can be called using near call instruction if it is stored in the same segment where the main program is also stored. A procedure can be called using far call instruction if the procedure and main program are stored in different memory segments.

The procedures are terminated with RET instructions. The 8086 has two types of RET instructions and they are near return and far return. The near return instruction is used to terminate a procedure stored in the same segment. The far return instruction is used to terminate a procedure stored in a different segment.

When a procedure is called by using far call instruction, the 8086 processor will push the contents of IP and CS-register in stack and the segment base address of procedure is loaded in CS-register and the effective address of procedure is loaded in IP. Now the program control is transferred to procedure stored in another segment and so the processor will start executing the instructions of the procedure. At the end of procedure, RET instruction is encountered. On executing the RET instruction, the top of stack (which is the previous stored value) is poped to CS-register and IP. Thus the program control is returned to main program.

When a procedure is called by using near call instruction, the 8086 processor will push the contents of IP alone in stack and the effective address of procedure is loaded in IP. Here the content of CS-register is not altered. Now the program control is transferred to procedure stored in same segment and so the processor will start executing the instructions of the procedures. At the end of procedure, RET instruction is encountered. On executing the RET instruction, the top of stack (which is the previous stored value) is poped to IP. Thus, the program control is returned to 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.

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. Macro is a small group of instructions enclosed by the assembler directives MACRO and ENDM. Macros are identified by their name and usually defined at the start of a program.

The macro is called by its name in the program. Whenever a macro is called in a program, the assembler will insert the defined group of instructions in place of the call. In other words, the macro call is like shorthand 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 represent 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 - 6.3 : COMPARISON OF PROCEDURE AND MACRO

Procedure	Macro
1. Accessed by CALL and RET mechanism during program execution. 2. Machine code for instructions are stored in memory once. 3. Parameters are passed in registers, memory locations or stack.	1. Accessed during assembly with name given to macro when defined. 2. Machine codes are generated for instructions in the macro each time it is called. 3. Parameters are passed as part of statement which calls macro.

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 **S**tack **P**ointer (SP) to hold the address of the stack. Also the processor will have facility to automatically decrement/increment the content of SP after every write/read into stack.

The user can initialize or create a stack by loading a RAM memory address in **S**tack **P**ointer (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 available registers are not sufficient for storing intermediate result and data, then some of intermediate result and data can be stored in the 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 poped to program counter and so the program control returns to the main program after execution of the subroutine.

Stack in 8086 microprocessor

The usage and functioning of the stack in 8086 is similar to that of 8085 except the computation of the stack address. In 8085, the content of SP is the address of top of stack, whereas in 8086 the content of the SP is the offset address of tthe op of stack. The physical address of the stack in 8086 is computed by using the contents of SS-register and SP.

In 8086 microprocessor-based system, the stack is created by loading a 16-bit base address in **S**tack **S**egment (SS) register and a 16-bit offset address in **S**tack **P**ointer (SP). The 20-bit physical address of the stack is computed by multiplying the contents of SS-register by 16_{10} and then adding the contents of SP to this product. Here the content of SP is the offset address of the stack. Upon reset, the SS-register and SP are cleared to zero.

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. The contents of SS-register will not be altered while reading or writing into the stack. Like 8085, the stack in 8086 is also a LIFO stack. A typical example of stack in 8086 is shown in Fig. 6.4.

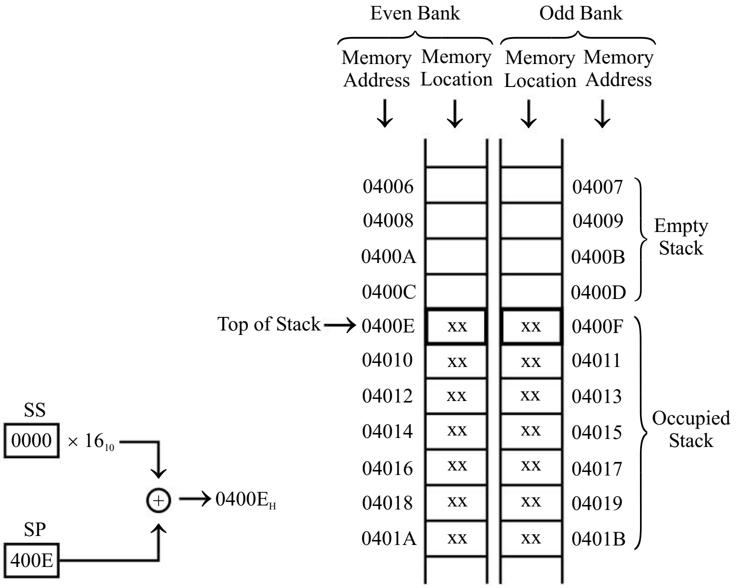


Fig. 6.4 : Example of stack in an 8086.

6.7 INTERRUPTS OF PERSONAL COMPUTERS

The 8086 assembly language program can be executed in any **P**ersonal **C**omputer (PC) based on a 80x86/pentium processor or its compatibility. While executing the programs in PC, the IO devices of the PC like keyboard, monitor, printer, etc., can be used as interactive IO devices for input data to the program and outputs the result of the program. These devices can be accessed by the programmer through the predefined interrupts of personal computer.

In the personal computers based on 80x86/pentium processor, specific interrupt type number are assigned to various activities. The interrupts predefined in personal computers can be broadly classified into the following three groups.

1. Interrupts generated from peripherals or exceptions.
2. Interrupts for services (system calls) through software interrupts.
3. Interrupts used to store pointers to the device parameters.

The interrupts of each group along with function assigned are listed in Table-6.4 to Table-6.6. In personal computers, the BIOS and OS programs will initialize the vector table for the interrupts listed from Table-6.4 to Table-6.6. These interrupt vector tables should not be modified by the programmer. The interrupts which are not mentioned in the Tables 6.5 to 6.7, are not predefined in personal computers and so the undefined interrupts can be used by the programmer for any specific task/function.

TABLE - 6.4 : HARDWARE OR EXCEPTION INTERRUPTS OF PC

Interrupt number	Function assigned
INT 00H	Division by zero
INT 01H	Single-step
INT 02H	Nonmaskable
INT 03H	Breakpoint
INT 04H	Overflow
INT 05H	Print screen
INT 06H	Reserved
INT 07H	Reserved
INT 08H	Timer
INT 09H	Keyboard
INT 0AH to INT 0DH	Hardware Interrupts
INT 0EH	Diskette
INT 0FH	Hardware Interrupt

TABLE - 6.5 : SOFTWARE INTERRUPTS IN PC FOR IMPLEMENTING SYSTEM CALLS

Interrupt number	Function assigned
INT 10H to INT 17H	BIOS Interrupts
INT 18H	ROM - BASIC
INT 19H	Bootstrap
INT 1AH	Time IO
INT 1BH	Keyboard Break
INT 1CH	User timer Interrupt
INT 20H to INT 2FH	DOS Interrupts
INT 67H	Expanded Memory Functions

TABLE - 6.6 : INTERRUPTS USED IN PC TO STORE POINTERS TO DEVICE PARAMETERS

Interrupt number	Function assigned
INT 1DH	Video Parameters
INT 1EH	Diskette Parameters
INT 1FH	Graphics Characters
INT 41H	Hard Disk-0 Parameters
INT 46H	Hard Disk-1 Parameters
INT 44H	EGA Graphic Characters
INT 4AH	User Alarm Address
INT 50H	CMOS Timer Interrupt

Some of the DOS and BIOS interrupts are explained in the following sections. For detailed discussion on the interrupts of PC, readers are advised to refer the IBM PC technical reference manual and DOS reference manual.

DOS Interrupts

The DOS (**D**isk **O**perating **S**ystem) provides a large number of procedures to access devices, files, memory and process control services. These procedures can be called in any user program using software interrupts "INT n" instruction. The various DOS interrupts are listed in Table-6.6.

The DOS interrupt INT 21H provides a large number of services. A function code has been allotted to each service provided by INT 21H. The function code should be loaded in AH-register before calling INT 21H to avail the service provided by the function.

TABLE - 6.7 : DOS INTERRUPTS

Interrupt type	Service provided by the interrupt
INT 20H	Program Terminate
INT 21H	DOS services (DOS system call)
INT 22H	Terminate Address
INT 23H	Control Break Address
INT 24H	Critical Error Handler Address
INT 25H	Absolute Disk Read
INT 26H	Absolute Disk Write
INT 27H	Terminate and Stay Resident (TSR)
INT 28H	DOS time slice
INT 2EH	Perform DOS Command
INT 2FH	Multiplex Interrupts

The various services provided by the INT 21H are classified depending on the function performed by them and they are listed in Appendix-II.

The following steps are involved in accessing DOS services :

1. Load a DOS function number in AH-register. If there is a subfunction, then its code is loaded in AL-register.
2. Load the other registers as indicated in the DOS service formats.
3. Prepare buffers, ASCIIZ (ASCII string terminated by zero) and control blocks, if necessary.
4. Set the location of Disk Transfer Area, if necessary.
5. Invoke DOS service INT 21H.
6. The DOS service will return the required parameters in the specified registers.

Note : All values entered in the register are preserved by the DOS service call except when information is returned in a register.

BIOS Interrupts

In personal computers, the basic interface between the hardware and software is provided by a program stored in ROM called BIOS program. (BIOS-**B**asic **I**nput **O**utput control **S**ystem). The BIOS program consists of a large number of procedures to access various hardwares in a PC. These procedures can be called in any user program using software interrupts "INT n" instruction. Even the DOS uses BIOS interrupts to control the hardware. The various BIOS interrupts are listed in Table-6.8.

TABLE - 6.8 : BIOS INTERRUPTS

Interrupt type	Service name
INT 10H	Video services
INT 11H	Machine configuration
INT 12H	Usable RAM Memory size
INT 13H	Disk IO
INT 14H	Serial port IO (RS 232C)
INT 15H	AT services
INT 16H	Keyboard IO
INT 17H	Printer IO

Each BIOS interrupt provides a large number of services. A function code has been allotted to each service provided by the BIOS interrupts. The function code should be loaded in the AH-register before calling the BIOS interrupt to avail the service provided by the function. The various functions performed by BIOS interrupts are listed in Appendix-II.

The following steps are involved in accessing the BIOS services:

1. Load a BIOS function number in the AH-register. If there is a subfunction, then its code is loaded in AL-register.
2. Load the other register as indicated in the BIOS service formats.
3. Prepare buffers, ASCIIZ (ASCII string terminated by zero) and control blocks , if necessary.
4. Invoke BIOS call.
5. The BIOS service will return the required parameters in the specified register.

Note : *All values entered in the register are preserved except when information is returned in a register.*

Explanation of DOS and BIOS Interrupts

The explanation provided for DOS and BIOS interrupts consists of following three sections :

1. **Operation** : This section explains the dedicated operation performed by the interrupt with specified function code.
2. **Expects** : This section explains the register and parameter settings required before accessing the service.
3. **Returns** : This section explains the status of a service call and return parameters after the response of the service.

INT 10H, FUNCTION CODE 02H : SET CURSOR POSITION

Operation : The INT 10H with function code 02H is used to set the position of the cursor on the monitor using text coordinates (row and column).

Expects : AH = 02H
BH = Video page (must be zero in graphics mode)
DH = Row (y–coordinate)
DL = Column (x–coordinate)

Returns : None

INT 10H, FUNCTION CODE 03H : READ CURSOR POSITION

Operation : The INT 10H with function code 03H is used to read the current position of cursor on the monitor in text coordinates.

Expects : AH = 03H
BH = Video page

Returns : DH = Current row (y-coordinate)
DL = Current column (x-coordinate)
CH = Starting line for cursor
CL = Ending line for cursor

INT 10H, FUNCTION CODE 06H : INITIALIZE/SCROLL RECTANGULAR WINDOW UP

Operation : The INT 10H with function code 06H is used to initialize a specified rectangular window on the monitor or scrolls the contents of a window up by a specified number of lines.

Expects : AH = 06H
AL = Number of lines to scroll up
(If AL = zero, entire window is cleared or blanked)
BH = Blanked area attributes
CH = y-coordinate, upper left corner of window
CL = x-coordinate, upper left corner of window
DH = y-coordinate, lower right corner of window
DL = x-coordinate, lower right corner of window

Returns : None

INT 10H, FUNCTION CODE 07H: INITIALIZE/SCROLL RECTANGULAR WINDOW DOWN

Operation : The INT 10H with function code 07H is used to initialize a specified rectangular window or scrolls the contents of a window down by a specified number of lines.

Expects : AH = 07H
AL = Number of lines to scroll down
(If AL = zero, entire window is cleared or blanked)
BH = Blanked area attributes
CH = y-coordinate, upper left corner of window
CL = x-coordinate, upper left corner of window
DH = y-coordinate, lower right corner of window
DL = x-coordinate, lower right corner of window

Returns : None

INT 10H, FUNCTION CODE 09H : WRITE CHARACTER AND ATTRIBUTE AT CURSOR

Operation : The INT 10H with function code 09H is used to write a specified ASCII character and its attribute to the monitor at the current cursor position.

Expects : AH = 09H
AL = ASCII character code
BH = Video page
BL = Attribute (in text mode) or colour (in graphics mode)
CX = Count of character to write (replication factor).

Returns : None

INT 10H, FUNCTION CODE 0AH : WRITE CHARACTER ONLY AT CURSOR

Operation : The INT 10H with function code 0AH is used to write an ASCII character to the monitor at current cursor position. The character uses the attribute of the previous character displayed at the same position.

Expects : AH = 0AH
AL = ASCII character code
BH = Video page
BL = Colour (graphics mode)
CX = Count of character to write (replication factor)

Returns : None

INT 16H, FUNCTION CODE 00H : READ KEYBOARD CHARACTER

Operation : The INT 16H with function code 00H is used to read a character from the keyboard. It also returns the keyboard scan code.

Expects : AH = 00H

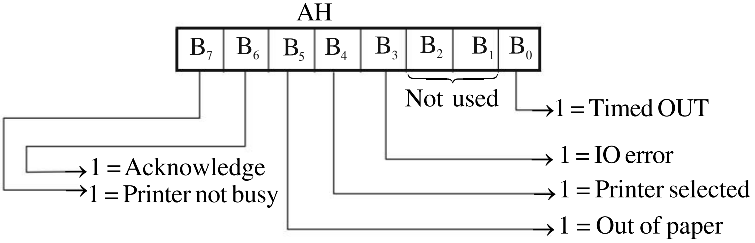
Returns : AH = Keyboard scan code
AL = ASCII character code

INT 17H, FUNCTION CODE 00H : WRITE TO PRINTER

Operation : The INT 17H with function code 00H is used to send a character to the specified parallel port to which a printer is connected. It also returns the current status of the printer.

Expects : AH = 00H
AL = Character to be written
DX = Port number ("0" for LPT1, "1" for LPT2 and "2" for LPT3)

Returns : AH = Printer status as shown below



INT 21H, FUNCTION CODE 01H : READ CHARACTER FROM STANDARD INPUT DEVICE

Operation : The INT 21H with function code 01H reads a character from the standard input device (keyboard) and echoes (send) the character to the standard output device (monitor). It waits for the character if no character is available on the input device.

Expects : AH = 01H

Returns : AL = ASCII code of the input key.

INT 21H, FUNCTION CODE 02H : WRITE CHARACTER TO STANDARD OUTPUT DEVICE

Operation : The INT 21H with function code 02H writes a character to the standard output device (monitor).

Expects : AH = 02H
DL = ASCII character code

Returns : None

INT 21H, FUNCTION CODE 05H : SEND A CHARACTER TO PRINTER

Operation : The INT 21H with function code 05H send a character to printer. The default parallel port is LPT1, unless explicitly redirected by the DOS.

Expects : AH = 05H
DL = ASCII code of the output character

Returns : None

INT 21H, FUNCTION CODE 08H : READ CHARACTER WITHOUT ECHO

Operation : The INT 21H with function code 08H reads a character from the standard input device (keyboard) without echo to the standard output device (monitor). It waits for the character, if no character is available on the input device.

Expects : AH = 08H

Returns : AL = ASCII code of the input key.

INT 21H, FUNCTION CODE 09H : DISPLAY STRING

Operation : The INT 21H with function code 09H writes a string terminated with $ to the standard output device(monitor).

Expects : AH = 09
DS = Segment address of the string terminated by the symbol $.
DX = Offset address of the string terminated by the symbol $.

Returns : None

INT 21H, FUNCTION CODE 4CH : TERMINATE WITH RETURN CODE-EXIT(n).

Operation	:	The INT 21H with function code 4CH is used to terminate the program with return code. The return code "0" is generally considered as program terminating with successful execution. DOS sets the error level to the return code.

Expects	:	AH = 4CH
		AL = return code

Returns	:	None

6.8 HAND CODING OF ASSEMBLY LANGUAGE PROGRAMS

The 8086 assembly language programs should be converted to machine codes (binary codes) for execution. This can be acheived by two methods.

In one method, the software development tools like editor, assembler and linker are used to generate the machine codes of the program. Using editor, the assembly language program is typed and saved as ".asm" file. Using assembler, it is converted to machine code and saved as ".obj" file. Using linker, the machine codes are mapped to the memory of the target hardware and saved as ".exe" file. The ".exe" file is the machine language program which can be run on an 8086 system or its compatible.

In another method, the machine language codes of each instruction is obtained manually by referring to the machine code templates of 8086 provided by INTEL. This method is referred to as hand coding of assembly language program.

The template of 8086 instructions are listed in Appendix-I. The templates of each instruction will have a fixed binary code called opcode, programmable fields (like mod, reg, segreg, r/m) and one bit special indicators (like w, d, sw, v or z). The various choice of binary codes for programmable fields and one bit special indicators are listed in Tables A-1 to A-5 in Appendix-I:

While hand coding 8086 instructions, the following hints will be useful.

1. The term "mem" in the operand field of instructions refers to operand in memory (or memory operand) and it can be specified in 24 different ways as shown in Table A-5 of Appendix-I. The different methods of specifying memory operand differs in the way of calculating effective and physical address of memory. These calculations are shown in Table-3.4 in Chapter-3.
2. The term "disp8" in the operand field of jump instructions (for both conditional and unconditional) refer to the number of memory locations to be jumped forward or backward. For forward jump the disp8 is a positive integer and for backward jump the disp8 is a negative integer. Therefore, for backward jump the disp8 should be expressed in 2's complement form.

The hand coding of example program 19 in Section 6.9 is shown in Table-6.9

After hand coding of the instructions, they should be stored in memory locations. In the example program 19, the origin of the program effective address is specified as 1000_H. Therefore, the instructions are stored in the memory starting from the address 1000_H. The machine codes of the instructions along with the address of each instructions are listed in Table-6.10. In this table, the address in the address column refers to the starting address of each instruction.

TABLE - 6.9 : HANDCODING OF EXAMPLE PROGRAM 19

Instruction	Template	Binary code	Hex code
MOV SI, 1100H	1011 w reg l.b.data h.b.data w = 1, reg = 110, l.b.data = 00_H, h.b.data = 11_H	1011 1110 0000 0000 0001 0001	BE 00 11
MOV DL, [SI]	1000 10dw mod reg r/m d = 1, w = 0, mod = 00, reg = 010, r/m = 100	1000 1010 0001 0100	8A 14
MOV DI, 1200H	1011 w reg l.b.data h.b.data w = 1, reg = 111, l.b.data = 00_H, h.b.data = 12_H	1011 1111 0000 0000 0001 0010	BF 00 12
MOV BL, 01H	1011 w reg l.b.data w = 0, reg = 011, l.b.data = 01_H	1011 0011 0000 0001	B3 01
MOV [DI], BL	1000 10dw mod reg r/m d = 0, w = 0, mod = 00, reg = 011, r/m = 101	1000 1000 0001 1101	88 1D
INC DI	0100 0 reg reg = 111	0100 0111	47
INC BL	1111 111w mod 000 r/m w = 0, mod = 11, r/m = 011	1111 1110 1100 0011	FE C3
MOV CL, 02H	1011 w reg l.b.data w = 0, reg = 001, l.b.data = 02_H	1011 0001 0000 0010	B1 02
CMP BL, CL	0011 10dw mod reg r/m d = 1, w = 0, mod = 11, reg = 011, r/m = 001	0011 1010 1101 1001	3A D9
JZ STORE	0111 0100 disp8 disp8 = STORE	0111 0100 STORE	74 STORE
MOV AH, 00H	1011 w reg l.b.data w = 0, reg = 100, l.b.data = 00_H	1011 0100 0000 0000	B4 00

Table-6.9 continued ...

Instruction	Template	Binary code	Hex code
MOV AL, BL	1000 10dw mod reg r/m d = 1, w = 0, mod = 11, reg = 000, r/m = 011	1000 1010 1100 0011	8A C3
DIV CL	1111 011w mod 110 r/m w = 0, mod = 11, r/m = 001	1111 0110 1111 0001	F6 F1
CMP AH, 00H	1000 00sw mod 111 r/m l.b.data sw = 00, mod = 11, r/m = 100, l.b.data = 00_H	1000 0000 1111 1100 0000 0000	80 FC 00
JZ NEXT	0111 0100 disp8 disp8 = NEXT	0111 0100 NEXT	74 NEXT
INC CL	1111 111w mod 000 r/m w = 0, mod = 11, r/m = 001	1111 1110 1100 0001	FE C1
JMP REPEAT	1110 1011 disp8 disp8 = REPEAT	1110 1011 REPEAT	EB REPEAT
MOV [DI], BL	1000 10dw mod reg r/m d = 0, w = 0, mod = 00, reg = 011, r/m = 101	1000 1000 0001 1101	88 1D
INC DI	0100 0reg reg = 111	0100 0111	47
INC BL	1111 111 w mod 000 r/m w = 0, mod = 11, r/m = 011	1111 1110 1100 0011	FE C3
CMP BL, DL	0011 10dw mod reg r/m d = 1, w = 0, mod = 11, reg = 011, r/m = 001	0011 1010 1101 1001	3A D9
JNZ GENERAT	0111 0101 disp8 disp8 = GENERAT	0111 0101 GENERAT	75 GENERAT
HLT	1111 0100	1111 0100	F4

TABLE - 6.10

Instruction	Effective address in Hex	Hex code
MOV SI, 1100H	1000	BE 00 11
MOV DL, [SI]	1003	BA 14
MOV DI, 1200H	1005	BF 00 12
MOV BL, 01H	1008	B3 01
MOV [DI], BL	100A	88 1D
INC DI	100C	47
INC BL	100D	FE C3
GENERAT : MOV CL, 02H	100F	B1 02
REPEAT : CMP BL, CL	1011	3A D9
JZ STORE	1013	74 0F
MOV AH, 00H	1015	B4 00
MOV AL, BL	1017	8A C3
DIV CL	1019	F6 F1
CMP AH, 00H	101B	80 FC 00
JZ NEXT	101E	74 07
INC CL	1020	FE C1
JMP REPEAT	1022	EB ED
STORE : MOV [DI], BL	1024	88 1D
INC DI	1026	47
NEXT : INC BL	1027	FE C3
CMP BL, DL	1029	3A DA
JNZ GENERAT	102B	75 E2
HLT	102D	F4

The machine codes (or binary codes) for the disp8 in the jump instructions are determined as explained below.

The instruction "JZ STORE" is a forward jump. In this instruction the program control should be transferred to the instruction labelled "STORE" if zero flag is one. The instruction labelled "STORE" is stored in memory after 15_{10} (or $0F_H$) memory locations from "JZ STORE". Hence, the machine code for "STORE" is $0F_H$.

The instruction "JZ NEXT" is a forward jump. In this instruction the program control should be transferred to the instruction labelled NEXT if zero flag is one. The instruction labeled "NEXT" is stored in the memory after 07_{10} (or 07_H) memory locations from "JZ NEXT". Hence, the machine code for "NEXT" is 07_H.

The instruction "JMP REPEAT" is a backward jump. In this instruction, the program control is unconditionally transferred to the instruction labelled REPEAT which is stored at the memory address 1011_H. After executing "JMP REPEAT" instruction the content of IP (**I**nstruction **P**ointer) will be the address of the next instruction, which is 1024_H. The difference between these two addresses gives the disp8, which is the number of locations to be jumped backward. Here, $1024_H - 1011_H = 13_H = 19_{10}$. Since JMP REPEAT is a backward jump, we have to express 13_H in 2's complement. The 2's complement of 13_H is ED_H. Therefore, the machine code for REPEAT is ED_H.

The instruction "JNZ GENERAT" is a backward jump. In this instruction the program control is transferred to the instruction labelled GENERAT if zero flag is zero. The instruction labeled GENERAT is stored at memory address $100F_H$. After executing "JNZ GENERAT" instruction the content of IP (**I**nstruction **P**ointer) will be the address of the next instruction, which is $102D_H$. The difference between these two addresses gives the disp8, which is the number of locations to be jumped backward. Here, $102D_H - 100F_H = 1E_H = 30_{10}$. Since "JNZ GENERAT" is a backward jump, we have to express $1E_H$ in 2's complement. The 2's complement of $1E_H$ is $E2_H$. Therefore, the machine code for GENERAT is $E2_H$.

6.9 EXAMPLES OF 8086 ASSEMBLY LANGUAGE PROGRAMS

Note : 1. *The example programs 1 to 26 given in this book can be run on any 8086 microprocessor trainer kit. Since the initialization of segment registers and stack are taken care by the monitor program in the trainer kits, those initializations are not included in the example programs 1 to 26.*

2. *The example programs 27 to 30 given in this book can be run on INTEL microprocessor-based PC (Personal Computer) or its compatibility. In these programs the PC keyboard is used as input device and the monitor is used as the output device. The DOS and BIOS interrupts are used to access these devices.*

EXAMPLE PROGRAM 1 : 16-bit Addition

Write an assembly language program to add two numbers of a 16-bit data.

Problem Analysis

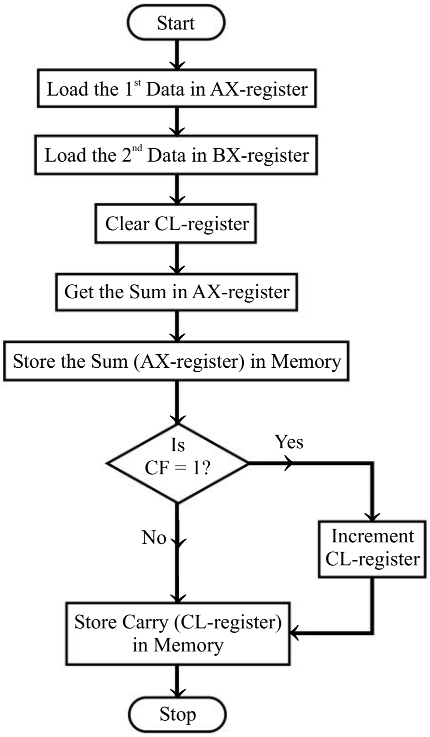
To perform addition in 8086, one of the data should be stored in a register and another data can be stored in register/memory. After addition, the sum will be available in the destination register/memory. The sum of two 16-bit data can be either 16 bits (sum only) or 17 bits (sum and carry). The destination register/memory can accommodate only the sum and if there is a carry the 8086 will indicate by setting carry flag.

Hence, one of the register is used to account for carry. The program for addition in 8086 has been presented by three methods. In method-I, immediate addressing is used for input data and direct addressing is used for output data. In method-II, direct addressing is used for input and output data. In method-III, indexed addressing is employed.

Algorithm (Method - I)

1. Load the first data in AX-register.
2. Load the second data in BX-register.
3. Clear CL-register.
4. Add the two data and get the sum in AX-register.
5. Store the sum in memory.
6. Check for carry. If carry flag is set then go to next step, otherwise go to step 8.
7. Increment CL-register.
8. Store the carry in memory.
9. Stop.

Flowchart (Method - I)


Start
Load the 1st Data in AX-register
Load the 2nd Data in BX-register
Clear CL-register
Get the Sum in AX-register
Store the Sum (AX-register) in Memory
Is CF = 1?
Yes
No
Increment CL-register
Store Carry (CL-register) in Memory
Stop

Assembly language program (Method - I)

```
;PROGRAM TO ADD TWO 16-BIT DATA (METHOD-1)
DATA SEGMENT          ;start of data segment.

        ORG    1104H  ;specify data segment starting address.
        SUM    DW 0   ;Reserve two memory locations for sum.
        CARRY  DB 0   ;Reserve one memory location for carry.

DATA ENDS             ;End of data segment.

CODE SEGMENT          ;Start of code segment.

       ASSUME CS:CODE ;Assembler directive.
       ASSUME DS:DATA ;Assembler directive.
       ORG 1000H      ;specify program starting address.

       MOV AX,205AH   ;Load the first data in AX-register.
       MOV BX,40EDH   ;Load the second data in BX-register.
       MOV CL,00H     ;Clear the CL-register for carry.
       ADD AX,BX      ;Add the two data, sum will be in AX.
       MOV SUM,AX     ;Store the sum in memory location (1104H).
       JNC AHEAD      ;Check the status of carry flag.
       INC CL         ;If carry flag is set,increment CL by one.
AHEAD: MOV CARRY,CL   ;Store the carry in memory location (1106H).
       HLT            ;Halt program execution.

CODE ENDS             ;End of code segment.
END                   ;Assembly end.
```

Assembler listing for example program 1 (Method - I)

```
;PROGRAM TO ADD TWO 16-BIT DATA (METHOD-1)
0000                DATA SEGMENT          ;start of data segment.

1104                       ORG   1104H    ;specify data segment starting address.
1104   0000                SUM   DW 0     ;Reserve two memory locations for sum.
1106   00                  CARRY DB 0     ;Reserve one memory location for carry.

1107                DATA ENDS             ;End of data segment.

0000                CODE SEGMENT          ;start of code segment.

                           ASSUME CS:CODE ;Assembler directive.
                           ASSUME DS:DATA ;Assembler directive.
1000                       ORG 1000H      ;specify program starting address.

1000   B8  205A            MOV  AX,205AH  ;Load the first data in AX-register.
1003   BB  40ED            MOV  BX,40EDH  ;Load the second data in BX-register.
1006   B1 00               MOV  CL,00H    ;Clear the CL-register for carry.
1008   03 C3               ADD  AX,BX     ;Add the two data, sum will be in AX.
100A   A3 1104 R           MOV  SUM,AX    ;Store the sum in memory location (1104H).
100D   73 02               JNC  AHEAD     ;Check the status of carry flag.
100F   FE  C1              INC  CL        ;If carry flag is set,increment CL by one.
1011   88  0E 1106 R AHEAD: MOV  CARRY,CL ;Store the carry in memory location (1106H).
1015   F4                  HLT            ;Halt program execution.

1016                CODE ENDS             ;End of code segment.
                    END                   ;Assembly end.
```

Sample data

Input Data : $205A_H$ Output Data : SUM = 6147_H

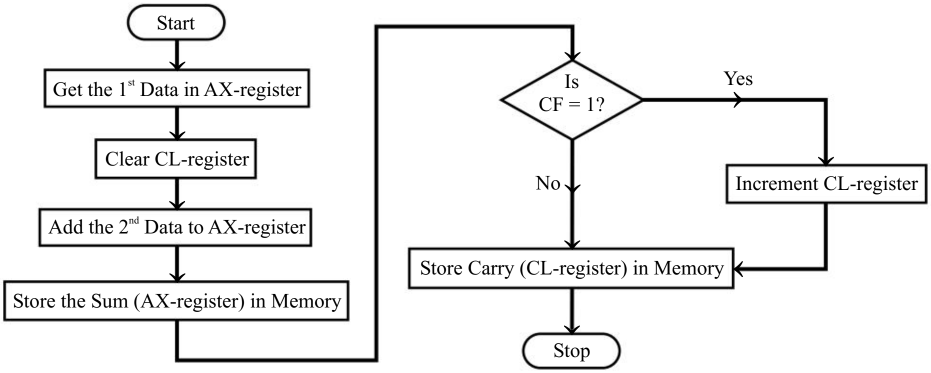
$40ED_H$ CARRY = 00_H

Memory address	Content
1104	47
1105	61
1106	00

Algorithm (Method - II)

1. Get the first data in AX-register.
2. Clear CL register.
3. Add the second data to AX-register and get the sum in AX-register.
4. Store the sum in memory.
5. Check for carry. If carry flag is set then go to next step, otherwise go to step 7.
6. Increment CL-register.
7. Store the carry in memory.
8. Stop.

Flowchart (Method - II)


Start
Get the 1st Data in AX-register
Clear CL-register
Add the 2nd Data to AX-register
Store the Sum (AX-register) in Memory
Is CF = 1?
Yes
No
Increment CL-register
Store Carry (CL-register) in Memory
Stop

Assembly language program (Method - II)

```
;PROGRAM TO ADD TWO 16-BIT DATA (METHOD-2)

DATA SEGMENT              ;Start of data segment.

        ORG    1100H      ;Specify data segment starting address.
        DATA1  DW 0       ;Reserve two memory locations for DATA1.
        DATA2  DW 0       ;Reserve two memory locations for DATA2.
        SUM    DW 0       ;Reserve two memory locations for sum.
        CARRY  DB 0       ;Reserve one memory location for carry.

DATA ENDS                 ;End of data segment.

CODE SEGMENT              ;Start of code segment.

        ASSUME CS:CODE    ;Assembler directive.
        ASSUME DS:DATA    ;Assembler directive.
        ORG 1000H         ;specify program starting address.

        MOV AX,DATA1      ;Load the first data in AX-register.
        MOV CL,00H        ;Clear the CL-register for carry.
        ADD AX,DATA2      ;Add 2nd data to AX, sum will be in AX.
        MOV SUM,AX        ;Store sum in memory location (1104H).
        JNC AHEAD         ;Check the status of carry flag.
        INC CL          ;If carry is set,increment CL by one.
AHEAD:  MOV CARRY,CL    ;Store carry in memory location (1106H).
        HLT             ;Halt program execution.

CODE ENDS               ;End of code segment.
END                     ;Assembly end.
```

Assembler listing for example program 1 (Method - II)

```
;PROGRAM TO ADD TWO 16-BIT DATA (METHOD-2)

0000                    DATA SEGMENT            ;Start of data segment.

1100                           ORG    1100H     ;Specify data segment starting address.
1100   0000                    DATA1  DW 0      ;Reserve two memory locations for DATA1.
1102   0000                    DATA2  DW 0      ;Reserve two memory locations for DATA2.
1104   0000                    SUM    DW 0      ;Reserve two memory locations for sum.
1106   00                      CARRY  DB 0      ;Reserve one memory location for carry.

1107                    DATA ENDS               ;End of data segment.

0000                    CODE SEGMENT            ;Start of code segment.

                               ASSUME CS:CODE   ;Assembler directive.
                               ASSUME DS:DATA   ;Assembler directive.
1000                           ORG  1000H       ;Specify program starting address.

1000   A1  1100 R              MOV  AX,DATA1    ;Load the first data in AX-register.
1003   B1  00                  MOV  CL,00H      ;Clear the CL-register for carry.
1005   03  06  1102 R          ADD  AX,DATA2    ;Add 2nd data to AX, sum will be in AX.
1009   A3  1104 R              MOV  SUM,AX      ;Store sum in memory location(1104H).
100C   73  02                  JNC  AHEAD       ;Check the status of carry flag.
100E   FE  C1                  INC  CL          ;If carry is set,increment CL by one.
1010   88  0E  1106 R   AHEAD: MOV  CARRY,CL    ;Store carry in memory location(1106H).
1014   F4                      HLT              ;Halt program execution.

1015                    CODE ENDS               ;End of code segment.
                        END                     ;Assembly end.
```

Sample Data

Input Data : Data1 = F048$_H$
Data2 = 1052$_H$

Memory address	Content
1100	48
1101	F0
1102	52
1103	10

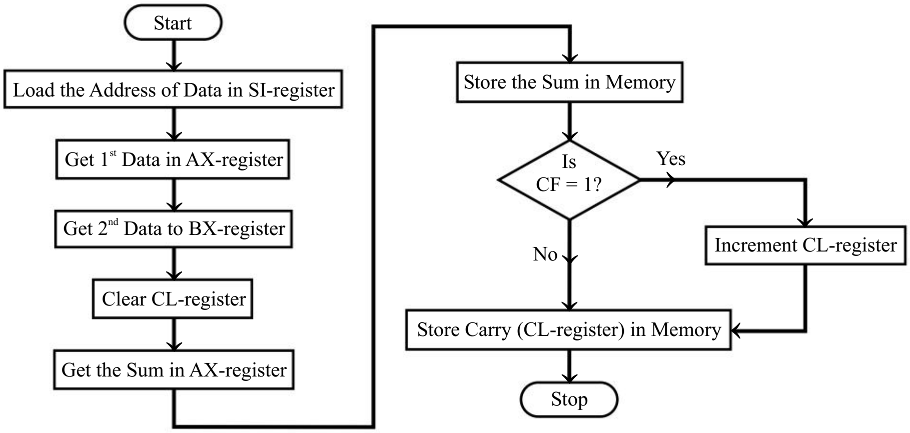
Output Data : Sum = 009A$_H$
Carry = 01$_H$

Memory address	Content
1104	9A
1105	00
1106	01

Algorithm (Method - III)

1. Set SI-register as pointer for data.
2. Get the first data in AX-register.
3. Get the second data in BX-register.
4. Clear CL-register.
5. Get the sum in AX-register.
6. Store the sum in memory.
7. Check for carry. If carry flag is set then go to next step, otherwise go to step 9.
8. Increment CL-register.
9. Store the carry in memory.
10. Stop.

Flowchart (Method - III)


Start
Load the Address of Data in SI-register
Get 1st Data in AX-register
Get 2nd Data to BX-register
Clear CL-register
Get the Sum in AX-register
Store the Sum in Memory
Is CF = 1?
Yes
No
Increment CL-register
Store Carry (CL-register) in Memory
Stop

Assembly language program (Method - III)

```
;PROGRAM TO ADD TWO 16-BIT DATA (METHOD-3)

CODE SEGMENT                ;Start of code segment.

       ASSUME CS:CODE       ;Assembler directive.
       ORG  1000H           ;specify program starting address.

       MOV  SI,1100H        ;Set SI-register as pointer for data.
       MOV  AX,[SI]         ;Get the first data in AX-register.
       MOV  BX,[SI+2]       ;Get the second data in BX-register.
       MOV  CL,00H          ;Clear the CL-register for carry.
       ADD  AX,BX           ;Add the two data, sum will be in AX-register.
       MOV  [SI+4],AX       ;Store the sum in memory location (1104H).
       JNC  AHEAD           ;Check the status of carry flag.
       INC  CL              ;If carry flag is set,increment CL by one.
AHEAD: MOV  [SI+6],CL       ;Store carry in memory location (1106H).
       HLT                  ;Halt program execution.

       CODE ENDS            ;End of code segment.
END                         ;Assembly end.
```

Assembler listing for example program 1 (Method - III)

```
;PROGRAM TO ADD TWO 16-BIT DATA (METHOD-3)

0000              CODE SEGMENT           ;Start of code segment.

                        ASSUME CS:CODE   ;Assembler directive.
1000                    ORG  1000H       ;specify program starting address.

1000   BE 1100          MOV  SI,1100H    ;Set SI-register as pointer for data.
1003   8B 04            MOV  AX,[SI]     ;Get the first data in AX-register.
1005   8B 5C 02         MOV  BX,[SI+2]   ;Get the second data in BX-register.
1008   B1 00            MOV  CL,00H      ;Clear the CL-register for carry.
100A   03 C3            ADD  AX,BX       ;Add the two data, sum will be in AX-register.
100C   89 44 04         MOV  [SI+4],AX   ;Store the sum in memory location (1104H).
100F   73 02            JNC  AHEAD       ;Check the status of carry flag.
1011   FE C1            INC  CL          ;If carry flag is set,increment CL by one.
1013   88 4C 06   AHEAD: MOV [SI+6],CL   ;Store carry in memory location (1106H).
```

```
1016    F4              HLT             ;Halt program execution.

1017            CODE    ENDS            ;End of code segment.
                END                     ;Assembly end.
```

Sample Data

Input Data : Data1 = $F048_H$
Data2 = 1052_H

Output Data : Sum = $009A_H$
Carry = 01_H

Memory address	Content
1100	48
1101	F0
1102	52
1103	10

Memory address	Content
1104	9A
1105	00
1106	01

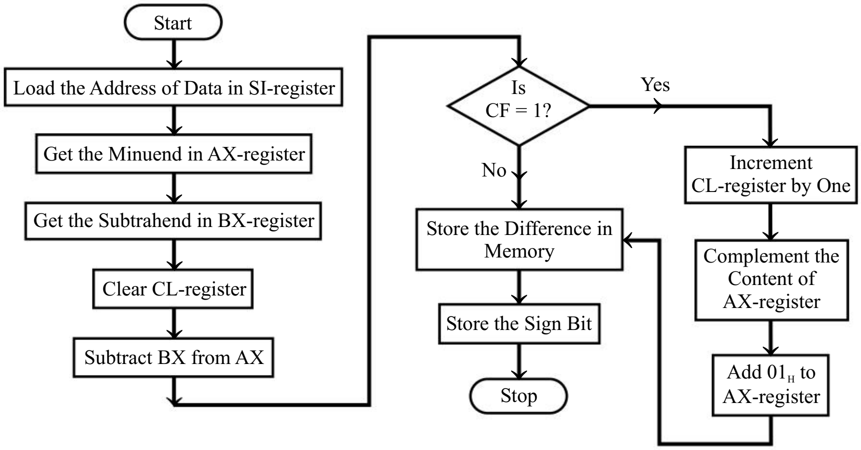
EXAMPLE PROGRAM 2 : 16-Bit Subtraction

Write an assembly language program to subtract two numbers of 16-bit data. Store the magnitude of the result in the memory. In one of the memory locations store 00_H to indicate positive result or store 01_H to indicate negative result.

Problem Analysis

To perform subtraction in 8086, one of the data should be stored in a register and another data should be stored in the register or memory. After subtraction the result will be available in the destination register/memory. The 8086 will perform 2's complement subtraction and then complement the carry. Therefore, if the result is negative then the carry flag is set and the destination register/memory will have 2's complement of the result. Hence, one of the register is used to account for sign of the result. To get the magnitude of the result again take 2's complement of the result.

Flowchart


Start
Load the Address of Data in SI-register
Get the Minuend in AX-register
Get the Subtrahend in BX-register
Clear CL-register
Subtract BX from AX
Is CF = 1?
Yes
No
Increment CL-register by One
Complement the Content of AX-register
Add 01_H to AX-register
Store the Difference in Memory
Store the Sign Bit
Stop

Algorithm

1. Set SI-register as pointer for data.
2. Get the minuend in AX-register.
3. Get the subtrahend in BX-register.
4. Clear CL-register to account for sign.
5. Subtract the content of BX from AX, the difference will be in AX.
6. Check for carry. If carry flag is set then go to next step, otherwise go to step 9.
7. Increment CL-register by one.

8. Take 2's complement of the difference in AX-register (For this complement AX and add one).
9. Store the magnitude of difference in memory.
10. Store the sign bit in memory.
11. Stop.

Assembly language program

```
;PROGRAM TO SUBTRACT TWO 16-BIT DATA

CODE SEGMENT            ;Start of code segment.

       ASSUME CS:CODE   ;Assembler directive.
       ORG 1000H        ;specify program starting address.

       MOV SI,1100H     ;Load the address of data in SI-register.
       MOV AX,[SI]      ;Get the minuend in AX-register.
       MOV BX,[SI+2]    ;Get the subtrahend in BX-register.
       MOV CL,00H       ;Clear the CL-register to account for sign.
       SUB AX,BX        ;Get the difference in AX-register.
       JNC STORE        ;Check the status of carry flag.
       INC CL           ;If carry flag is set,increment CL by one,
       NOT AX           ;then take 2's complement of difference.
       ADD AX,0001H

STORE: MOV [SI+4],AX    ;Store difference in memory location (1104H).
       MOV [SI+6],CL    ;Store sign bit in memory location (1106H).
       HLT              ;Halt program execution.

       CODE ENDS        ;End of code segment.
END                     ;Assembly end.
```

Assembler listing for example program 2

```
;PROGRAM TO SUBTRACT TWO 16-BIT DATA

0000               CODE SEGMENT            ;Start of code segment.

                          ASSUME CS:CODE   ;Assembler directive.
1000                      ORG  1000H       ;specify program starting address.

1000   BE 1100            MOV  SI,1100H    ;Load the address of data in SI-register.
1003   8B 04              MOV  AX,[SI]     ;Get the minuend in AX-register.
1005   8B 5C 02           MOV  BX,[SI+2]   ;Get the subtrahend in BX-register.
1008   B1 00              MOV  CL,00H      ;Clear the CL-register to account for sign.
100A   2B C3              SUB  AX,BX       ;Get the difference in AX-register.
100C   73 07              JNC  STORE       ;Check the status of carry flag.
100E   FE C1              INC  CL          ;If carry flag is set,increment CL by one,
1010   F7 D0              NOT  AX          ;then take 2's complement of the difference.
1012   05 0001            ADD  AX,0001H

1015   89 44 04    STORE: MOV  [SI+4],AX   ;Store difference in memory location (1104H).
1018   88 4C 06           MOV  [SI+6],CL   ;Store sign bit in memory location (1106H).
101B   F4                 HLT              ;Halt program execution.

101C               CODE ENDS               ;End of code segment.
                   END                     ;Assembly end.
```

Sample data

Input Data : Minuend = $840C_H$
Subtrahend = $B2CA_H$

Output Data : Difference = $2EBE_H$
Sign Bit = 01_H

Memory address	Content
1100	0C
1101	84
1102	CA
1103	B2
1104	BE
1105	2E
1106	01

EXAMPLE PROGRAM 3 : Multibyte Addition

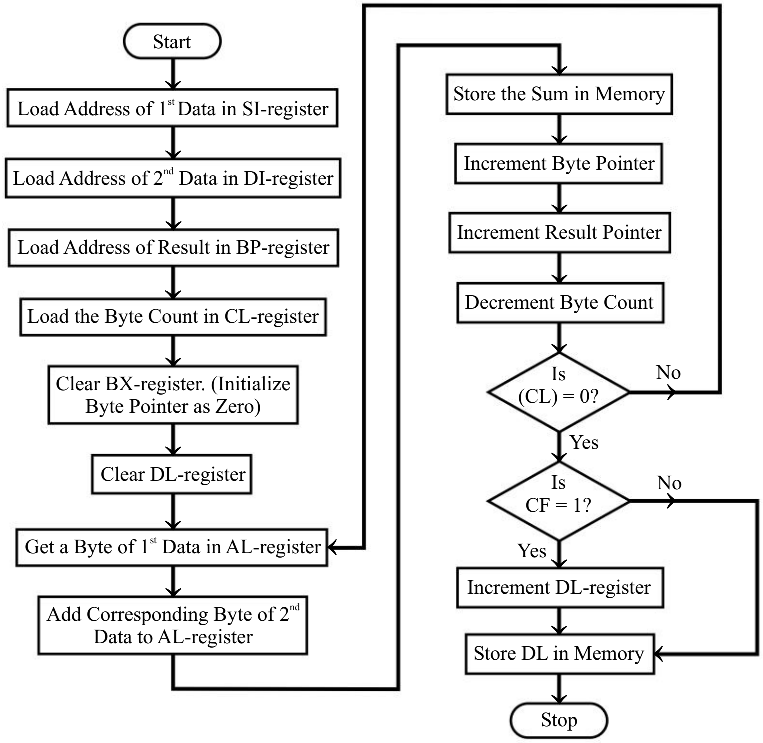
Write an assembly language program to add two numbers of multibyte data.

Problem Analysis

In the 8086 processor, the multibyte data can be added either byte by byte or word by word. The number of bytes in the data can be used as a count for the number of additions. One of the register is used to account for the final carry.

To perform addition we require three address pointers : Two pointers for input data and one pointer for output data.

Flowchart


Start
Load Address of 1st Data in SI-register
Load Address of 2nd Data in DI-register
Load Address of Result in BP-register
Load the Byte Count in CL-register
Clear BX-register. (Initialize Byte Pointer as Zero)
Clear DL-register
Get a Byte of 1st Data in AL-register
Add Corresponding Byte of 2nd Data to AL-register
Store the Sum in Memory
Increment Byte Pointer
Increment Result Pointer
Decrement Byte Count
Is (CL) = 0?
No
Yes
Is CF = 1?
No
Yes
Increment DL-register
Store DL in Memory
Stop

Algorithm

1. Load the starting address of 1st data in SI-register.
2. Load the starting address of 2nd data in DI-register.
3. Load the starting address of result in BP-register.
4. Load the byte count in CL-register.
5. Let BX-register be byte pointer. Initialize byte pointer as zero.
6. Clear DL-register to account for final carry.
7. Clear carry flag (i.e., initial carry is zero).
8. Load a byte of 1st data in AL-register.
9. Add the corresponding byte of 2nd data in memory to AL-register along with previous carry.
10. Store the sum in memory.
11. Increment the byte pointer (BX) and result pointer (BP).
12. Decrement the byte count (CL).

13. If byte count (CL) is zero go to next step, otherwise go to step 8.
14. Check for carry. If carry flag is set then go to next step, otherwise go to step 16.
15. Increment DL-register.
16. Store final carry in memory.
17. Stop.

Assembly language program

```
;PROGRAM TO ADD TWO MULTIBYTE DATA

CODE SEGMENT            ;Start of code segment.

        ASSUME CS:CODE  ;Assembler directive.
        ORG  1000H      ;specify program starting address.

        MOV  SI,1100H   ;Set SI-register as pointer for 1st data.
        MOV  DI,1201H   ;Set DI-register as pointer for 2nd data.
        MOV  BP,1301H   ;Set BP-register as pointer for result.
        MOV  CL,[SI]    ;Load the count for number of bytes in CL.
        INC  SI         ;Set SI to point to 1st byte of 1st data.
        MOV  BX,00H     ;Initialize byte pointer as zero.
        MOV  DL,00H     ;Initialize final carry as zero.
        CLC             ;Clear carry flag.

REPEAT: MOV  AL,[SI+BX] ;Get a byte of 1st data in AL-register.
        ADC  AL,[DI+BX] ;Add the corresponding byte of 2nd data to AL.
        MOV  [BP],AL    ;Store sum of corresponding bytes in memory.
        INC  BX         ;Increment the byte pointer.
        INC  BP         ;Increment the result pointer.
        LOOP REPEAT     ;Repeat addition until byte count is zero.

        JNC  AHEAD      ;Check for final carry.
        INC  DL         ;If carry flag is set then increment DL.
AHEAD:  MOV  [BP],DL    ;Store the final carry in memory.
        HLT             ;Halt program execution.

        CODE ENDS       ;End of code segment.
END                     ;Assembly end.
```

Assembler listing for example program 3

```
;PROGRAM TO ADD TWO MULTIBYTE DATA

0000              CODE SEGMENT            ;Start of code segment.

                          ASSUME CS:CODE  ;Assembler directive.
1000                      ORG  1000H      ;specify program starting address.

1000   BE 1100            MOV  SI,1100H   ;Set  SI-register as pointer for 1st data.
1003   BF 1201            MOV  DI,1201H   ;Set  DI-register as pointer for 2nd data.
1006   BD 1301            MOV  BP,1301H   ;Set  BP-register as pointer for result.
1009   8A 0C              MOV  CL,[SI]    ;Load the count for number of bytes in CL.
100B   46                 INC  SI         ;Set SI to point to 1st byte of 1st data.
100C   BB 0000            MOV  BX,00H     ;Initialize byte pointer as zero.
100F   B2 00              MOV  DL,00H     ;Initialize final carry as zero.
1011   F8                 CLC             ;Clear carry flag.

1012   8A 00      REPEAT: MOV  AL,[SI+BX] ;Get a byte of 1st data in AL-register.
1014   12 01              ADC  AL,[DI+BX] ;Add corresponding byte of 2nd data to AL.
1016   88 46 00           MOV  [BP],AL    ;Store sum of corresponding bytes in memory.
1019   43                 INC  BX         ;Increment the byte pointer.
101A   45                 INC  BP         ;Increment the result pointer.
101B   E2 F5              LOOP REPEAT     ;Repeat addition until byte count is zero.
```

```
101D   73 02                JNC  AHEAD      ;Check for final carry.
101F   FE C2                INC  DL         ;If carry flag is set then increment DL.
1021   88 56 00    AHEAD:   MOV  [BP],DL    ;Store the final carry in memory.
1024   F4                   HLT             ;Halt program execution.

1025               CODE ENDS                ;End of code segment.
                    END                     ;Assembly end.
```

Sample data

1st Data : F5C2647217$_H$

Memory address	Content
1100	05
1101	17
1102	72
1103	64
1104	C2
1105	F5

2nd Data : C265750712$_H$

Memory address	Content
1200	05
1201	12
1202	07
1203	75
1204	65
1205	C2

Output Data : 01B827D97929$_H$

Memory address	Content
1301	29
1302	79
1303	D9
1304	27
1305	B8
1306	01

EXAMPLE PROGRAM 4 : Multibyte Subtraction

Write an assembly language program to subtract two numbers of multibyte data.

Problem Analysis

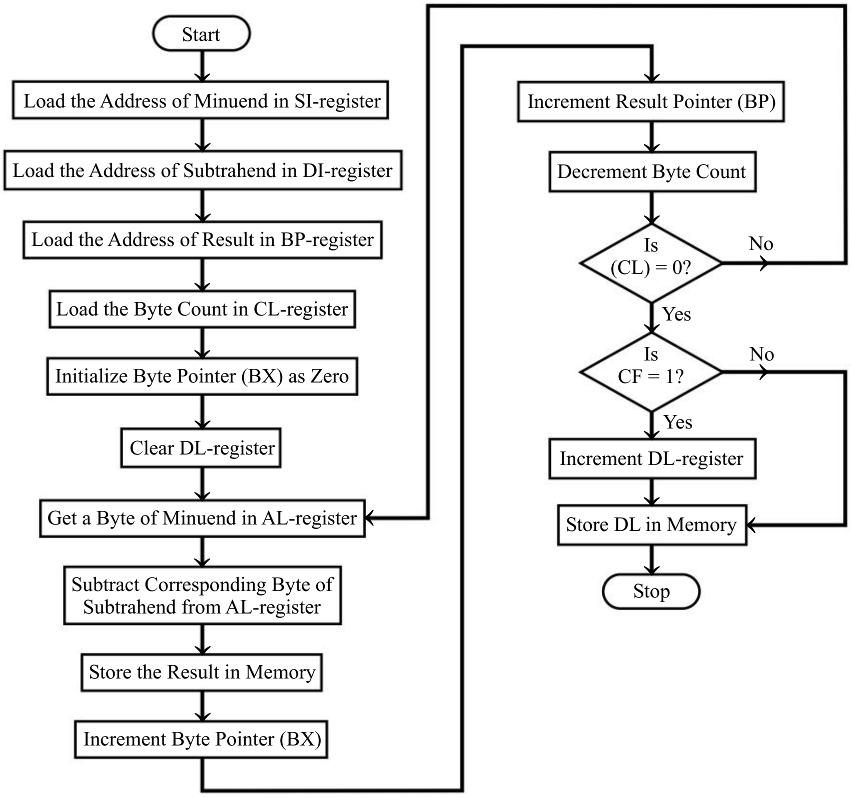
In the 8086 processor, the multibyte data can be subtracted either byte by byte or word by word. The number of bytes in the data can be used as count for number of subtractions. One of the registers is used to account for the final borrow.

To perform subtraction we require three pointers : Two pointers for input data and one pointer for output data.

Algorithm

1. Load the starting address of minuend in SI-register.
2. Load the starting address of subtrahend in DI-register.
3. Load the starting address of result in BP-register.
4. Load the byte count in CL-register.
5. Let BX-register be byte pointer. Initialize byte pointer as zero.
6. Clear DL-register to account for final borrow.
7. Clear carry flag, i.e., initial borrow is zero.
8. Load a byte of minuend in AL-register.
9. Subtract the corresponding byte of subtrahend in memory from AL-register along with previous borrow.
10. Store the difference in memory.
11. Increment the byte pointer (BX) and result pointer (BP).
12. Decrement the byte count (CL).
13. If byte count (CL) is zero then go to next step, otherwise go to step 8.
14. Check for carry flag, if carry flag is set then go to next step, otherwise go to step 16.
15. Increment DL-register.
16. Store final borrow in memory.
17. Stop.

Flowchart


Start
Load the Address of Minuend in SI-register
Load the Address of Subtrahend in DI-register
Load the Address of Result in BP-register
Load the Byte Count in CL-register
Initialize Byte Pointer (BX) as Zero
Clear DL-register
Get a Byte of Minuend in AL-register
Subtract Corresponding Byte of Subtrahend from AL-register
Store the Result in Memory
Increment Byte Pointer (BX)
Increment Result Pointer (BP)
Decrement Byte Count
Is (CL) = 0?
No
Yes
Is CF = 1?
No
Yes
Increment DL-register
Store DL in Memory
Stop

Assembly language program

```
;PROGRAM TO SUBTRACT TWO MULTIBYTE DATA

        CODE SEGMENT      ;Start of code segment.

        ASSUME CS:CODE    ;Assembler directive.
        ORG   1000H       ;specify program starting address.

        MOV   SI,1100H    ;Set SI-register as pointer for minuend.
        MOV   DI,1201H    ;Set DI-register as pointer for subtrahend.
        MOV   BP,1301H    ;Set BP-register as pointer for result.
        MOV   CL,[SI]     ;Load the count for number of bytes in CL.
        INC   SI          ;Set SI to point to 1st byte of minuend.
        MOV   BX,00H      ;Initialize byte pointer as zero.
        MOV   DL,00H      ;Initialize final borrow as zero.
        CLC               ;Clear carry flag.

REPEAT: MOV   AL,[SI+BX]  ;Get a byte of minuend in AL-register.
        SBB   AL,[DI+BX]  ;Subtract corresponding byte of subtrahend.
        MOV   [BP],AL     ;Store difference in memory.
        INC   BX          ;Increment the byte pointer.
        INC   BP          ;Increment the result pointer.
        LOOP  REPEAT      ;Repeat subtraction until byte count is zero.

        JNC   AHEAD       ;Check for final borrow.
        INC   DL          ;If carry flag is set then increment DL.
```

```
AHEAD:  MOV   [BP],DL       ;Store the final borrow in memory.
        HLT                 ;Halt program execution.

CODE ENDS                   ;End of code segment.
END                         ;Assembly end.
```

Assembler listing for example program 4

```
;PROGRAM TO SUBTRACT TWO MULTIBYTE DATA

0000                CODE SEGMENT              ;Start of code segment.

                            ASSUME CS:CODE ;Assembler directive.
1000                        ORG   1000H    ;Specify program starting address.

1000   BE 1100              MOV   SI,1100H ;Set SI-register as pointer for minuend.
1003   BF 1201              MOV   DI,1201H ;Set DI-register as pointer for subtrahend.
1006   BD 1301              MOV   BP,1301H ;Set BP-register as pointer for result.
1009   8A 0C                MOV   CL,[SI]  ;Load the count for number of bytes in CL.
100B   46                   INC   SI       ;Set SI to point to 1st byte of minuend.
100C   BB 0000              MOV   BX,00H   ;Initialize byte pointer as zero.
100F   B2 00                MOV   DL,00H   ;Initialize final borrow as zero.
1011   F8                   CLC            ;Clear carry flag.

1012   8A 00        REPEAT: MOV   AL,[SI+BX];Get a byte of minuend in AL-register.
1014   1A 01                SBB   AL,[DI+BX];Subtract corresponding byte of subtrahend.

1016   88 46 00             MOV   [BP],AL  ;Store difference in memory.
1019   43                   INC   BX       ;Increment the byte pointer.
101A   45                   INC   BP       ;Increment the result pointer.
101B   E2 F5                LOOP  REPEAT   ;Repeat subtraction until byte count is zero.

101D   73 02                JNC   AHEAD    ;Check for final borrow.
101F   FE C2                INC   DL       ;If carry flag is set then increment DL.
1021   88 56 00     AHEAD:  MOV   [BP],DL  ;Store the final borrow in memory.
1024   F4                   HLT            ;Halt program execution.

1025                CODE ENDS              ;End of code segment.
                     END                   ;Assembly end.
```

Sample Data

Minuend : D2564A6756$_H$

Memory address	Content
1100	05
1101	56
1102	67
1103	4A
1104	56
1105	D2

Subtrahend : F2C579F2E7$_H$

Memory address	Content
1200	05
1201	E7
1202	F2
1203	79
1204	C5
1205	F2

Output Data : 01DF90D0746F

Memory address	Content
1301	6F
1302	74
1303	D0
1304	90
1305	DF
1306	01

EXAMPLE PROGRAM 5 : Sum of an Array

Write an assembly language program to determine the sum of elements in an array.

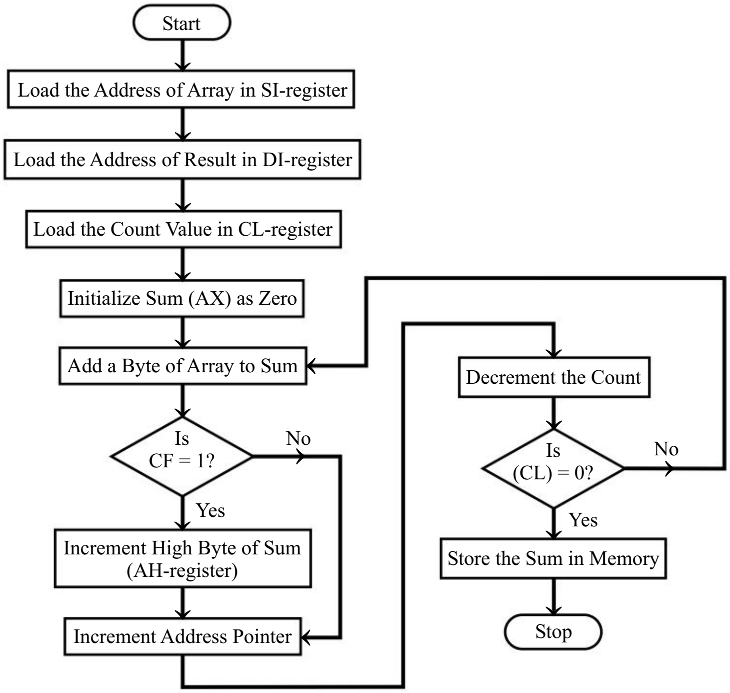
Problem Analysis

Declare the content of one of the register as sum and take initial value of sum as zero. The sum of elements of array can be obtained by adding the elements of array one by one (i.e., byte by byte) to sum. The number of bytes in the array can be used as count for number of additions to be performed. The carry in each addition can be separately added in a register and saved as high byte of sum.

Algorithm

1. Load the address of the array in SI-register.
2. Load the address of the result in DI-register.
3. Load the count value in CL-register.
4. Let the content of AX be sum and keep initial sum as zero.
5. Add a byte of array to sum.
6. Check for carry. If carry flag is set then go to next step otherwise go to step 8.
7. Increment high byte of sum (AH-register).
8. Increment the array pointer.
9. Decrement the count (CL-register).
10. If count (CL) is zero then go to next step, otherwise go to step 5.
11. Store the 16-bit sum in memory.
12. Stop.

Flowchart


Start
Load the Address of Array in SI-register
Load the Address of Result in DI-register
Load the Count Value in CL-register
Initialize Sum (AX) as Zero
Add a Byte of Array to Sum
Is CF = 1?
No
Yes
Increment High Byte of Sum (AH-register)
Increment Address Pointer
Decrement the Count
Is (CL) = 0?
No
Yes
Store the Sum in Memory
Stop

Assembly language program

```
;PROGRAM TO FIND THE SUM OF THE ELEMENTS IN AN ARRAY

CODE SEGMENT           ;Start of code segment.

       ASSUME CS:CODE  ;Assembler directive.
       ORG  1000H      ;specify program starting address.

       MOV  SI,1100H   ;Set SI-register as pointer for array.
       MOV  DI,1200H   ;Set DI-register as pointer for result.
       MOV  CL,[SI]    ;Set CL as count for number of bytes in array.
       INC  SI         ;Set SI to point to 1st byte of array.
       MOV  AX,0000H   ;Set initial sum as zero.
```

```
AGAIN: ADD  AL,[SI]     ;Add a byte of array to sum.
       JNC  AHEAD       ;Check for carry flag.
       INC  AH          ;If carry flag is set then increment AH.
AHEAD: INC  SI          ;Increment array pointer.
       LOOP AGAIN       ;Repeat addition until count is zero.

       MOV  [DI],AX     ;Store the sum in memory.
       HLT              ;Halt program execution.

       CODE ENDS        ;End of code segment.
END                     ;Assembly end.
```

Assembler listing for example program 5

```
;PROGRAM TO FIND THE SUM OF THE ELEMENTS IN AN ARRAY

0000              CODE SEGMENT       ;Start of code segment.

                      ASSUME CS:CODE ;Assembler directive.
1000                  ORG  1000H     ;specify program starting address.

1000   BE 1100        MOV  SI,1100H  ;Set SI-register as pointer for array.
1003   BF 1200        MOV  DI,1200H  ;Set DI-register as pointer for result.
1006   8A 0C          MOV  CL,[SI]   ;Set CL as count for number of bytes in array.
1008   46             INC  SI        ;Set SI to point to 1st byte of array.
1009   B8 0000        MOV  AX,0000H  ;Set initial sum as zero.
100C   02 04   AGAIN: ADD  AL,[SI]   ;Add a byte of array to sum.
100E   73 02          JNC  AHEAD     ;Check for carry flag.
1010   FE C4          INC  AH        ;If carry flag is set then increment AH.
1012   46      AHEAD: INC  SI        ;Increment array pointer.
1013   E2 F7          LOOP AGAIN     ;Repeat addition until count is zero.

1015   89 05          MOV  [DI],AX   ;Store the sum in memory.
1017   F4             HLT            ;Halt program execution.

1018              CODE ENDS          ;End of code segment.
                  END                ;Assembly end.
```

Sample data

Input Data :06
12
47
C2
F5
47
56

Output Data : $02AD_H$

Memory address	Content	
1100	06	←Count
1101	12	Array
1102	47	
1103	C2	
1104	F5	
1105	47	
1106	56	
1200	AD	Sum
1201	02	

EXAMPLE PROGRAM 6 : BCD Addition

Write an assembly language program to add two numbers of BCD data.

Problem Analysis

The 8086 processor 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 the binary addition the following correction should be made to get the result in BCD :

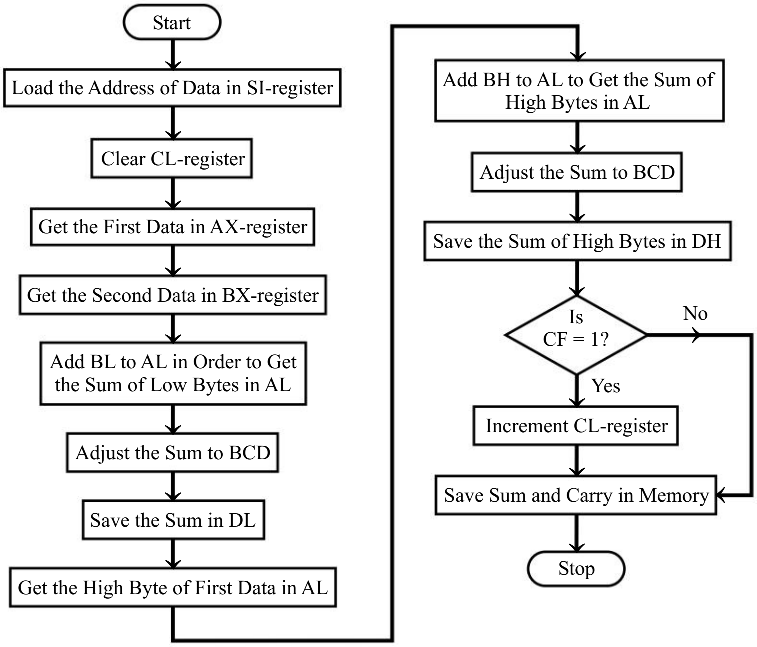
1. If the sum of lower nibble exceeds 9 or if there is auxiliary carry then 6 is added to lower nibble.
2. If the sum of upper nibble exceeds 9 or if there is carry then 6 is added to upper nibble.

The above correction is taken care of by the DAA(**D**ecimal **A**djust **A**ccumulator) instruction. Therefore after binary addition, execute the DAA instruction to do the above correction in the sum.

Algorithm

1. Load the address of data in SI-register.
2. Clear CL-register to account for carry.
3. Load the first data in AX-register and second data in BX-register.
4. Perform binary addition of low byte of data to get the sum in AL-register.
5. Adjust the sum of low bytes to BCD.
6. Save the sum of low bytes in DL-register.
7. Get the high byte of first data in AL-register.
8. Add the high byte of second data and previous carry to AL-register. Now the sum of high bytes will be in AL-register.
9. Adjust the sum of high bytes to BCD.
10. Save the sum of high bytes in DH-register.
11. Check for carry. If carry flag is set then go to next step, otherwise go to step 13.
12. Increment CL-register.
13. Save the sum (DX-register) in memory.
14. Save the carry (CL-register) in memory.
15. Stop.

Flowchart


Start
Load the Address of Data in SI-register
Clear CL-register
Get the First Data in AX-register
Get the Second Data in BX-register
Add BL to AL in Order to Get the Sum of Low Bytes in AL
Adjust the Sum to BCD
Save the Sum in DL
Get the High Byte of First Data in AL
Add BH to AL to Get the Sum of High Bytes in AL
Adjust the Sum to BCD
Save the Sum of High Bytes in DH
Is CF = 1?
No
Yes
Increment CL-register
Save Sum and Carry in Memory
Stop

Assembly language program

```
;PROGRAM TO ADD TWO BCD DATA

CODE SEGMENT            ;Start of code segment.

        ASSUME CS:CODE  ;Assembler directive.
        ORG  1000H      ;specify program starting address.
        MOV  SI,1100H   ;Set SI-register as pointer for data.
        MOV  CL,00H     ;Clear CL-register.
        MOV  AX,[SI]    ;Get first data in AX-register.
        MOV  BX, [SI+2] ;Get second data in BX-register.

        ADD  AL,BL      ;Get sum of low bytes in AL-register.
        DAA             ;Adjust the sum to BCD.
        MOV  DL,AL      ;Save sum of low bytes in DL-register.

        MOV  AL,AH      ;Move high byte of first data to AL.
        ADC  AL,BH      ;Get sum of high bytes in AL-register.
        DAA             ;Adjust the sum to BCD.
        MOV  DH,AL      ;Save sum of high bytes in DH-register.

        JNC  AHEAD      ;Check for carry flag.
        INC  CL         ;If carry flag is set then increment CL.
AHEAD:  MOV  [SI+4],DX  ;Store the sum in memory.
        MOV  [SI+6],CL  ;Store the carry in memory.
        HLT             ;Halt program execution.

CODE ENDS               ;End of code segment.
END                     ;Assembly end.
```

Assembler listing for example program 6

```
;PROGRAM TO ADD TWO BCD DATA

0000          CODE SEGMENT                ;Start of code segment.

                                ASSUME CS:CODE ;Assembler directive.
1000                            ORG 1000H      ;specify program starting address.

1000  BE 1100                   MOV SI,1100H   ;Set SI-register as pointer for data.
1003  B1 00                     MOV CL,00H     ;Clear CL-register.
1005  8B 04                     MOV AX,[SI]    ;Get first data in AX-register.
1007  8B 5C 02                  MOV BX, [SI+2] ;Get second data in BX-register.

100A  02 C3                     ADD AL,BL      ;Get sum of low bytes in AL-register.
100C  27                        DAA            ;Adjust the sum to BCD.
100D  8A D0                     MOV DL,AL      ;Save sum of low bytes in DL-register.

100F  8A C4                     MOV AL,AH      ;Move high byte of first data to AL.
1011  12 C7                     ADC AL,BH      ;Get sum of high bytes in AL-register.
1013  27                        DAA            ;Adjust the sum to BCD.
1014  8A F0                     MOV DH,AL      ;Save sum of high bytes in DH-register.

1016  73 02                     JNC AHEAD      ;Check for carry flag.
1018  FE C1                     INC CL         ;If carry flag is set then increment CL.
101A  89 54 04          AHEAD:  MOV [SI+4],DX  ;Store the sum in memory.
101D  88 4C 06                  MOV [SI+6],CL  ;Store the carry in memory.
1020  F4                        HLT            ;Halt program execution.

1021                          CODE ENDS        ;End of code segment.
                               END             ;Assembly end.
```

Sample data

Input Data : Data1 = 4578_{10} **Output Data** : 013176_{10}

Data2 = 8598_{10}

Memory address	Content	
1100	78	Data 1
1101	45	
1102	98	Data 2
1103	85	
1104	76	Sum
1105	31	
1106	01	←Carry

EXAMPLE PROGRAM 7 : BCD Subtraction

Write an assembly language program to subtract two numbers of BCD data.

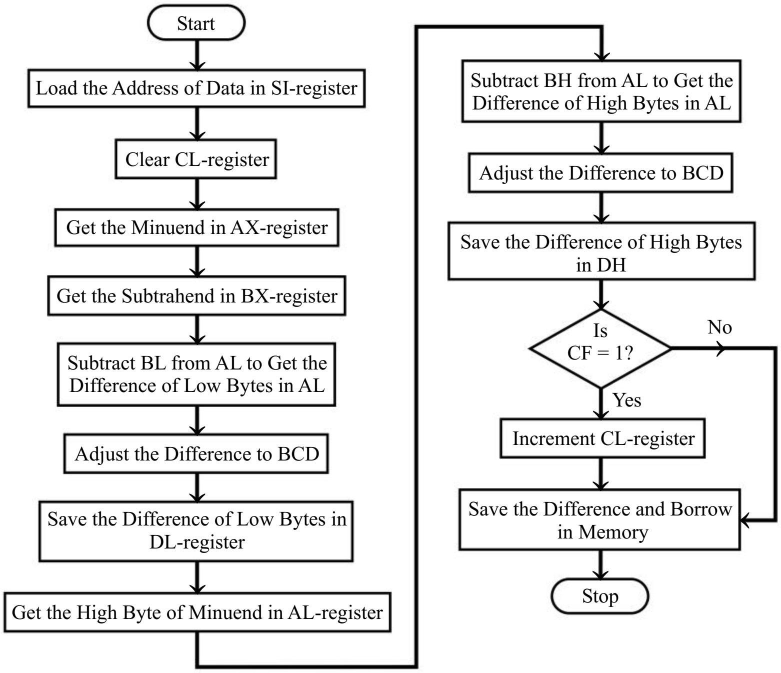
Problem Analysis

The 8086 processor will perform only binary subtraction. Hence for BCD subtraction, the binary subtraction of BCD data is performed and then the difference is corrected to get the result in BCD. After binary subtraction the following correction should be made to get the result in BCD.

1. If the difference of lower nibble exceeds 9 or if there is auxiliary carry then 6 is subtracted from lower nibble.
2. If the difference of upper nibble exceeds 9 or if there is carry then 6 is subtracted from upper nibble.

The above correction is taken care by the DAS (**D**ecimal **A**djust after **S**ubtraction) instruction. Therefore, after binary subtraction, execute DAS to do the above correction in the difference.

Flowchart


Start
Load the Address of Data in SI-register
Clear CL-register
Get the Minuend in AX-register
Get the Subtrahend in BX-register
Subtract BL from AL to Get the Difference of Low Bytes in AL
Adjust the Difference to BCD
Save the Difference of Low Bytes in DL-register
Get the High Byte of Minuend in AL-register
Subtract BH from AL to Get the Difference of High Bytes in AL
Adjust the Difference to BCD
Save the Difference of High Bytes in DH
Is CF = 1?
No
Yes
Increment CL-register
Save the Difference and Borrow in Memory
Stop

Algorithm

1. Load the address of data in SI-register.
2. Clear CL-register to account for borrow.
3. Load the minuend in AX-register.
4. Get the subtrahend in BX-register.
5. Subtract BL from AL to get the difference of low bytes in AL.
6. Adjust the difference of low bytes to BCD and then save it in DL-register.
7. Get the high byte of minuend in AL-register.
8. Subtract BH and the previous borrow from AL to get the difference of high bytes in AL-register.
9. Adjust the difference of high bytes to BCD and save it in DH-register.
10. Check for carry. If carry flag is set then go to next step, otherwise go to step 12.
12. Increment CL-register.
13. Save the difference (DX-register) and the borrow (CL-register) in memory.
14. Stop.

Assembly language program

```
;PROGRAM TO SUBTRACT TWO BCD DATA

CODE SEGMENT            ;Start of code segment.

        ASSUME CS:CODE  ;Assembler directive.
        ORG  1000H      ;specify program starting address.
        MOV  SI,1100H   ;Set SI-register as pointer for data.
        MOV  CL,00H     ;Clear CL-register.
        MOV  AX,[SI]    ;Get minuend in AX-register.
        MOV  BX,[SI+2]  ;Get subtrahend in BX-register.
        SUB  AL,BL      ;Get difference of low bytes in AL-register.
        DAS             ;Adjust the difference to BCD.
        MOV  DL,AL      ;Save difference of low bytes in DL-register.
        MOV  AL,AH      ;Move high byte of minuend to AL-register.
        SBB  AL,BH      ;Get difference of high bytes in AL-register.
        DAS             ;Adjust the difference to BCD.
        MOV  DH,AL      ;Save difference of high bytes in DH-register.
        JNC  AHEAD      ;Check for carry flag.
        INC  CL         ;If carry flag is set then increment CL.
AHEAD:  MOV  [SI+4],DX  ;Store the difference in memory.
        MOV  [SI+6],CL  ;Store the borrow in memory.
        HLT             ;Halt program execution.

CODE ENDS               ;End of code segment.
END                     ;Assembly end.
```

Assembler listing for example program 7

```
;PROGRAM TO SUBTRACT TWO BCD DATA

0000              CODE SEGMENT            ;Start of code segment.

                         ASSUME  CS:CODE ;Assembler directive.
1000                     ORG  1000H      ;specify program starting address.
1000   BE 1100           MOV  SI,1100H   ;Set SI-register as pointer for data.
1003   B1 00             MOV  CL,00H     ;Clear CL-register.
1005   8B 04             MOV  AX,[SI]    ;Get minuend in AX-register.
1007   8B 5C 02          MOV  BX, [SI+2] ;Get subtrahend in BX-register.
100A   2A C3             SUB  AL,BL      ;Get difference of low bytes in AL-register.
100C   2F                DAS             ;Adjust the difference to BCD.
100D   8A D0             MOV  DL,AL      ;Save difference of low bytes in DL-register.
100F   8A C4             MOV  AL,AH      ;Move high byte of minuend to AL-register.
1011   1A C7             SBB  AL,BH      ;Get difference of high bytes in AL-register.
1013   2F                DAS             ;Adjust the difference to BCD.
1014   8A F0             MOV  DH,AL      ;Save difference of high bytes in DH-register.
1016   73 02             JNC  AHEAD      ;Check for carry flag.
1018   FE C1             INC  CL         ;If carry flag is set then increment CL.
101A   89 54 04   AHEAD: MOV  [SI+4],DX  ;Store the difference in memory.
101D   88 4C 06          MOV  [SI+6],CL  ;Store the borrow in memory.
1020   F4                HLT             ;Halt program execution.

1021              CODE ENDS               ;End of code segment.
                  END                     ;Assembly end.
```

Sample Data

Input Data : Minuend = 9572_{10}
Subtrahend = 4793_{10}

Output Data : 004779_{10}

Memory address	Content	
1100	72	Minuend
1101	95	
1102	93	Subtrahend
1103	47	
1104	79	Difference
1105	47	
1106	00	← Borrow

EXAMPLE PROGRAM 8 : Multiplication

Write an assembly language program to multiply two numbers of 16-bit data.

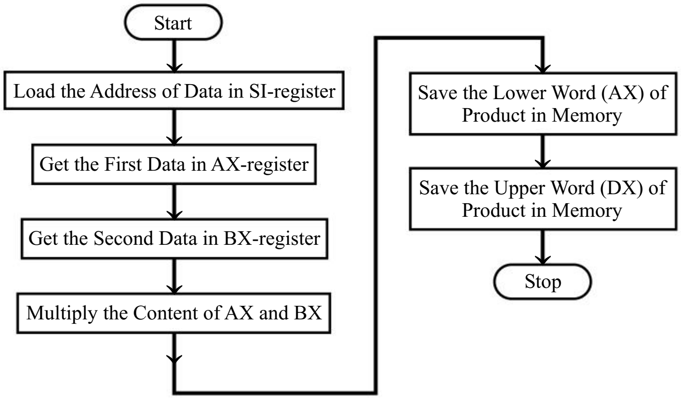
Problem Analysis

To perform multiplication in 8086 processor one of the data should be stored in AX-register and another data can be stored in the register/memory. After multiplication the product will be in AX and DX registers.

Algorithm

1. Load the address of data in SI-register.
2. Get the first data in AX-register.
3. Get the second data in BX-register.
4. Multiply the content of AX and BX. The product will be in AX and DX.
5. Save the product (AX and DX) in memory.
6. Stop.

Flowchart


Start
Load the Address of Data in SI-register
Get the First Data in AX-register
Get the Second Data in BX-register
Multiply the Content of AX and BX
Save the Lower Word (AX) of Product in Memory
Save the Upper Word (DX) of Product in Memory
Stop

Assembly language program

```
;PROGRAM TO MULTIPLY TWO 16-BIT DATA

CODE SEGMENT                ;Start of code segment.

          ASSUME CS:CODE ;Assembler directive.
          ORG 1000H      ;specify program starting address.

          MOV SI,1100H   ;Set SI as pointer for data.
          MOV AX,[SI]    ;Get the 1st data in AX-register.
          MOV BX,[SI+2]  ;Get the 2nd data in BX-register.
          MUL BX         ;Multiply AX and BX.
                         ;The product will be in AX and DX registers.
          MOV [SI+4],AX  ;Save the lower 16 bits of product in memory.
          MOV [SI+6],DX  ;Save the upper 16 bits of product in memory.
          HLT            ;Halt program execution.

CODE ENDS                ;End of code segment.
END                      ;Assembly end.
```

Assembler listing for example program 8

```
;PROGRAM TO MULTIPLY TWO 16-BIT DATA

0000                CODE SEGMENT             ;Start of code segment.

                            ASSUME CS:CODE   ;Assembler directive.
1000                        ORG  1000H       ;specify program starting address.

1000    BE 1100             MOV  SI,1100H    ;Set SI as pointer for data.
1003    8B 04               MOV  AX,[SI]     ;Get the 1st data in AX-register.
1005    8B 5C 02            MOV  BX,[SI+2]   ;Get the 2nd data in BX-register.
1008    F7 E3               MUL  BX          ;Multiply AX and BX.
                                             ;The product will be in AX and DX registers.
100A    89 44 04            MOV  [SI+4],AX   ;Save the lower 16 bits of product in memory.
100D    89 54 06            MOV  [SI+6],DX   ;Save the upper 16 bits of product in memory.
1010    F4                  HLT              ;Halt program execution.

1011                CODE ENDS                ;End of code segment.
                    END                      ;Assembly end.
```

Sample Data

Input Data : Data1 = $EF1A_H$
Data2 = $CD50_H$

Output Data : $BFC28A20_H$

Memory address	Content
1100	1A
1101	EF
1102	50
1103	CD

Memory address	Content
1104	20
1105	8A
1106	C2
1107	BF

EXAMPLE PROGRAM 9 : 32-Bit Multiplication

Write an assembly language program to multiply two numbers of 32-bit data.

Problem Analysis

In 8086 processor the 32-bit multiplication can be implemented in terms of 16-bit multiplication. The given data can be divided into two words (**L**ower **W**ord (LW) and **U**pper **W**ord (UW)) as shown below:

Data1 (32-bit) → $D1_{UW}$ (16-bit), $D1_{LW}$ (16-bit)

Data2 (32-bit) → $D2_{UW}$ (16-bit), $D2_{LW}$ (16-bit)

Then perform the following four multiplications. Each multiplication will give a 32-bit result which can be divided into **L**ower **W**ord (LW) and **U**pper **W**ord (UW) as shown below:

Product 1 (P1) : $D1_{LW} \times D2_{LW} = P1_{UW}, P1_{LW}$

Product 2 (P2) : $D1_{UW} \times D2_{LW} = P2_{UW}, P2_{LW}$

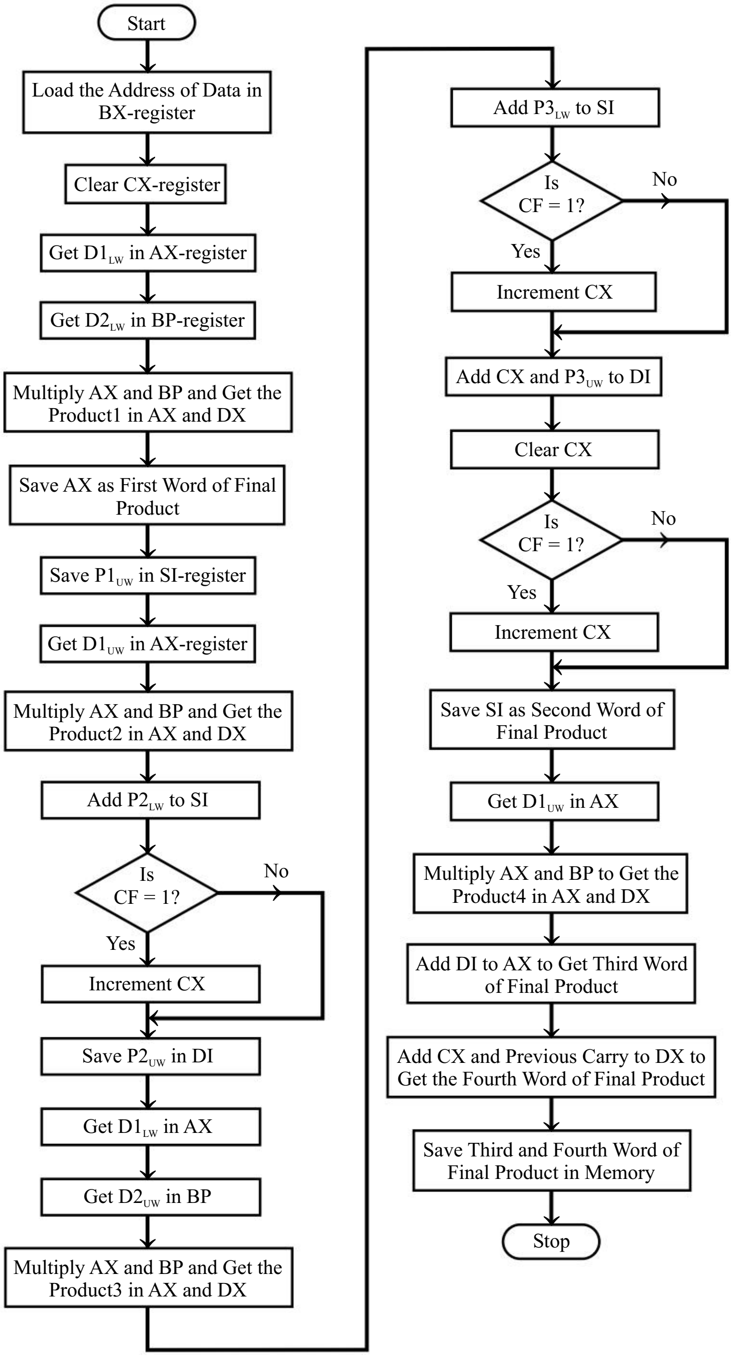
Product 3 (P3) : $D1_{LW} \times D2_{UW} = P3_{UW}, P3_{LW}$

Product 4 (P4) : $D1_{UW} \times D2_{UW} = P4_{UW}, P4_{LW}$

The result of the above four multiplications can be added to get the final result as shown below: The final product will have a size of four words and they are denoted as P_{W1}, P_{W2}, P_{W3} and P_{W4}.

$$\begin{array}{cccc} & & D1_{UW} & D1_{LW} \\ & & \times\, D2_{UW} & D2_{LW} \\ \cline{3-4} & & P1_{UW} & P1_{LW} \\ & P2_{UW} & P2_{LW} & \\ & P3_{UW} & P3_{LW} & \\ P4_{UW} & P4_{LW} & & \\ \hline P_{W4} & P_{W3} & P_{W2} & P_{W1} \\ \hline \end{array}$$

Flowchart


Start
Load the Address of Data in BX-register
Clear CX-register
Get D1$_{LW}$ in AX-register
Get D2$_{LW}$ in BP-register
Multiply AX and BP and Get the Product1 in AX and DX
Save AX as First Word of Final Product
Save P1$_{UW}$ in SI-register
Get D1$_{UW}$ in AX-register
Multiply AX and BP and Get the Product2 in AX and DX
Add P2$_{LW}$ to SI
Is CF = 1?
No
Yes
Increment CX
Save P2$_{UW}$ in DI
Get D1$_{LW}$ in AX
Get D2$_{UW}$ in BP
Multiply AX and BP and Get the Product3 in AX and DX
Add P3$_{LW}$ to SI
Is CF = 1?
No
Yes
Increment CX
Add CX and P3$_{UW}$ to DI
Clear CX
Is CF = 1?
No
Yes
Increment CX
Save SI as Second Word of Final Product
Get D1$_{UW}$ in AX
Multiply AX and BP to Get the Product4 in AX and DX
Add DI to AX to Get Third Word of Final Product
Add CX and Previous Carry to DX to Get the Fourth Word of Final Product
Save Third and Fourth Word of Final Product in Memory
Stop

Algorithm

1. Load the address of data in BX-register.
2. Clear CX-register to account for carry in additions.
3. Get $D1_{LW}$ in AX-register and $D2_{LW}$ in BP-register.
4. Multiply AX and BP to get product 1 in AX and DX.
5. Save AX as first word of final product.
6. Save $P1_{UW}$ in SI-register.
7. Get $D1_{UW}$ in AX-register.
8. Multiply AX and BP to get product 2 in AX and DX.
9. Add $P2_{LW}$ to SI-register.
10. Check for carry. If carry flag is set then go to next step, otherwise go to step 12.
11. Increment CX-register.
12. Save $P2_{UW}$ in DI-register.
13. Get $D1_{LW}$ in AX-register and $D2_{UW}$ in BP-register.
14. Multiply AX and BP to get product 3 in AX and DX.
15. Add $P3_{LW}$ to SI-register.
16. Check for carry. If carry flag is set then go to next step, otherwise go to step 18.
17. Increment CX-register.
18. Add CX and $P3_{UW}$ to DI-register.
19. Clear CX-register.
20. Check for carry. If carry flag is set then go to next step, otherwise go to step 22.
21. Increment CX-register.
22. Save SI as second word of final product.
23. Get $D1_{UW}$ in AX-register.
24. Multiply AX and BP to get product 4 in AX and DX.
25. Add DI to AX to get third word of final product in AX.
26. Add CX to DX along with previous carry to get fourth word of final product in DX.
27. Save third (AX) and fourth (DX) word of final product in memory.
28. Stop.

Assembly language program

```
;PROGRAM TO MULTIPLY TWO 32-BIT DATA

CODE SEGMENT               ;Start of code segment.

        ORG  1000H         ;specify program starting address.
        ASSUME CS:CODE     ;Assembler directive.

        MOV  BX,1100H      ;Set BX as pointer for data.
        MOV  CX,0000H      ;Clear CX.

P1:     MOV  AX,[BX]       ;Get D1LW in AX-register.
        MOV  BP,[BX+04]    ;Get D2LW in BP-register.
        MUL  BP            ;Get P1 in AX and DX.
        MOV  [BX+08],AX    ;Save P1LW (first word of product) in memory.
        MOV  SI,DX         ;Save P1UW in SI-register.

P2:     MOV  AX,[BX+02]    ;Get D1UW in AX-register.
        MUL  BP            ;Get P2 in AX and DX.
        ADD  SI,AX         ;Add P1UW and P2LW.
        JNC  SKIP1
        INC  CX
SKIP1:  MOV  DI,DX         ;Save P2UW in DI-register.

P3:     MOV  AX,[BX]       ;Get D1LW in AX-register.
        MOV  BP,[BX+06]    ;Get D2UW in BP-register.
        MUL  BP            ;Get P3 in AX and DX.
        ADD  SI,AX       ;Get sum of P1UW, P2LW and P3LW in SI.
        JNC  SKIP2
        INC  CX
SKIP2:  ADD  DX,CX       ;Get sum of P2UW and P3UW in DI-register.
        MOV  CX,0000H
        ADC  DI,DX
        JNC  SKIP3
        INC  CX
SKIP3:  MOV  [BX+0AH],SI ;Save second word of the product in memory.

P4:     MOV  AX,[BX+02]  ;Get D1UW in AX-register.
        MUL  BP          ;Get P4 in AX and DX.
        ADD  AX,DI       ;Get sum of P2UW, P3UW and P4LW in AX.
        ADC  DX,CX
        MOV  [BX+0CH],AX ;Save third word of the product in memory.
        MOV  [BX+0EH],DX ;Save fourth word of the product in memory.
        HLT              ;Halt program execution.

CODE ENDS                ;End of code segment.
END                      ;Assembly end.
```

Assembler listing for example program 9

```
;PROGRAM TO MULTIPLY TWO 32-BIT DATA

0000              CODE SEGMENT              ;Start of code segment.

1000                      ORG  1000H        ;specify program starting address.
                          ASSUME CS:CODE    ;Assembler directive.

1000    BB 1100           MOV  BX,1100H     ;Set BX as pointer for data.
1003    B9 0000           MOV  CX,0000H     ;Clear CX.

1006    8B 07     P1:     MOV  AX,[BX]      ;Get D1LW in AX-register.
1008    8B 6F 04          MOV  BP,[BX+04]   ;Get D2LW in BP-register.
100B    F7 E5             MUL  BP           ;Get P1 in AX and DX.
100D    89 47 08          MOV  [BX+08],AX   ;Save P1LW (first word of product) in memory.
1010    8B F2             MOV  SI,DX        ;Save P1UW in SI-register.

1012    8B 47 02  P2:     MOV  AX,[BX+02]   ;Get D1UW in AX-register.
1015    F7 E5             MUL  BP           ;Get P2 in AX and DX.
1017    03 F0             ADD  SI,AX        ;Add P1UW and P2LW.
1019    73 01             JNC  SKIP1
101B    41                INC  CX
101C    8B FA     SKIP1:  MOV  DI,DX        ;Save P2UW in DI-register.

101E    8B 07     P3:     MOV  AX,[BX]      ;Get D1LW in AX-register.
1020    8B 6F 06          MOV  BP,[BX+06]   ;Get D2UW in BP-register.
1023    F7 E5             MUL  BP           ;Get P3 in AX and DX.
1025    03 F0             ADD  SI,AX        ;Get sum of P1UW, P2LW and P3LW in SI.
1027    73 01             JNC  SKIP2
1029    41                INC  CX
102A    03 D1     SKIP2:  ADD  DX,CX        ;Get sum of P2UW and P3UW in DI-register.
102C    B9 0000           MOV  CX,0000H
102F    13 FA             ADC  DI,DX
1031    73 01             JNC  SKIP3
1033    41                INC  CX
1034    89 77 0A  SKIP3:  MOV  [BX+0AH],SI  ;Save second word of the product in memory.

1037    8B 47 02  P4:     MOV  AX,[BX+02]   ;Get D1UW in AX-register.
103A    F7 E5             MUL  BP           ;Get P4 in AX and DX.
103C    03 C7             ADD  AX,DI        ;Get sum of P2UW, P3UW and P4LW in AX.

103E    13 D1             ADC  DX,CX
1040    89 47 0C          MOV  [BX+0CH],AX  ;Save third word of the product in memory.
```

```
1043    89 57 0E          MOV  [BX+0EH],DX ;Save fourth word of the product in memory.
1046    F4                HLT              ;Halt program execution.

1047           CODE ENDS                   ;End of code segment.
               END                         ;Assembly end.
```

Sample Data

```
Input Data : Data1 : 42107F6CH        Output Data : 0436 636C B64F 17D4H
             Data2 : 1052C26FH
```

Memory address	Content
1100	6C
1101	7F
1102	10
1103	42
1104	6F
1105	C2
1106	52
1107	10

Memory address	Content
1108	D4
1109	17
110A	4F
110B	B6
110C	6C
110D	63
110E	36
110F	04

EXAMPLE PROGRAM 10 : Division

Write an assembly language program to divide 32-bit data by 16-bit data.

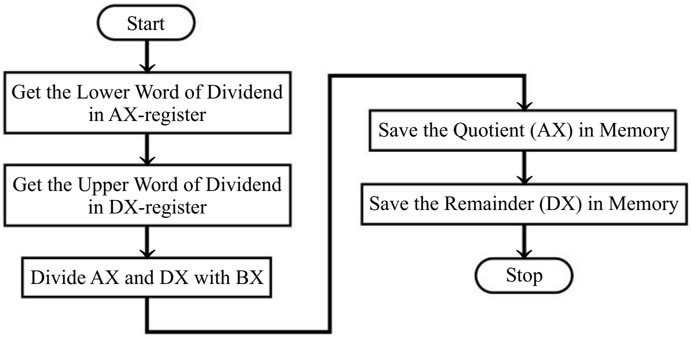
Problem Analysis

To perform division in the 8086 processor the 32-bit dividend should be stored in AX and DX registers (The lower word in AX and upper word in DX). The 16-bit divisor can be stored in the register/ memory. After division the quotient will be in AX-register and the remainder will be in DX-register.

Algorithm

1. Load the address of data in SI-register.
2. Get the lower word of dividend in AX-register.
3. Get the upper word of dividend in DX-register.
4. Get the divisor in BX-register.
5. Perform division to get quotient in AX and remainder in DX.
6. Save the quotient (AX) and the remainder (DX) in memory.
7. Stop.

Flowchart


Start
Get the Lower Word of Dividend in AX-register
Get the Upper Word of Dividend in DX-register
Divide AX and DX with BX
Save the Quotient (AX) in Memory
Save the Remainder (DX) in Memory
Stop

Assembly language program

```
;PROGRAM TO DIVIDE 32-BIT DATA BY 16-BIT DATA

CODE SEGMENT             ;Start of code segment.

          ASSUME CS:CODE ;Assembler directive.
          ORG 1000H      ;specify program starting address.

          MOV SI,1100H   ;Set SI as pointer for data.
          MOV AX,[SI]    ;Get the lower 16-bit of dividend in AX-register.
          MOV DX,[SI+2]  ;Get the upper 16-bit of dividend in DX-register.
          MOV BX,[SI+4]  ;Get the divisor in BX-register.
          DIV BX         ;Divide the content of AX and DX with content of BX.
                         ;The quotient will be in AX-register.
                         ;The remainder will be in DX-register.
          MOV [SI+6],AX  ;Save the quotient in memory.
          MOV [SI+8],DX  ;Save the remainder in memory.
          HLT            ;Halt program execution.

CODE ENDS                ;End of code segment.
END                      ;Assembly end.
```

Assembler listing for example program 10

```
;PROGRAM TO DIVIDE 32-BIT DATA BY 16-BIT DATA

0000                CODE SEGMENT             ;Start of code segment.

                             ASSUME CS:CODE ;Assembler directive.
1000                         ORG  1000H     ;specify program starting address.

1000   BE 1100               MOV  SI,1100H  ;Set SI as pointer for data.
1003   8B 04                 MOV  AX,[SI]   ;Get the lower 16-bit of dividend in AX-register.
1005   8B 54 02              MOV  DX,[SI+2] ;Get the upper 16-bit of dividend in DX-register.
1008   8B 5C 04              MOV  BX,[SI+4] ;Get the divisor in BX-register.
100B   F7 F3                 DIV  BX        ;Divide the content of AX and DX with content of BX.
                                            ;The quotient will be in AX-register.
                                            ;The rezmainder will be in DX-register.
100D   89 44 06              MOV  [SI+6],AX ;Save the quotient in memory.
1010   89 54 08              MOV  [SI+8],DX ;Save the remainder in memory.
1013   F4                    HLT            ;Halt program execution.

1014                CODE ENDS                ;End of code segment.
                    END                      ;Assembly end.
```

Sample Data

Input Data : Dividend = $71C2580A_H$ **Output Data :** Quotient = $75EE_H$

Divisor = $F6F2_H$ Remainder = $290E_H$

Memory address	Content
1100	0A
1101	58
1102	C2
1103	71
1104	F2
1105	F6

Memory address	Content
1106	EE
1107	75
1108	0E
1109	29

EXAMPLE PROGRAM 11 : Search for a Given Data

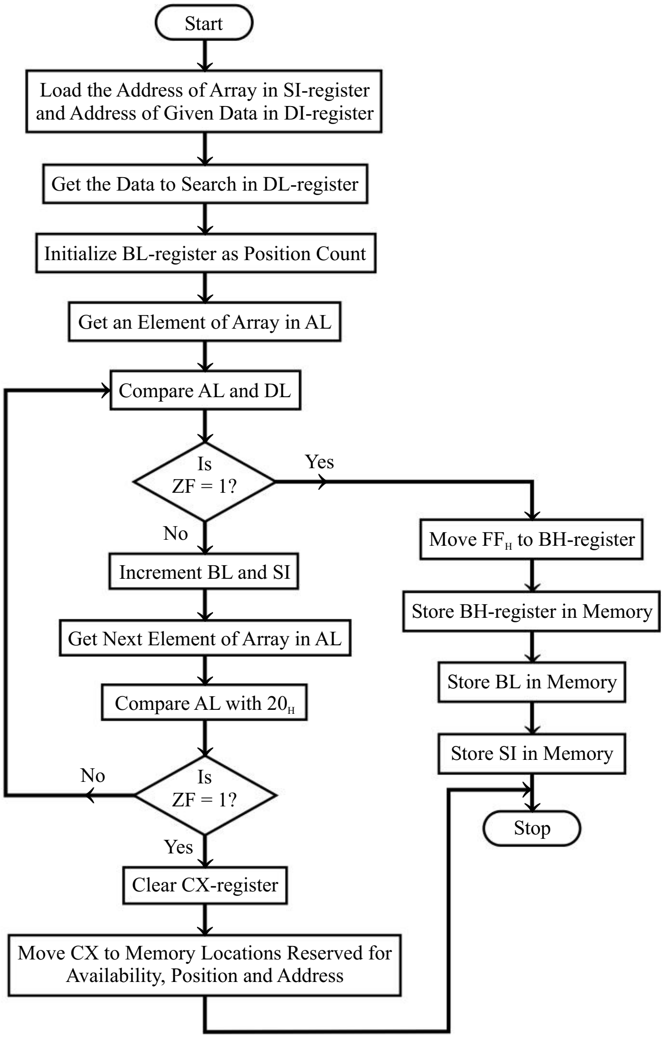
Write an assembly language program to search a given data in an array. Also determine the position and address of the data in the array.

Problem Analysis

The given data is stored in a register, and then it is compared with each element of the array. The comparison is terminated once the data is found or after comparing all elements of the array. One register can be used to keep track of the position of the element being compared. One of the index registers can be used to hold the address of the element being compared.

If the data is found then store FF_H in a memory location to show availability, and store the position and address in consecutive memory locations. If data is not available then store zero in all these locations. The array is terminated with character 20_H.

Flowchart


Start
Load the Address of Array in SI-register and Address of Given Data in DI-register
Get the Data to Search in DL-register
Initialize BL-register as Position Count
Get an Element of Array in AL
Compare AL and DL
Is ZF = 1?
Yes
No
Move FF_H to BH-register
Store BH-register in Memory
Store BL in Memory
Store SI in Memory
Stop
Increment BL and SI
Get Next Element of Array in AL
Compare AL with 20_H
Is ZF = 1?
No
Yes
Clear CX-register
Move CX to Memory Locations Reserved for Availability, Position and Address

Algorithm

1. Set SI-register as pointer for the array.
2. Set DI-register as pointer for given data and result.
3. Get the data to search in DL-register.
4. Let BL-register keep track of position. Initialize the position count as one.
5. Get an element of array in AL.
6. Compare an element of array (AL) with given data (DL).
7. Check for zero flag. If zero flag is set then go to step 14, otherwise go to next step.
8. Increment the array pointer (SI) and position count (BL).
9. Get next element of array in AL-register.
10. Compare AL with end marker (20_H).
11. Check zero flag. If zero flag is not set then go to step 6, otherwise go to next step.
12. Clear CX-register and store CX-register in four consecutive locations in memory after the given data.
13. Jump to end (step 17).
14. Move FF_H to BH-register and store it in memory.
15. Store the position count (BL) in memory.
16. Store the address (SI) in memory.
17. Stop.

Assembly language program

```
;PROGRAM TO SEARCH A GIVEN DATA IN AN ARRAY

CODE SEGMENT            ;Start of code segment.

        ASSUME CS:CODE  ;Assembler directive.
        ORG 1000H       ;specify program starting address.

START:  MOV SI,1100H    ;Set SI-register as pointer for array.
        MOV DI,1200H    ;Load the address of data to search in DI-register.
        MOV DL,[DI]     ;Get the data to search in DL-register.
        MOV BL,01H      ;Set BL-register as position count.
        MOV AL,[SI]     ;Get first element of array in AL-register.

AGAIN:  CMP AL,DL       ;Compare an element of array with data to search.
        JZ  AVAIL       ;If data are equal then jump to AVAIL.
        INC SI          ;If data are not equal, increment address pointer.
        INC BL          ;Increment position count.
        MOV AL,[SI]     ;Get the next element of array in AL-register.
        CMP AL,20H      ;Check for end of array,
        JNZ AGAIN       ;if not end, repeat search,otherwise go to NOTAVA.

NOTAVA: MOV CX,0000H    ;If search data is not found, then store zero.
        MOV [DI+1],CX
        MOV [DI+3],CX
        JMP OVER

AVAIL:  MOV BH,0FFH
        MOV [DI+1],BH   ;Store FFH to indicate availability of data.
        MOV [DI+2],BL   ;Store the position of data.
        MOV [DI+3],SI   ;Store the address of data.
OVER:   HLT             ;Halt program execution.

CODE ENDS               ;End of code segment.
END                     ;Assembly end.
```

Assembler listing for example program 11

```
;PROGRAM TO SEARCH A GIVEN DATA IN AN ARRAY

0000                  CODE SEGMENT          ;Start of code segment.

                             ASSUME CS:CODE ;Assembler directive.
1000                         ORG 1000H      ;specify program starting address.

1000   BE 1100      START:   MOV SI,1100H   ;Set SI-register as pointer for array.
1003   BF 1200               MOV DI,1200H   ;Load the address of data to search in DI-register.
1006   8A 15                 MOV DL,[DI]    ;Get the data to search in DL-register.
1008   B3 01                 MOV BL,01H     ;Set BL register as position count.
100A   8A 04                 MOV AL,[SI]    ;Get first element of array in AL-register.

100C   3A C2        AGAIN:   CMP AL,DL      ;Compare an element of array with data to search.
100E   74 15                 JZ  AVAIL      ;If data are equal then jump to AVAIL.
1010   46                    INC SI         ;If data are not equal,increment address pointer.
1011   FE C3                 INC BL         ;Increment position count.
1013   8A 04                 MOV AL,[SI]    ;Get the next element of array in AL-register.
1015   3C 20                 CMP AL,20H     ;Check for end of array,
1017   75 F3                 JNZ AGAIN      ;if not end, repeat search, otherwise go to NOTAVA.

1019   B9 0000      NOTAVA: MOV CX,0000H    ;If search data is not found, then store zero.
101C   89 4D 01              MOV [DI+1],CX
101F   89 4D 03              MOV [DI+3],CX
1022   EB 0C 90              JMP OVER

1025   B7 FF       AVAIL :   MOV BH,0FFH
1027   88 7D 01              MOV [DI+1],BH  ;Store FFH to indicate availability of data.
102A   88 5D 02              MOV [DI+2],BL  ;Store the position of data.
102D   89 75 03              MOV [DI+3],SI  ;Store the address of data.
1030   F4          OVER :    HLT            ;Halt program execution.

1031                       CODE ENDS        ;End of code segment.
                            END             ;Assembly end.
```

Sample Data

Input Data :
Array = $1F_H$
AC_H
$D0_H$
89_H
72_H
20_H
Given Data = 89_H

Memory address	Content
1100	1F
1101	AC
1102	D0
1103	89
1104	72
1105	20
1200	89

Output Data :
Availability = FF_H
Position = 04H
Address = 1103_H

Memory address	Content
1201	FF
1202	04
1203	03
1204	11

EXAMPLE PROGRAM 12 : Search for the Smallest Data

Write an assembly language program to search the smallest data in an array.

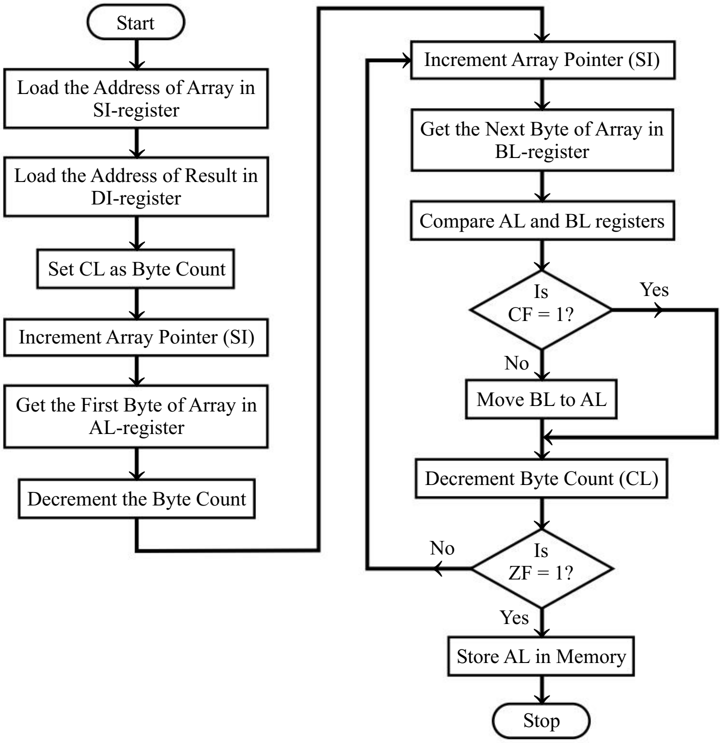
Problem Analysis

Let the size of the array be N bytes. Let us reserve AL-register to store the smallest data. The first byte of the array is assumed to be the smallest and it is saved in AL-register. Then each byte of the array is compared with AL. After each comparison the smaller among the two is brought to AL-register. Therefore, after the N–1 comparison the AL-register will have the smallest data.

Algorithm

1. Load the starting address of the array in SI-register.
2. Load the address of the result in DI-register.
3. Load the number of bytes in the array in CL-register.
4. Increment the array pointer (SI-register).
5. Get the first byte of the array in AL-register.
6. Decrement the byte count (CL-register).
7. Increment the array pointer (SI-register).
8. Get next byte of the array in BL-register.
9. Compare current smallest (AL) and next byte (BL) of the array.
10. Check carry flag. If carry flag is set then go to step 12, otherwise go to next step.
11. Move BL to AL.
12. Decrement the byte count (CL-register).
13. Check zero flag. If zero flag is reset then go to step 7, otherwise go to next step.
14. Save the smallest data in memory pointed by DI.
15. Stop.

Flowchart


Start
Load the Address of Array in SI-register
Load the Address of Result in DI-register
Set CL as Byte Count
Increment Array Pointer (SI)
Get the First Byte of Array in AL-register
Decrement the Byte Count
Increment Array Pointer (SI)
Get the Next Byte of Array in BL-register
Compare AL and BL registers
Is CF = 1?
Yes
No
Move BL to AL
Decrement Byte Count (CL)
No
Is ZF = 1?
Yes
Store AL in Memory
Stop

Assembly language program

```
;PROGRAM TO FIND SMALLEST DATA IN AN ARRAY

CODE SEGMENT           ;Start of code segment.

       ASSUME CS:CODE ;Assembler directive.
       ORG  1000H     ;specify program starting address.
```

```
START:  MOV  SI,1100H  ;Set SI-register as pointer for array.
        MOV  DI,1200H  ;Set DI-register as pointer for result.
        MOV  CL,[SI]   ;Set CL as count for elements in the array.
        INC  SI        ;Increment the address pointer.
        MOV  AL,[SI]   ;Set first data as smallest.
        DEC  CL        ;Decrement the count.

AGAIN:  INC  SI        ;Make SI to point to next data in array.
        MOV  BL,[SI]   ;Get the next data in BL-register.
        CMP  AL,BL     ;Compare current smallest data in AL with BL.
        JC   AHEAD     ;If carry is set then  AL is less than BL,
                       ;hence proceed to AHEAD.
        MOV  AL,BL     ;If carry is not set,
                       ;then make BL as current smallest.

AHEAD:  DEC  CL        ;Decrement the count.
        JNZ  AGAIN     ;If count is not zero repeat search.
        MOV  [DI],AL   ;Store the smallest data in memory.
        HLT            ;Halt program execution.

CODE ENDS              ;End of code segment.
END                    ;Assembly end.
```

Assembler listing for example program 12

```
;PROGRAM TO FIND SMALLEST DATA IN AN ARRAY

0000              CODE SEGMENT          ;Start of code segment.

                         ASSUME CS:CODE ;Assembler directive.
1000                     ORG 1000H      ;specify program starting address.

1000   BE 1100    START: MOV SI,1100H   ;Set SI-register as pointer for array.
1003   BF 1200           MOV DI,1200H   ;Set DI-register as pointer for result.
1006   8A 0C             MOV CL,[SI]    ;Set CL as count for elements in the array.
1008   46                INC SI         ;Increment the address pointer.
1009   8A 04             MOV AL,[SI]    ;Set first data as smallest.
100B   FE C9             DEC CL         ;Decrement the count.

100D   46         AGAIN: INC SI         ;Make SI to point to next data in array.
100E   8A 1C             MOV BL,[SI]    ;Get the next data in BL-register.
1010   3A C3             CMP AL,BL      ;Compare current smallest data in AL with BL.
1012   72 02             JC  AHEAD      ;If carry is set then AL is less than BL,
                                        ;hence proceed to AHEAD.
1014   8A C3             MOV AL,BL      ;If carry is not set,
                                        ;then make BL as current smallest.

1016   FE C9     AHEAD:  DEC CL         ;Decrement the count.
1018   75 F3             JNZ AGAIN      ;If count is not zero repeat search.
101A   88 05             MOV [DI],AL    ;Store the smallest data in memory.
101C   F4                HLT            ;Halt program execution.

101D              CODE   ENDS           ;End of code segment.
                  END                   ;Assembly end.
```

Sample Data

Input Data : 06_H(count) **Output Data** : $2D_H$
$4E_H$
$2D_H$
30_H
98_H
AC_H
FE_H

Memory address	Content
1100	06
1101	4E
1102	2D
1103	30
1104	98
1105	AC
1106	FE
1200	2D

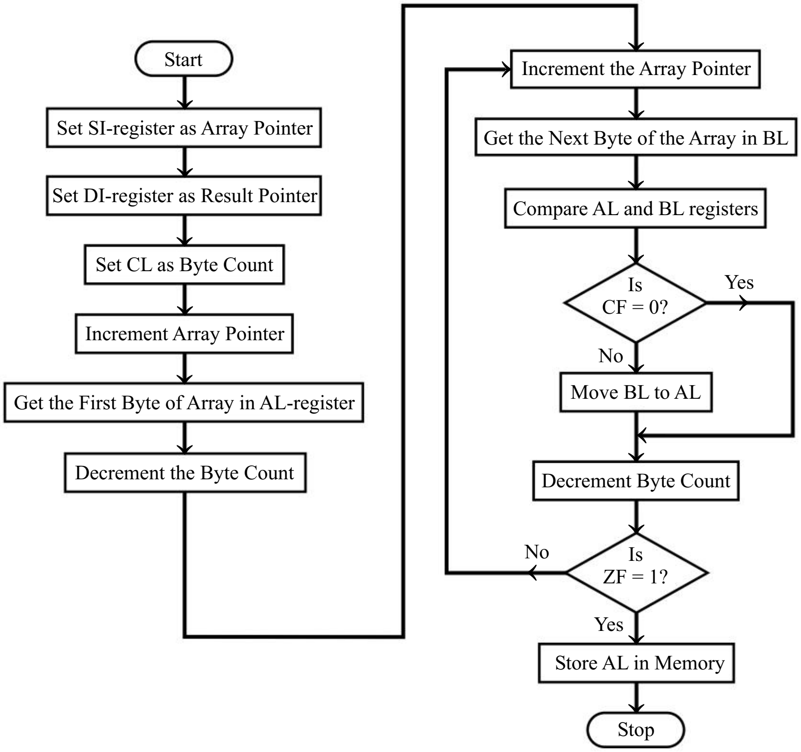
EXAMPLE PROGRAM 13 : Search for Largest Data

Write an assembly language program to search the largest data in an array.

Problem Analysis

Let the size of array be N bytes. Let us reserve AL-register to store the largest data. The first byte of the array is assumed to be the largest and it is saved in AL-register. Then each byte of the array is compared with AL. After each comparison the larger among the two is brought to AL-register. Therefore, after N–1 comparisons the AL-register will have the largest data.

Flowchart


Start
Set SI-register as Array Pointer
Set DI-register as Result Pointer
Set CL as Byte Count
Increment Array Pointer
Get the First Byte of Array in AL-register
Decrement the Byte Count
Increment the Array Pointer
Get the Next Byte of the Array in BL
Compare AL and BL registers
Is CF = 0?
Yes
No
Move BL to AL
Decrement Byte Count
No
Is ZF = 1?
Yes
Store AL in Memory
Stop

Algorithm

1. Load the starting address of the array in SI-register.
2. Load the address of result in DI-register.
3. Load the number of bytes in the array in CL-register.
4. Increment the array pointer (SI-register).
5. Get the first byte of the array in AL-register.
6. Decrement the byte count (CL-register).
7. Increment the array pointer (SI-register).
8. Get next byte of the array in BL-register.
9. Compare current largest (AL) and next byte (BL) of the array.
10. Check carry flag. If carry flag is reset then go to step 12, otherwise go to next step.

11. Move BL to AL.
12. Decrement byte count (CL-register).
13. Check zero flag. If zero flag is reset then go to step 7, otherwise go to next step.
14. Store the largest data in memory pointed by DI.
15. Stop.

Assembly language program

```
;PROGRAM TO FIND THE LARGEST DATA IN AN ARRAY

CODE SEGMENT          ;Start of code segment.

       ASSUME CS:CODE ;Assembler directive.
       ORG 1000H      ;specify program starting address.

START: MOV SI,1100H   ;Set SI-register as pointer for array.
       MOV DI,1200H   ;Set DI-register as pointer for result.

       MOV CL,[SI]    ;Set CL as count for elements in the array.
       INC SI         ;Increment the address pointer.
       MOV AL,[SI]    ;Set first data as largest.
       DEC CL         ;Decrement the count.

AGAIN: INC SI         ;Make SI to point to next data in array.
       MOV BL,[SI]    ;Get the next data in BL-register.
       CMP AL,BL      ;Compare the current largest data in AL with BL.
       JNC AHEAD      ;If carry is not set then AL is greater than BL,
                      ;hence proceed to AHEAD.
       MOV AL,BL      ;If carry is set then make BL as current largest.

AHEAD: DEC CL         ;Decrement the count.
       JNZ AGAIN      ;If count is not zero repeat search.
       MOV [DI],AL    ;Store the largest data in memory.
       HLT            ;Halt program execution.

CODE ENDS             ;End of code segment.
END                   ;Assembly end.
```

Assembler listing for example program 13

```
;PROGRAM TO FIND THE LARGEST DATA IN AN ARRAY

0000            CODE SEGMENT           ;Start of code segment.

                       ASSUME CS:CODE  ;Assembler directive.
1000                   ORG  1000H      ;specify program starting address.

1000    BE 1100 START: MOV  SI,1100H   ;Set SI-register as pointer for array.
1003    BF 1200        MOV  DI,1200H   ;Set DI-register as pointer for result.

1006    8A 0C          MOV  CL,[SI]    ;Set CL as count for elements in the array.
1008    46             INC  SI         ;Increment the address pointer.
1009    8A 04          MOV  AL,[SI]    ;Set first data as largest.
100B    FE C9          DEC  CL         ;Decrement the count.

100D    46      AGAIN: INC  SI         ;Make SI to point to next data in array.
100E    8A 1C          MOV  BL,[SI]    ;Get the next data in BL-register.
1010    3A C3          CMP  AL,BL      ;Compare the current largest data in AL with BL.
1012    73 02          JNC  AHEAD      ;If carry is not set then AL is greater than BL,
                                       ;hence proceed to AHEAD.
1014    8A C3          MOV  AL,BL      ;If carry is set then make BL as current largest.
```

```
1016    FE C9     AHEAD: DEC  CL          ;Decrement the count.
1018    75 F3            JNZ  AGAIN       ;If count is not zero repeat search.
101A    88 05            MOV  [DI],AL     ;Store the largest data in memory.
101C    F4               HLT              ;Halt program execution.

101D              CODE ENDS               ;End of code segment.
                  END                     ;Assembly end.
```

Sample Data

Input Data : 06 (count) Output Data : FE_H

$4E_H$

$2D_H$

30_H

98_H

AC_H

FE_H

Memory address	Content
1100	06
1101	4E
1102	2D
1103	30
1104	98
1105	AC
1106	FE
1200	FE

EXAMPLE PROGRAM 14 : Sorting an Array in Ascending Order

Write an assembly language program to sort an array of data in ascending order.

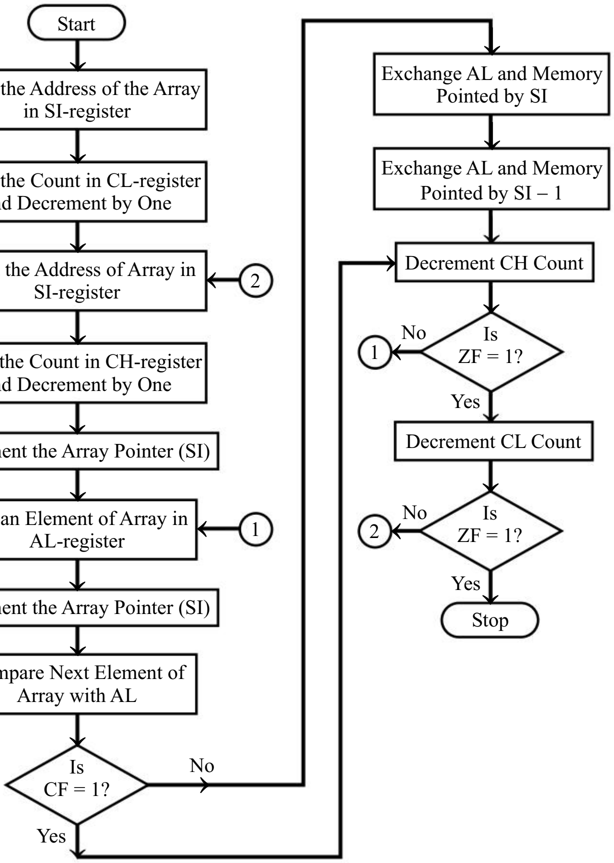
Problem Analysis

The array can be sorted in ascending order by bubble sorting. In bubble sorting of N-data, N–1 comparisons are performed by taking two consecutive data at a time. After each comparison the two data can be rearranged in the ascending order in the same memory locations, i.e., smaller first and larger next. When the above N–1 comparisons are performed N–1 times, the array will be sorted in ascending order in the same locations.

Algorithm

1. Set SI-register as pointer for array.
2. Set CL-register as count for N–1 repetitions.
3. Initialize array pointer.
4. Set CH as count for N–1 comparisons.
5. Increment the array pointer.
6. Get an element of array in AL-register.
7. Increment the array pointer.
8. Compare the next element of the array with AL.
9. Check carry flag. If carry flag is set then go to step 12, otherwise go to next step.
10. Exchange the content of memory pointed by SI and the content of previous memory location. (For this, exchange AL and memory pointed by SI, and then exchange AL and memory pointed by SI–1.)
11. Decrement the count for comparisons (CH-register).
12. Check zero flag. If zero flag is reset then go to step 6, otherwise go to next step.
13. Decrement the count for repetitions (CL-register).
14. Check zero flag. If zero flag is reset then go to step 3, otherwise go to next step.
15. Stop.

Flowchart


Start
Load the Address of the Array in SI-register
Load the Count in CL-register and Decrement by One
Load the Address of Array in SI-register
2
Load the Count in CH-register and Decrement by One
Increment the Array Pointer (SI)
Get an Element of Array in AL-register
1
Increment the Array Pointer (SI)
Compare Next Element of Array with AL
Is CF = 1?
No
Yes
Exchange AL and Memory Pointed by SI
Exchange AL and Memory Pointed by SI – 1
Decrement CH Count
No
1
Is ZF = 1?
Yes
Decrement CL Count
No
2
Is ZF = 1?
Yes
Stop

Assembly language program

```
;PROGRAM TO SORT AN ARRAY OF DATA IN ASCENDING ORDER

CODE SEGMENT              ;Start of code segment.

        ASSUME CS:CODE    ;Assembler directive.
        ORG    1000H      ;specify program starting address.

START:  MOV    SI,1100H   ;Set SI-register as pointer for array.
        MOV    CL,[SI]    ;Set CL as count for N-1 repetitions.
        DEC    CL

REPEAT: MOV    SI,1100H   ;Initialize pointer.
        MOV    CH,[SI]    ;Set CH as count for N-1 comparisons.
        DEC    CH
        INC    SI         ;Increment the pointer.

REPCOM: MOV    AL,[SI]    ;Get an element of array in AL-register.
        INC    SI
        CMP    AL,[SI]    ;Compare with next element of array in memory.
        JC     AHEAD      ;If AL is lesser than memory, then go to AHEAD.
        XCHG AL,[SI]   ;If AL is less than memory then,
        XCHG AL,[SI-1] ;exchange the content of memory pointed by SI,
                       ;and the previous memory location.

AHEAD:  DEC  CH        ;Decrement the count for comparisons.
        JNZ  REPCOM    ;Repeat comparison until CH count is zero.
        DEC  CL        ;Decrement the count for repetitions.
        JNZ  REPEAT    ;Repeat N-1 comparisons until CL count is zero.
        HLT            ;Halt program execution.

CODE ENDS              ;End of code segment.
END                    ;Assembly end.
```

Assembler listing for example program 14

```
;PROGRAM TO SORT AN ARRAY OF DATA IN ASCENDING ORDER

0000                CODE SEGMENT              ;Start of code segment.

                            ASSUME CS: CODE   ;Assembler directive.
1000                        ORG  1000H        ;specify program starting address.

1000    BE 1100     START:  MOV  SI,1100H     ;Set SI-register as pointer for array.
1003    8A 0C               MOV  CL,[SI]      ;Set CL as count for N-1 repetitions.
1005    FE C9               DEC  CL

1007    BE 1100     REPEAT: MOV  SI,1100H     ;Initialize pointer.
100A    8A 2C               MOV  CH,[SI]      ;Set CH as count for N-1 comparisons.
100C    FE CD               DEC  CH
100E    46                  INC  SI           ;Increment the pointer.

100F    8A 04       REPCOM: MOV  AL,[SI]      ;Get an element of array in AL-register.
1011    46                  INC  SI
1012    3A 04               CMP  AL,[SI]      ;Compare with next element of array in memory.
1014    72 05               JC   AHEAD        ;If AL is lesser than memory, then go to AHEAD.

1016    86 04               XCHG AL,[SI]      ;If AL is less than memory then,
1018    86 44 FF            XCHG AL,[SI-1]    ;exchange the content of memory pointed by SI,
                                              ;and the previous memory location.

101B    FE CD       AHEAD:  DEC  CH           ;Decrement the count for comparisons.
101D    75 F0               JNZ  REPCOM       ;Repeat comparison until CH count is zero.
101F    FE C9               DEC  CL           ;Decrement the count for repetitions.
1021    75 E4               JNZ  REPEAT       ;Repeat N-1 comparisons until CL count is zero.
1023    F4                  HLT               ;Halt program execution.

1024                CODE ENDS                 ;End of code segment.
                    END                       ;Assembly end.
```

Sample Data

Before Execution

Input Data : 07
AA
77
FF
22
11
44
BB

Memory address	Content
1100	07
1101	AA
1102	77
1103	FF
1104	22
1105	11
1106	44
1107	BB

After Execution

Output Data : 11
22
44
77
AA
BB
FF

Memory address	Content
1101	11
1102	22
1103	44
1104	77
1105	AA
1106	BB
1107	FF

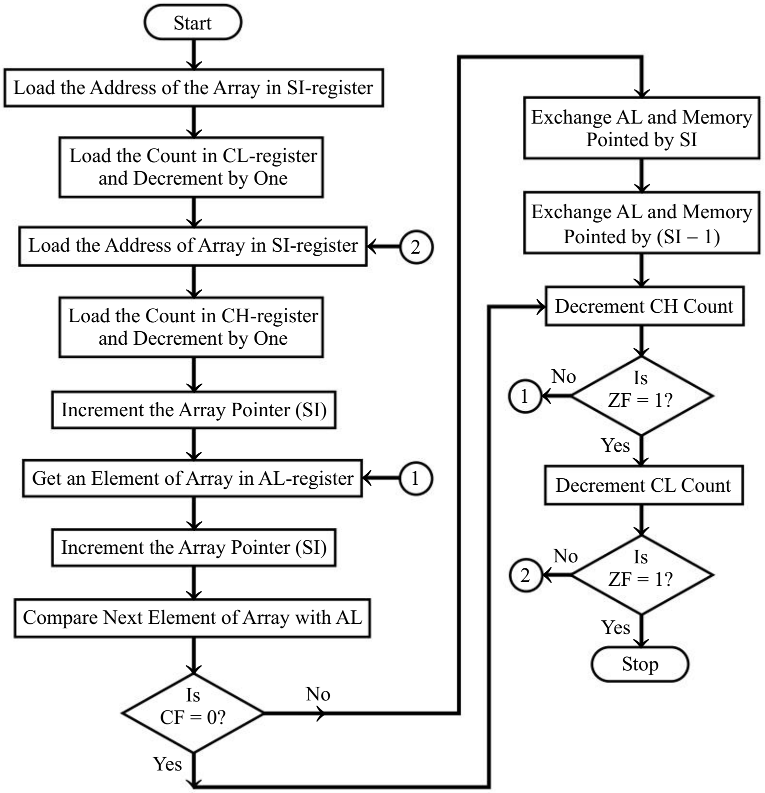
EXAMPLE PROGRAM 15 : Sorting an Array in Descending Order

Write an assembly language program to sort an array of data in descending order.

Problem Analysis

The array can be sorted in descending order by bubble sorting. In bubble sorting of N-data, N–1 comparisons are performed by taking two consecutive data at a time. After each comparison, the two data can be rearranged in the descending order in the same memory locations, i.e., larger first and smaller next. When the above N–1 comparisons are performed N–1 times, the array will be sorted in descending order in the same locations.

Flowchart


Start
Load the Address of the Array in SI-register
Load the Count in CL-register and Decrement by One
Load the Address of Array in SI-register
2
Load the Count in CH-register and Decrement by One
Increment the Array Pointer (SI)
Get an Element of Array in AL-register
1
Increment the Array Pointer (SI)
Compare Next Element of Array with AL
Is CF = 0?
No
Yes
Exchange AL and Memory Pointed by SI
Exchange AL and Memory Pointed by (SI – 1)
Decrement CH Count
Is ZF = 1?
No
1
Yes
Decrement CL Count
Is ZF = 1?
No
2
Yes
Stop

Algorithm

1. Set SI-register as pointer for array.
2. Set CL-register as count for N–1 repetitions.
3. Initialize array pointer.
4. Set CH as count for N–1 comparisons.
5. Increment the array pointer.
6. Get an element of array in AL-register.
7. Increment the array pointer.
8. Compare the next element of the array with AL.

9. Check carry flag. If carry flag is reset then go to step 12, otherwise go to next step.
10. Exchange the content of memory pointed by SI and the content of previous memory location. (For this, exchange AL and memory pointed by SI, and then exchange AL and memory pointed by SI – 1.)
11. Decrement the count for comparisons (CH-register).
12. Check zero flag. If zero flag is reset then go to step 6, otherwise go to next step.
13. Decrement the count for repetitions (CL-register).
14. Check zero flag. If zero flag is reset then go to step 3, otherwise go to next step.
15. Stop.

Assembly language program

```
;PROGRAM TO SORT AN ARRAY OF DATA IN DESCENDING ORDER

CODE SEGMENT             ;Start of code segment.

        ASSUME CS:CODE   ;Assembler directive.
        ORG   1000H      ;specify program starting address.

START:  MOV   SI,1100H   ;Set SI-register as pointer for array.
        MOV   CL,[SI]    ;Set CL as count for N-1 repetitions.
        DEC   CL

REPEAT: MOV   SI,1100H   ;Initialize pointer.
        MOV   CH,[SI]    ;Set CH as count for N-1 comparisons.
        DEC   CH
        INC   SI         ;Increment the pointer.

REPCOM: MOV   AL,[SI]    ;Get an element of array in AL-register.
        INC   SI
        CMP   AL,[SI]    ;Compare with next element of the array in memory.
        JNC   AHEAD      ;If AL is greater than memory, then go to AHEAD.

        XCHG  AL,[SI]    ;If AL is less than memory then,
        XCHG  AL,[SI-1]  ;exchange the content of memory pointed by SI,
                         ;and the previous memory location.
AHEAD:  DEC   CH         ;Decrement the count for comparisons.
        JNZ   REPCOM     ;Repeat comparison until CH count is zero.
        DEC   CL         ;Decrement the count for repetitions.
        JNZ   REPEAT     ;Repeat N-1 comparisons until CL count is zero.
        HLT              ;Halt program execution.

CODE ENDS                ;End of code segment.
END                      ;Assembly end.
```

Assembler listing for example program 15

```
;PROGRAM TO SORT AN ARRAY OF DATA IN DESCENDING ORDER

0000            CODE SEGMENT             ;Start of code segment.

                        ASSUME CS:CODE   ;Assembler directive.
1000                    ORG   1000H      ;specify program starting address.

1000  BE 1100   START:  MOV   SI,1100H   ;Set SI-register as pointer for array.
1003  8A 0C             MOV   CL,[SI]    ;Set CL as count for N-1 repetitions.
1005  FE C9             DEC   CL

1007  BE 1100   REPEAT: MOV   SI,1100H   ;Initialize pointer.
100A  8A 2C             MOV   CH,[SI]    ;Set CH as count for N-1 comparisons.
100C  FE CD             DEC   CH
100E  46                INC   SI         ;Increment the pointer.

100F  8A 04     REPCOM: MOV   AL,[SI]    ;Get an element of array in AL-register.
```

```
1011   46                 INC   SI
1012   3A 04              CMP   AL,[SI]    ;Compare with next element of the array in memory.
1014   73 05              JNC   AHEAD      ;If AL is greater than memory, then go to AHEAD.

1016   86 04              XCHG AL,[SI]     ;If AL is less than memory then,
1018   86 44 FF           XCHG AL,[SI-1]   ;exchange the content of memory pointed by SI,
                                           ;and the previous memory location.

101B   FE CD     AHEAD:   DEC   CH         ;Decrement the count for comparisons.
101D   75 F0              JNZ   REPCOM     ;Repeat comparison until CH count is zero.
101F   FE C9              DEC   CL         ;Decrement the count for repetitions.
1021   75 E4              JNZ   REPEAT     ;Repeat N-1 comparisons until CL count is zero.
1023   F4                 HLT              ;Halt program execution.

1024             CODE ENDS                 ;End of code segment.
                 END                       ;Assembly end.
```

Sample Data

Before Execution

Input Data : 07
AA
77
FF
22
11
44
BB

Memory address	Content
1100	07
1101	AA
1102	77
1103	FF
1104	22
1105	11
1106	44
1107	BB

After Execution

Output Data : FF
BB
AA
77
44
22
11

Memory address	Content
1101	FF
1102	BB
1103	AA
1104	77
1105	44
1106	22
1107	11

EXAMPLE PROGRAM 16 : GCD of Two 16-Bit Data

Write an assembly language program to determine the GCD of two 16-bit data.

Problem Analysis

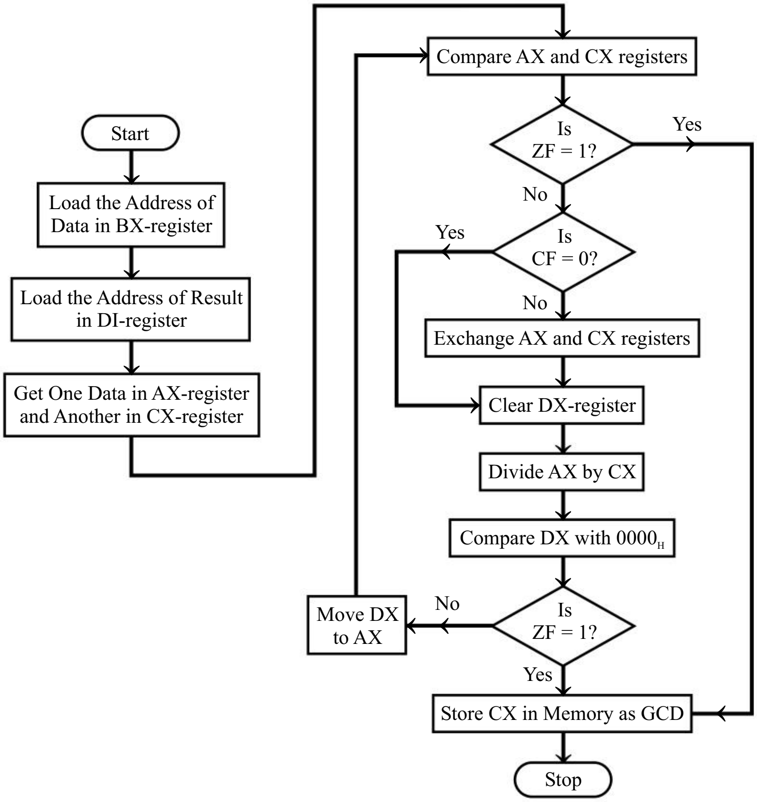
First divide the larger data by the smaller data and check for remainder. If the remainder is zero, then smaller data is the GCD.

If the remainder is not zero then take the remainder as the divisor and the previous divisor as the dividend and repeat division until the remainder is zero. When the remainder is zero, we can store the divisor as GCD. Before performing division we can even check whether the dividend and divisor are equal. If they are equal, then we can directly store the divisor as GCD without performing division.

Algorithm

1. Set BX as pointer for input data.
2. Set DI as pointer for result.
3. Get one data in AX-register.
4. Get another data in CX-register.
5. Compare the two data (AX and CX).
6. Check zero flag. If zero flag is set then go to step 14, otherwise go to next step.
7. Check carry flag. If carry flag is reset then go to step 9, otherwise go to next step.
8. Exchange the content of AX and CX, so that the larger among the two is dividend and smaller is the divisor.
9. Clear DX-register.
10. Divide AX-register by CX-register.
11. Compare DX-register (Remainder) with 0000_H.
12. Check zero flag. If zero flag is set then go to step 14, otherwise go to next step.
13. Move the remainder (DX-register) to AX and go to step 5.
14. Save the content of CX-register as GCD in memory.
15. Stop.

Flowchart


Start
Load the Address of Data in BX-register
Load the Address of Result in DI-register
Get One Data in AX-register and Another in CX-register
Compare AX and CX registers
Is ZF = 1?
Yes
No
Is CF = 0?
Yes
No
Exchange AX and CX registers
Clear DX-register
Divide AX by CX
Compare DX with 0000_H
Is ZF = 1?
No
Move DX to AX
Yes
Store CX in Memory as GCD
Stop

Assembly language program

```
;PROGRAM TO FIND GCD OF TWO 16-BIT DATA

CODE SEGMENT             ;start of code segment.

       ASSUME CS:CODE    ;Assembler directive.
       ORG   1000H       ;specify program starting address.

       MOV   BX,1100H    ;Set BX-register as pointer for data.
       MOV   DI,1200H    ;Set DI-register as pointer for result.
       MOV   AX,[BX]     ;Get the first data in AX-register.
       MOV   CX,[BX+02]  ;Get the second data in CX-register.

RPT:   CMP   AX,CX       ;Compare the two data.
       JE    STORE       ;If the data are equal, store CX as GCD.
       JNC   SKIP        ;If AX is greater than CX, then go to SKIP.
       XCHG  AX,CX       ;If AX is less than CX, then exchange AX and CX.

SKIP:  MOV   DX,0000H
       DIV   CX          ;Divide the two data.
       CMP   DX,0000H    ;Check whether remainder is zero.
       JE    STORE       ;If remainder is zero, then store CX as GCD.
       MOV   AX,DX       ;If remainder is not zero, move remainder to AX.
       JMP   RPT         ;Repeat comparison and division.

STORE: MOV   [DI],CX     ;Store CX as GCD.
       HLT               ;Halt program execution.
```

```
CODE ENDS                ;End of code segment.
END                      ;Assembly end.
```

Assembler listing for example program 16

```
;PROGRAM TO FIND GCD OF TWO 16-BIT DATA

0000           CODE SEGMENT           ;Start of code segment.

                      ASSUME CS:CODE  ;Assembler directive.
1000                  ORG   1000H     ;specify program starting address.

1000  BB 1100         MOV   BX,1100H  ;Set BX-register as pointer for data.
1003  BF 1200         MOV   DI,1200H  ;Set DI-register as pointer for result.
1006  8B 07           MOV   AX,[BX]   ;Get the first data in AX-register.
1008  8B 4F 02        MOV   CX,[BX+02] ;Get the second data in CX-register.

100B  3B C1    RPT:   CMP   AX,CX     ;Compare the two data.
100D  74 11           JE    STORE     ;If the data are equal, store CX as GCD.
100F  73 01           JNC   SKIP      ;If AX is greater than CX, then go to SKIP.
1011  91              XCHG  AX,CX     ;If AX is less than CX, then exchange AX and CX.

1012  BA 0000  SKIP:  MOV   DX,0000H
1015  F7 F1           DIV   CX        ;Divide the two data.
1017  83 FA 00        CMP   DX,0000H  ;Check whether remainder is zero.
101A  74 04           JE    STORE     ;If remainder is zero, then store CX as GCD.
101C  8B C2           MOV   AX,DX     ;If remainder is not zero, move remainder to AX.
101E  EB EB           JMP   RPT       ;Repeat comparison and division.

1020  89 0D    STORE: MOV   [DI],CX   ;Store CX as GCD.
1022  F4              HLT             ;Halt program execution.

1023           CODE ENDS              ;End of code segment.
               END                    ;Assembly end.
```

Sample Data

Input Data : Data1 = $358E_H$ Output Data : 0005_H

Data2 = $01BD_H$

Memory address	Content
1100	8E
1101	35
1102	BD
1103	01
1200	05
1201	00

EXAMPLE PROGRAM 17 : LCM of Two 16-Bit Data

Write an assembly language program to determine the LCM of two 16-bit data.

Problem Analysis

First determine the GCD of two data. Then determine the product of two data. Here it is assumed that the product does not exceed 32 bits. When the product is divided by GCD, the quotient will be the LCM of the two data. (For the GCD of two data please refer to example program 16.)

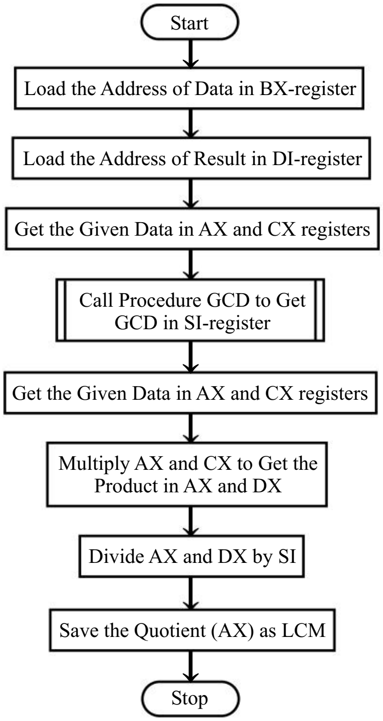
Algorithm

1. Set BX as pointer for input data.
2. Set DI as pointer for result.
3. Get first data in AX-register and second data in CX-register.
4. Call procedure GCD to get the GCD in SI-register.
5. Again get first data in AX-register and second data in CX-register.

6. Determine the product of two data. The product will be in AX and DX registers.
7. Divide the product (AX and DX registers) by GCD (SI-register).
8. Save the quotient (AX-register) as LCM in memory.
9. Stop.

Note : The algorithm for procedure GCD can be obtained from example progra m 16.

Flowchart


Start
Load the Address of Data in BX-register
Load the Address of Result in DI-register
Get the Given Data in AX and CX registers
Call Procedure GCD to Get GCD in SI-register
Get the Given Data in AX and CX registers
Multiply AX and CX to Get the Product in AX and DX
Divide AX and DX by SI
Save the Quotient (AX) as LCM
Stop

Note : The flowchart for procedure GCD can be obtained from example progra m 16.

Assembly language program

```
;PROGRAM TO FIND LCM OF TWO 16-BIT DATA

CODE SEGMENT           ;Start of code segment.

      ASSUME CS:CODE ;Assembler directive.
      ORG  1000H     ;specify program starting address.

      MOV  BX,1100H  ;Set BX-register as pointer for data.
      MOV  DI,1200H  ;Set DI-register as pointer for result .
      MOV  AX,[BX]   ;Get the first data in AX-register.
      MOV  CX,[BX+02] ;Get the second data in CX-register.
      CALL GCD       ;Call procedure GCD.

      MOV  AX,[BX]   ;Get the first data in AX-register.
      MOV  CX,[BX+02] ;Get the second data in CX-register.
      MUL  CX        ;Get product of two numbers in AX and DX.
      DIV  SI        ;Divide the product with GCD.
      MOV  [DI],AX   ;Save the quotient as LCM.
      HLT            ;Halt program execution.
```

```
GCD PROC NEAR

RPT:   CMP  AX,CX       ;Compare the two data.
       JE   SAVE        ;If the data are equal, store CX as GCD.
       JNC  SKIP        ;If AX is greater than CX, then go to SKIP.
       XCHG AX,CX       ;If AX is less than CX, then exchange AX and CX.
SKIP:  MOV  DX,0000H
       DIV  CX          ;Divide the two data.
       CMP  DX,0000H    ;Check whether remainder is zero.
       JE   SAVE        ;If remainder is zero, then store CX as GCD.
       MOV  AX,DX       ;If remainder is not zero, move remainder to AX.
       JMP  RPT         ;Repeat comparison and division.

SAVE:  MOV  SI,CX       ;Store CX as GCD.
       RET

GCD    ENDP             ;Assembler directive.

CODE   ENDS             ;End of code segment.
END                     ;Assembly end.
```

Assembler listing for example program 17

```
;PROGRAM TO FIND LCM OF TWO 16-BIT DATA

0000            CODE SEGMENT            ;Start of code segment.

                       ASSUME CS:CODE   ;Assembler directive.
1000                   ORG  1000H       ;specify program starting address.

1000   BB 1100 R       MOV  BX,1100H    ;Set BX-register as pointer for data.
1003   BF 1200 R       MOV  DI,1200H    ;Set DI-register as pointer for result.
1006   8B 07           MOV  AX,[BX]     ;Get the first data in AX-register.
1008   8B 4F 02        MOV  CX,[BX+02]  ;Get the second data in CX-register.
100B   E8 000C R       CALL GCD         ;Call procedure GCD.

100E   8B 07           MOV  AX,[BX]     ;Get the first data in AX-register.
1010   8B 4F 02        MOV  CX,[BX+02]  ;Get the second data in CX-register.
1013   F7 E1           MUL  CX          ;Get product of two numbers in AX and DX.
1015   F7 F6           DIV  SI          ;Divide the product with GCD.
1017   89 05           MOV  [DI],AX     ;Save the quotient as LCM.
1019   F4              HLT              ;Halt program execution.

101A            GCD PROC NEAR

101A   3B C1    RPT:   CMP  AX,CX       ;Compare the two data.
101C   74 11           JE   SAVE        ;If the data are equal, store CX as GCD.
101E   73 01           JNC  SKIP        ;If AX is greater than CX, then go to SKIP.
1020   91              XCHG AX,CX       ;If AX is less than CX, then exchange AX and CX.
1021   BA 0000  SKIP:  MOV  DX,0000H
1024   F7 F1           DIV  CX          ;Divide the two data.
1026   83 FA 00        CMP  DX,0000H    ;Check whether remainder is zero.
1029   74 04           JE   SAVE        ;If remainder is zero, then store CX as GCD.
102B   8B C2           MOV  AX,DX       ;If remainder is not zero, move remainder to AX.
102D   EB EB           JMP  RPT         ;Repeat comparison and division.

102F   8B F1     SAVE: MOV  SI,CX       ;Store CX as GCD.
1031   C3              RET

1032            GCD  ENDP               ;Assembler directive.

1032            CODE ENDS               ;End of code segment.
                END                     ;Assembly end.
```

Sample Data

Input Data :
Data1 = 0042_H
Data2 = $003A_H$

Output Data : $077A_H$

Memory address	Content
1100	42
1101	00
1102	3A
1103	00
1200	7A
1201	07

EXAMPLE PROGRAM 18 : Factorial of 8-Bit Data

Write an assembly language program to determine the factorial of 8-bit data.

Problem Analysis

The factorial can be calculated by repeated multiplication. In the first multiplication the given data is taken as multiplicand and data–1 (data minus one) is taken as multiplier. In each subsequent multiplication the previous product is taken as multiplicand and previous multiplier is decremented by one and used as current multiplier. The multiplications are repeated until the multiplier becomes zero. The final product after data–1 (data minus one) multiplications will be the factorial of the data.

In this example it is assumed that the product/factorial does not exceed 32 bits. The given data is converted to 16-bit data by taking the high byte as zero and in each multiplication 32-bit by 16-bit multiplication is performed. The logic of 32-bit by 16-bit multiplication is given below :

The multiplicand can be divided into two words : **L**ower **W**ord (LW) and **U**pper **W**ord (UW) as shown below.

Multiplicand (32-bit) → MD_{UW} (16-bit), MD_{LW} (16-bit)

Let the 16-bit multiplier be MR. Then perform the following two multiplications. Each multiplication will give a 32-bit result which can be divided into **L**ower **W**ord (LW) and **U**pper **W**ord (UW) as shown below :

Product 1 (P1) : $MD_{LW} \times MR = P1_{UW}, P1_{LW}$

Product 2 (P2) : $MD_{UW} \times MR = P2_{UW}, P2_{LW}$

The result of the above two multiplications can be added to get the final result as shown below. The final product will have a size of three words and they are denoted as P_{W1}, P_{W2} and P_{W3}. Since, we restrict the product to 32-bit the third word PW_3 is discarded.

$$\begin{array}{ccc} & MD_{UW} & MD_{LW} \\ & & \times\ MR \\ \hline & P1_{UW} & P1_{LW} \\ P2_{UW} & P2_{LW} & \\ \hline P_{W3} & P_{W2} & P_{W1} \\ \hline \end{array}$$

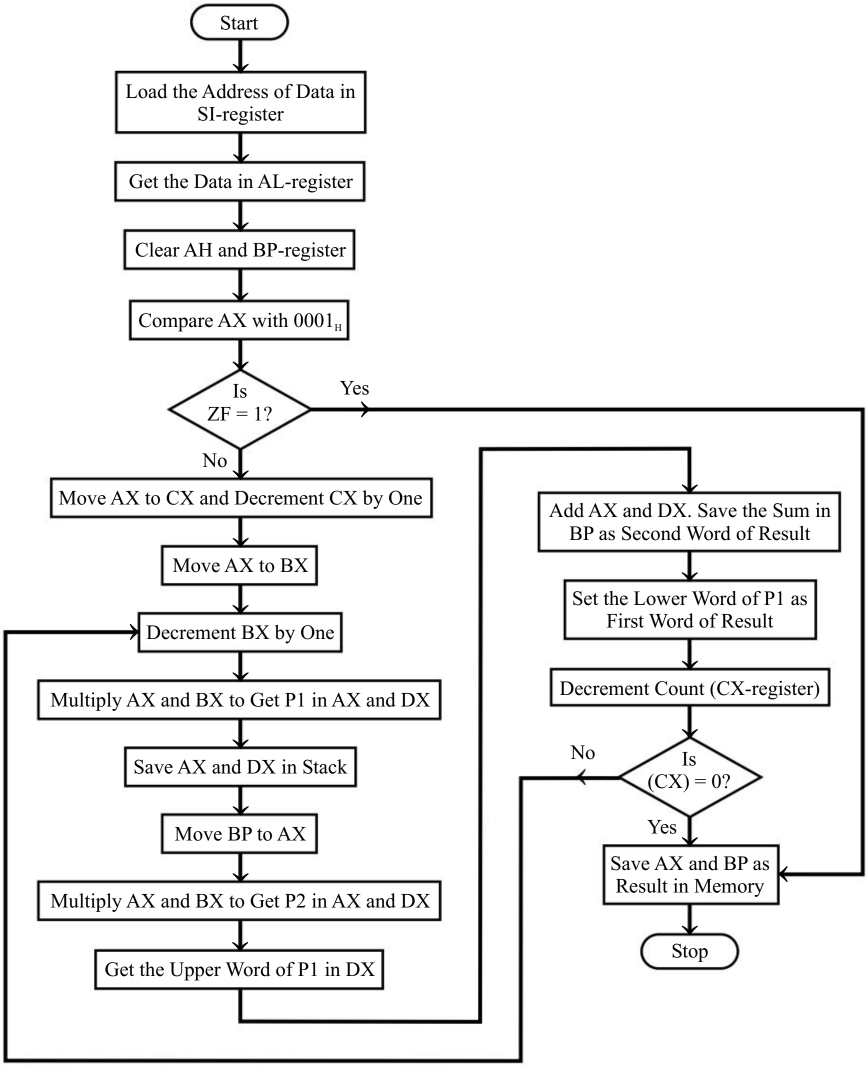
Discard ← P_{W3}

Algorithm

1. Set SI as pointer for data.
2. Get the data in AL-register and clear AH-register to convert the data to 16-bit.
3. Clear BP-register to keep initial value of second word of final product as zero.
4. Compare AX-register with 01_H.
5. Check zero flag. If zero flag is set then go to step 19, otherwise go to next step.
6. Set CX-register as count for data–1 (data minus one) multiplications.
7. Move AX-register to BX-register, so that the initial mutiplier in BX is the given data.

8. Decrement the multiplier (BX-register).
9. Multiply AX and BX to get the product1 in AX and DX.
10. Save the product1 in stack.
11. Load the second word of previous product in BP to AX-register.
12. Multiply AX and BX to get the product 2 in AX and DX.
13. Get the upper word of product1 in DX.
14. Add AX and DX to get the second word of final product in AX.
15. Move AX to BP to save the second word of final product in BP.
16. Get the first word of final product in AX-register.
17. Decrement multiplication count (CX-register).
18. If content of CX-register is not zero, then go to step 8, otherwise go to next step.
19. Store AX and BP in memory.
20. Stop.

Flowchart


Start
Load the Address of Data in SI-register
Get the Data in AL-register
Clear AH and BP-register
Compare AX with 0001_H
Is ZF = 1?
Yes
No
Move AX to CX and Decrement CX by One
Move AX to BX
Decrement BX by One
Multiply AX and BX to Get P1 in AX and DX
Save AX and DX in Stack
Move BP to AX
Multiply AX and BX to Get P2 in AX and DX
Get the Upper Word of P1 in DX
Add AX and DX. Save the Sum in BP as Second Word of Result
Set the Lower Word of P1 as First Word of Result
Decrement Count (CX-register)
No
Is (CX) = 0?
Yes
Save AX and BP as Result in Memory
Stop

Assembly language program

```
;PROGRAM TO FIND FACTORIAL OF 8-BIT DATA

CODE SEGMENT                ;Start of code segment.

        ASSUME CS:CODE      ;Assembler directive.
        ORG   1000H         ;specify program starting address.

        MOV   SI,1100H      ;Set SI as pointer for data.
        MOV   AL,[SI]       ;Get the data in AL.
        MOV   AH,00H        ;Clear AH-register.
        MOV   BP,0000H      ;Initialize upper word of the result as zero.
        CMP   AX,0001H      ;Check whether data is 01.
        JNG   STORE         ;If data is 01, then store 01 as factorial.

        MOV   CX,AX         ;Set CX as count for number of multiplications.
        DEC   CX            ;Decrement the count.

        MOV   BX,AX         ;Set the data as multiplier.
REPEAT: DEC   BX            ;Decrement the multiplier.
        MUL   BX            ;Get the product1(P1) in AX and DX registers.
        PUSH  AX            ;Save lower word of product 1 in stack.
        PUSH  DX            ;Save upper word of product 1 in stack.
        MOV   AX,BP
        MUL   BX            ;Get the product 2(P2) in AX and DX registers.
        POP   DX            ;Get the upper word of product 1 in DX-register.
        ADD   AX,DX         ;Get sum of lower word of P1 and
                            ;upper word of P2 in AX.
        MOV   BP,AX         ;Set the sum as second word of the result.
        POP   AX            ;Set lower word of P1 as first word of result.
        LOOP  REPEAT        ;Repeat multiplication until count is zero.

STORE:  MOV   [SI+1],AX     ;Store the lower word of the result in memory.
        MOV   [SI+3],BP     ;Store the upper word of the result in memory.
        HLT                 ;Halt program execution.

CODE ENDS                   ;End of code segment.
END                         ;Assembly end.
```

Assembler listing for example program 18

```
;PROGRAM TO FIND FACTORIAL OF 8-BIT DATA

0000              CODE SEGMENT               ;Start of code segment.

                          ASSUME CS:CODE     ;Assembler directive.
1000                      ORG  1000H         ;specify program starting address.

1000   BE 1100            MOV  SI,1100H      ;Set SI as pointer for data.
1003   8A 04              MOV  AL,[SI]       ;Get the data in AL.
1005   B4 00              MOV  AH,00H        ;Clear AH-register.
1007   BD 0000            MOV  BP,0000H      ;Initialize upper word of the result as zero.
100A   3D 0001            CMP  AX,0001H      ;Check whether data is 01.
100D   7E 16              JNG  STORE         ;If data is 01, then store 01 as factorial.

100F   8B C8              MOV  CX,AX         ;Set CX as count for number of multiplications.
1011   49                 DEC  CX            ;Decrement the count.

1012   8B D8              MOV  BX,AX         ;Set the data as multiplier.
1014   4B         REPEAT: DEC  BX            ;Decrement the multiplier.
1015   F7 E3              MUL  BX            ;Get the product 1(P1) in AX and DX registers.
1017   50                 PUSH AX            ;Save lower word of product 1 in stack.
1018   52                 PUSH DX            ;Save upper word of product 1 in stack.
```

```
1019   8B C5                 MOV  AX,BP
101B   F7 E3                 MUL  BX         ;Get the product 2(P2) in AX and DX registers.
101D   5A                    POP  DX         ;Get the upper word of product 1 in DX-register.
101E   03 C2                 ADD  AX,DX      ;Get sum of lower word of P1 and
                                             ;upper word of P2 in AX.
1020   8B E8                 MOV  BP,AX      ;Set the sum as second word of the result.
1022   58                    POP  AX         ;Set lower word of P1 as first word of result.
1023   E2 EF                 LOOP REPEAT     ;Repeat multiplication until count is zero.

1025   89 44 01    STORE:    MOV  [SI+1],AX  ;Store the lower word of the result in memory.
1028   89 6C 03              MOV  [SI+3],BP  ;Store the upper word of the result in memory.
102B   F4                    HLT             ;Halt program execution.

102C               CODE ENDS                 ;End of code segment.
                   END                       ;Assembly end.
```

Sample data

Input Data : $0B_H$ Output Data : 02611500_H

Memory address	Data
1100	0B
1101	00
1102	15
1103	61
1104	02

EXAMPLE PROGRAM 19 : Generation of Prime Numbers

Write an assembly language program to generate all possible prime numbers less than the given data.

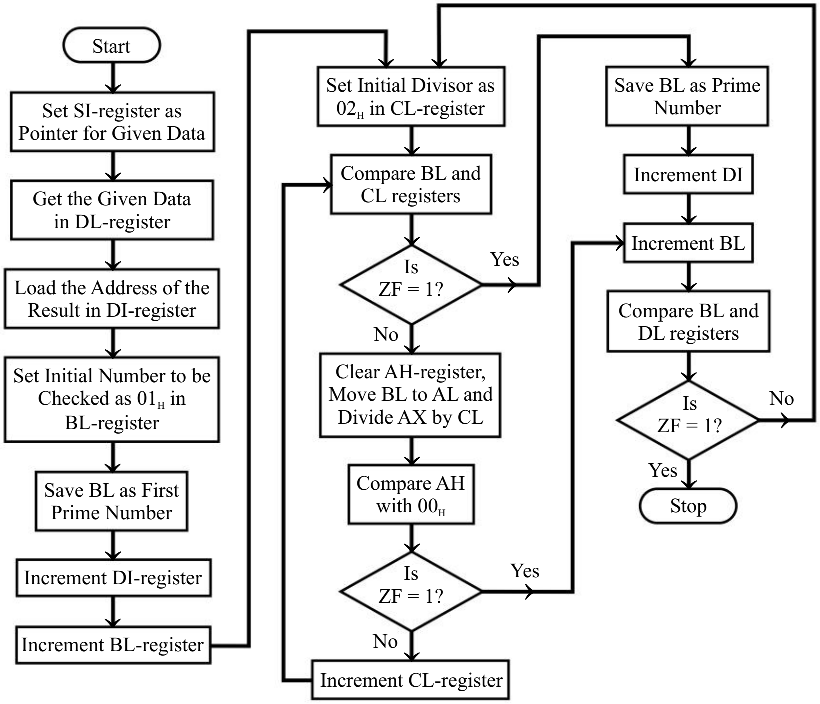
Problem Analysis

A number is prime if it is divisible only by one and the same number, and it should not be divisible by any other number. Hence to check whether a number is prime or not, we can divide the number by all possible integer less than the given number and verify the remainder. (The initial divisor is 02.) If the remainder is zero in any of the division, then the number is not prime. If the remainder is nonzero in all the divisions, then we can say the number is prime.

Algorithm

1. Set SI-register as pointer for data.
2. Load the given data in DL-register.
3. Set DI as pointer for result.
4. Initialize the number to checked as 01_H in BL-register.
5. Save 01_H as first prime number.
6. Increment the result pointer (DI).
7. Increment the number (BL-register) to be checked.
8. Load the initial divisor 02_H in CL-register.
9. Compare BL and CL registers.
10. Check zero flag. If zero flag is set then go to step 16, otherwise go to next step.
11. Clear AH-register and load the number to be checked in AL-register.
12. Divide AX by CL-register.
13. Compare the remainder (AH-register) with 00_H.
14. Check zero flag. If zero flag is set then go to step 18, otherwise go to next step.
15. Increment the divisor (CL-register) and go to step 9.
16. Save the prime number.
17. Increment the result pointer (DL-register).
18. Increment the number to be checked (BL-register)
19. Compare DL and BL registers.
20. Check zero flag. If zero flag is reset then go to step 8, otherwise stop.

Flowchart


Start
Set SI-register as Pointer for Given Data
Get the Given Data in DL-register
Load the Address of the Result in DI-register
Set Initial Number to be Checked as 01H in BL-register
Save BL as First Prime Number
Increment DI-register
Increment BL-register
Set Initial Divisor as 02H in CL-register
Compare BL and CL registers
Is ZF = 1?
Yes
No
Clear AH-register, Move BL to AL and Divide AX by CL
Compare AH with 00H
Is ZF = 1?
Yes
No
Increment CL-register
Save BL as Prime Number
Increment DI
Increment BL
Compare BL and DL registers
Is ZF = 1?
No
Yes
Stop

Assembly language program

```
;PROGRAM TO GENERATE PRIME NUMBERS

CODE SEGMENT            ;Start of code segment.

         ASSUME CS:CODE ;Assembler directive.
         ORG  1000H     ;specify program starting address.

         MOV  SI,1100H  ;Set SI-register as pointer for end data N.
         MOV  DL,[SI]   ;Get the data N in DL-register.
         MOV  DI,1200H  ;Set DI as pointer for storing prime numbers.
         MOV  BL,01H    ;Initialize the number to be checked as 01.
         MOV  [DI],BL   ;Save the first prime number.
         INC  DI        ;Increment address pointer.
         INC  BL        ;Increment the number to be checked.

GENERAT: MOV  CL,02H    ;Set initial divisor as 02.
REPEAT:  CMP  BL,CL     ;If BL=CL, jump to store.
         JZ   STORE

         MOV  AH,00H    ;Clear AH register.
         MOV  AL,BL     ;Set the number to be checked as dividend.
         DIV  CL        ;Check whether divisible by any other number.
         CMP  AH,00H    ;Check whether the remainder is zero.
         JZ   NEXT      ;If remainder is zero, verify next number.
         INC  CL        ;If remainder is non-zero then,
         JMP  REPEAT    ;increment the divisor and jump to REPEAT.

STORE:   MOV  [DI],BL   ;Save the prime number.
         INC  DI        ;Increment address pointer.
```

```
NEXT:     INC  BL        ;Increment the number to be checked.
          CMP  BL,DL     ;Check whether number to be checked is N.
          JNZ  GENERAT   ;If number to be checked is not equal to N,
          HLT            ;then continue generation,otherwise stop.

CODE ENDS                ;End of code segment.
END                      ;Assembly end.
```

Assembler listing for example program 19

```
;PROGRAM TO GENERATE PRIME NUMBERS

0000                      CODE SEGMENT    ;Start of code segment.

                          ASSUME CS:CODE  ;Assembler directive.
1000                      ORG  1000H      ;specify program starting address.

1000  BE 1100             MOV  SI,1100H   ;Set SI-register as pointer for end data N.
1003  8A 14               MOV  DL,[SI]    ;Get the data N in DL-register.
1005  BF 1200             MOV  DI,1200H   ;Set DI as pointer for storing prime numbers.
1008  B3 01               MOV  BL,01H     ;Initialize the number to be checked as 01.
100A  88 1D               MOV  [DI],BL    ;Save the first prime number.
100C  47                  INC  DI         ;Increment address pointer.
100D  FE C3               INC  BL         ;Increment the number to be checked.

100F  B1 02     GENERAT:  MOV  CL,02H     ;Set initial divisor as 02.
1011  3A D9     REPEAT:   CMP  BL,CL      ;If BL=CL, jump to store.
1013  74 0F               JZ   STORE

1015  B4 00               MOV  AH,00H     ;Clear AH-register.
1017  8A C3               MOV  AL,BL      ;Set the number to be checked as dividend.
1019  F6 F1               DIV  CL         ;Check whether divisible by any other number.
101B  80 FC 00            CMP  AH,00H     ;Check whether the remainder is zero.
101E  74 07               JZ   NEXT       ;If remainder is zero, verify next number.
1020  FE C1               INC  CL         ;If remainder is non-zero then,
1022  EB ED               JMP  REPEAT     ;increment the divisor and jump to REPEAT.

1024  88 1D     STORE:    MOV  [DI],BL    ;Save the prime number.
1026  47                  INC  DI         ;Increment address pointer.

1027  FE C3     NEXT:     INC  BL         ;Increment the number to be checked.
1029  3A DA               CMP  BL,DL      ;Check whether number to be checked is N.
102B  75 E2               JNZ  GENERAT    ;If number to be checked is not equal to N,
102D  F4                  HLT             ;then continue generation, otherwise stop.

102E            CODE ENDS                 ;End of code segment.
                END                       ;Assembly end.
```

Sample data

Input Data : $0C_H$ Output Data : 01_H 02_H 03_H 05_H 07_H $0B_H$

Memory address	Content
1100	0C
1200	01
1201	02
1202	03

Memory address	Content
1203	05
1204	07
1205	0B

EXAMPLE PROGRAM 20 : Generation of Fibanocci Series

Write an assembly language program to generate fibanocci series.

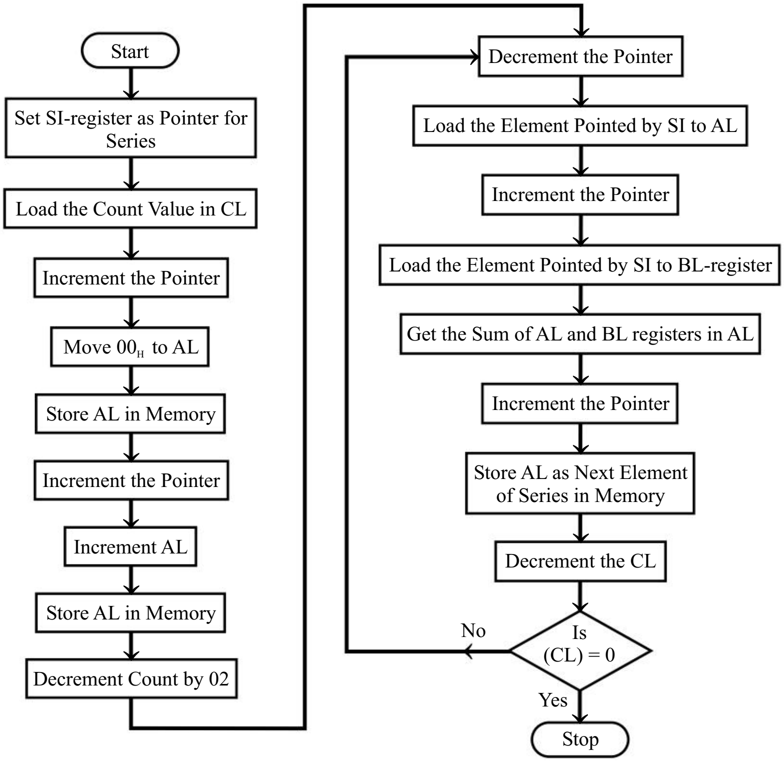
Problem Analysis

The first and second term of fibanocci series are 00_H and 01_H. The third element is given by sum of first and second element. The fourth element is given by sum of second and third element, and so on. In general an element of fibanocci series is given by sum of immediate two previous element.

Algorithm

1. Set SI-register as pointer for fibanocci series.
2. Set CL-register as count for number of elements to be generated.
3. Increment the pointer (SI).
4. Initialize the first element of fibanocci series as 00_H in AL-register.
5. Store first element in memory.
6. Increment the pointer (SI).
7. Increment AL to get second element (01_H) of fibanocci series in AL-register.
8. Store the second element in memory.
9. Decrement the count (CL-register) by 02.
10. Decrement the pointer (SI).
11. Get the element prior to last generated element in AL.
12. Increment the pointer (SI).
13. Get the last generated element in BL.
14. Add the previous two element (AL and BL) to get the next element in AL.
15. Increment the pointer.
16. Store the next element (AL) of the fibanocci series in memory.
17. Decrement the count (CL-register).
18. If the content of CL is not zero then go to step 10, otherwise stop.

Flowchart



Assembly language program

```
;PROGRAM TO GENERATE FIBANOCCI SERIES

CODE SEGMENT            ;Start of code segment.

         ASSUME CS:CODE ;Assembler directive.
         ORG  1000H     ;specify program starting address.

         MOV  SI,1100H  ;Set SI-register as pointer for series.
         MOV  CL,[SI]   ;Set CL as count for number of elements of series.
         INC  SI        ;Increment the pointer.
         MOV  AL,00H    ;Load the first element of the series in AL.
         MOV  [SI],AL   ;Save the first element of the series.
         INC  SI        ;Increment the pointer.
         INC  AL        ;Get the second number of the series in AL.
         MOV  [SI],AL   ;Save the second number of the series.
         SUB  CL,02H    ;Decrement the count by two.

GENERAT: DEC  SI        ;Let SI point to element prior to last element
                        ;of generated series.
         MOV  AL,[SI]   ;Get the element prior to last element of
                        ;generated series in AL.
         INC  SI        ;SI point to last element of generated series.
         MOV  BL,[SI]   ;Get last element of generated series in BL.
         ADD  AL,BL     ;Get the next element of the series in AL.
         INC  SI        ;Increment the pointer.
         MOV  [SI],AL   ;Save next element of series in memory.
         LOOP GENERAT   ;Decrement count and continue generation
                        ;until count is zero.
         HLT            ;Halt program execution.

CODE ENDS               ;End of code segment.
END                     ;Assembly end.
```

Assembler listing for example program 20

```
;PROGRAM TO GENERATE FIBANOCCI SERIES

0000            CODE SEGMENT             ;Start of code segment.

                          ASSUME CS:CODE ;Assembler directive.
1000                      ORG  1000H     ;specify program starting address.

1000  BE 1100             MOV  SI,1100H  ;Set SI-register as pointer for series.
1003  8A 0C               MOV  CL,[SI]   ;Set CL as count for number of elements of series.
1005  46                  INC  SI        ;Increment the pointer.
1006  B0 00               MOV  AL,00H    ;Load the first element of the series in AL.
1008  88 04               MOV  [SI],AL   ;Save the first element of the series.
100A  46                  INC  SI        ;Increment the pointer.
100B  FE C0               INC  AL        ;Get the second number of the series in AL.
100D  88 04               MOV  [SI],AL   ;Save the second number of the series.
100F  80 E9 02            SUB  CL,02H    ;Decrement the count by two.

1012  4E        GENERAT:  DEC  SI        ;Let SI point to element prior to last element
                                         ;of generated series.
1013  8A 04               MOV  AL,[SI]   ;Get the element prior to last element of
                                         ;generated series in AL.
1015  46                  INC  SI        ;SI point to last element of generated series.
1016  8A 1C               MOV  BL,[SI]   ;Get last element of generated series in BL.
1018  02 C3               ADD  AL,BL     ;Get the next element of the series in AL.
101A  46                  INC  SI        ;Increment the pointer.
101B  88 04               MOV  [SI],AL   ;Save next element of series in memory.
```

```
101D  E2 F3            LOOP GENERAT    ;Decrement count and continue generation
                                       ;until count is zero.
101F  F4               HLT             ;Halt program execution.

1020          CODE ENDS                ;End of code segment.
              END                      ;Assembly end.
```

Sample Data

Input Data : 08_H

Output Data :00_H
01_H
01_H
02_H
03_H
05_H
08_H
$0D_H$

Memory address	Content
1100	08
1101	00
1102	01
1103	01
1104	02

Memory address	Content
1105	03
1106	05
1107	08
1108	0D

EXAMPLE PROGRAM 21 : Matrix Addition

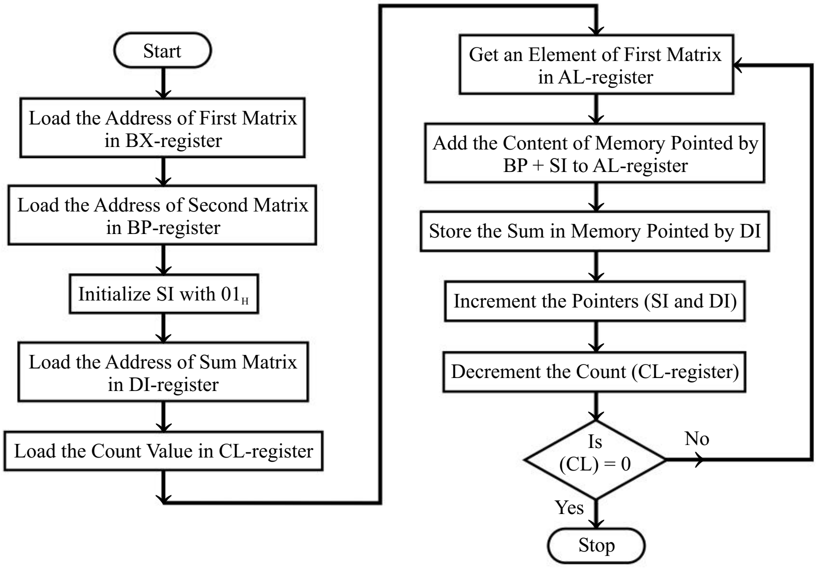
Write an assembly language program to add two numbers of 3×3 matrices.

Problem Analysis

While storing the matrices in the memory, the first row elements are stored first, followed by second row elements and then third row elements. For addition operation the matrices can be addressed as an array with number of elements in the matrices as the count value.

The two input matrices can be stored in different memory areas in the same order as mentioned above. The base registers BX and BP can be used to hold the base address of the input matrices and SI-register can be used as pointer for the elements in the matrices. Another index register DI can be used as pointer to store the sum matrix.

Flowchart


Start
Load the Address of First Matrix in BX-register
Load the Address of Second Matrix in BP-register
Initialize SI with 01_H
Load the Address of Sum Matrix in DI-register
Load the Count Value in CL-register
Get an Element of First Matrix in AL-register
Add the Content of Memory Pointed by BP + SI to AL-register
Store the Sum in Memory Pointed by DI
Increment the Pointers (SI and DI)
Decrement the Count (CL-register)
Is (CL) = 0
No
Yes
Stop

Algorithm

1. Load the base address of first input matrix in BX-register.
2. Load the base address of second input matrix in BP-register.
3. Set SI-register as index (or pointer) for elements of the matrix.

4. Set DI-register as pointer for sum matrix.
5. Load the count value in CL-register.
6. Get an element of first matrix in AL-register.
7. Add the corresponding element of second matrix to AL-register.
8. Store the sum in memory.
9. Increment the SI and DI registers.
10. Repeat steps 6 to 9 until the count value in CL-register is zero.
11. Stop.

Assembly language program

```
;PROGRAM TO ADD TWO 3X3 MATRIX

CODE SEGMENT              ;Start of code segment.

        ASSUME CS:CODE    ;Assembler directive.
        ORG  1000H        ;specify program starting address.

        MOV  BX,1300H     ;Load the base address of 1st input matrix in BX.
        MOV  BP,1400H     ;Load the base address of 2nd input matrix in BP.
        MOV  SI,0001H     ;Initialize pointer for element of matrix.
        MOV  DI,1501H     ;Set DI-register as pointer for sum matrix.
        MOV  CL,09H       ;Set CL as count for elements in matrix.

REPEAT: MOV  AL,[BX+SI]   ;Get an element of 1st matrix in AL.
        ADD  AL,[BP+SI]   ;Add corresponding element of 2nd matrix to AL.
        MOV  [DI],AL      ;Store the sum of an element in memory.
        INC  SI           ;Increment the pointers.
        INC  DI
        LOOP REPEAT       ;Repeat addition until count is zero.
        HLT               ;Halt program execution.

CODE ENDS                 ;End of code segment.
END                       ;Assembly end.
```

Assembler listing for example program 21

```
;PROGRAM TO ADD TWO 3X3 MATRIX

0000              CODE SEGMENT              ;Start of code segment.

                          ASSUME CS:CODE    ;Assembler directive.
1000                      ORG  1000H        ;specify program starting address.

1000   BB 1300            MOV  BX,1300H     ;Load the base address of 1st input matrix in BX.
1003   BD 1400            MOV  BP,1400H     ;Load the base address of 2nd input matrix in BP.
1006   BE 0001            MOV  SI,0001H     ;Initialize pointer for element of matrix.
1009   BF 1501            MOV  DI,1501H     ;Set DI-register as pointer for sum matrix.
100C   B1 09              MOV  CL,09H       ;Set CL as count for elements in matrix.

100E   8A 00      REPEAT: MOV  AL,[BX+SI]   ;Get an element of 1st matrix in AL.
1010   02 02              ADD  AL,[BP+SI]   ;Add corresponding element of 2nd matrix to AL.
1012   88 05              MOV  [DI],AL      ;Store the sum of an element in memory.
1014   46                 INC  SI           ;Increment the pointers.
1015   47                 INC  DI
1016   E2 F6              LOOP REPEAT       ;Repeat addition until count is zero.
1018   F4                 HLT               ;Halt program execution.

1019              CODE ENDS                 ;End of code segment.
                  END                       ;Assembly end.
```

Sample data

MATRIX 1 : 01 02 03 04 05 06 07 08 09

Memory address	Content
1301	01
1302	02
1303	03
1304	04
1305	05
1306	06
1307	07
1308	08
1309	09

MATRIX 2 : F0 E1 D2 C3 B4 A5 96 87 78

Memory address	Content
1401	F0
1402	E1
1403	D2
1404	C3
1405	B4
1406	A5
1407	96
1408	87
1409	78

SUM MATRIX : F1 E3 D5 C7 B9 AB 9D 8F 81

Memory address	Content
1501	F1
1502	E3
1503	D5
1504	C7
1505	B9
1506	AB
1507	9D
1508	8F
1509	81

EXAMPLE PROGRAM 22 : Matrix Multiplication

Write an assembly language program to multiply two numbers of 3×3 matrices.

Problem Analysis

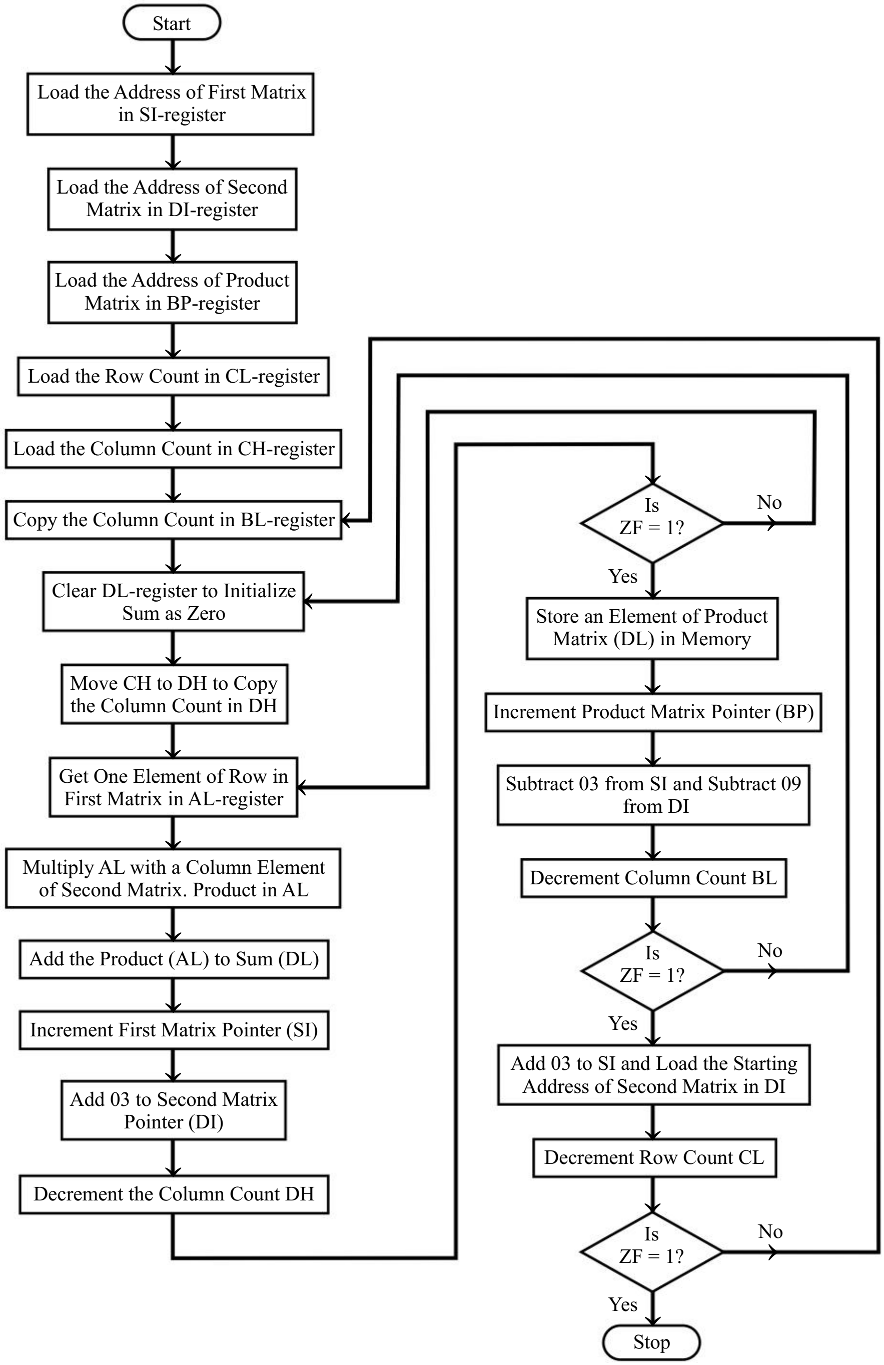
While storing the matrices in memory, the first row elements are stored first, followed by second row elements and then third row elements. For multiplication operation the matrices should be addressed as two dimensional array. Here the two-dimensional arrays are addressed by using pointers and counters.

The SI and DI registers are used as address pointers for two input matrices. The CL and CH registers are used as row and column count, respectively. The BP-register is used as pointer for storing the elements of product matrix.

Algorithm

1. Load the address of first input matrix in SI-register.
2. Load the address of second input matrix in DI-register.
3. Load the address of product matrix in BP-register.
4. Load the row count in CL-register.
5. Load the column count in CH-register.
6. Copy the column count in BL-register (Let it be second column count).
7. Initialize sum as zero in DL-register.
8. Get the column count in DH-register.
9. Get a row element of first matrix in AL-register.
10. Multiply a column element of second matrix with AL, the product will be in AL. (Because it is assumed that the product does not exceed 8-bit.)
11. Add the product (AL) to sum (DL).
12. Increment SI to point to next element of same row in first input matrix.
13. Increment DI by 03 to point to next element of same column in second input matrix.
14. Decrement the column count (DH-register).
15. Check zero flag. If zero flag is reset then go to step 9, otherwise go to next step.
16. Store an element of product matrix (DL) in memory.
17. Increment the product matrix pointer (BP).
18. Subtract 03_H from SI to point to the first element of same row.
19. Subtract 09_H from DI to point to first element of next row.
20. Decrement BL-register (second column count).
21. Check zero flag. If zero flag is reset then go to step 7, otherwise go to next step.
22. Add 03_H to SI to point to first element of next row in first matrix.
23. Load the starting address of second matrix in DI-register.
24. Decrement the row count.
25. Check zero flag. If zero is reset then go to step 6, otherwise stop.

Flowchart


Start
Load the Address of First Matrix in SI-register
Load the Address of Second Matrix in DI-register
Load the Address of Product Matrix in BP-register
Load the Row Count in CL-register
Load the Column Count in CH-register
Copy the Column Count in BL-register
Clear DL-register to Initialize Sum as Zero
Move CH to DH to Copy the Column Count in DH
Get One Element of Row in First Matrix in AL-register
Multiply AL with a Column Element of Second Matrix. Product in AL
Add the Product (AL) to Sum (DL)
Increment First Matrix Pointer (SI)
Add 03 to Second Matrix Pointer (DI)
Decrement the Column Count DH
Is ZF = 1?
No
Yes
Store an Element of Product Matrix (DL) in Memory
Increment Product Matrix Pointer (BP)
Subtract 03 from SI and Subtract 09 from DI
Decrement Column Count BL
Is ZF = 1?
No
Yes
Add 03 to SI and Load the Starting Address of Second Matrix in DI
Decrement Row Count CL
Is ZF = 1?
No
Yes
Stop

Assembly language program

```
;PROGRAM TO MULTIPLY TWO 3X3 MATRIX

CODE SEGMENT                ;Start of code segment.

          ASSUME CS:CODE    ;Assembler directive.
          ORG 1000H         ;specify program starting address.

          MOV SI,1301H      ;Set SI as pointer for first input matrix.
          MOV DI,1401H      ;Set DI as pointer for second input matrix.
          MOV BP,1501H      ;Set BP as pointer for product matrix.
          MOV CL,03H        ;Set CL as count for elements in a row.
          MOV CH,03H        ;Set CH as count for elements in a column.

REPEAT3:  MOV BL,CH         ;Copy the column count in BL-register.

REPEAT2:  MOV DL,00H        ;Initialize sum as zero.
          MOV DH,CH         ;Get the column count in DH.

REPEAT1:  MOV AL,[SI]       ;Get one element of the row in AL-register.
          MUL [DI]          ;Get product of row and column element in AL.
          ADD DL,AL         ;Add the product to sum.
          INC SI            ;Increment the first input matrix pointer.
          ADD DI,03H        ;Let DI point to next element of
                            ;same column of 2nd matrix.
          DEC DH            ;Decrement the column count.
          JNZ REPEAT1       ;Repeat multiplication and addition until
                            ;DH count is zero.

          MOV [BP],DL       ;Store an element of product matrix in memory.
          INC BP            ;Increment the product matrix pointer.
          SUB SI,03H        ;Make SI to point to first element of the row.
          SUB DI,09H
          INC DI
          DEC BL            ;Decrement the column count.
          JNZ REPEAT2       ;Repeat multiplication and addition of a row
                            ;in 1st matrix with next column of 2nd matrix.
          ADD SI,03H        ;Let SI point to first element of
                            ;next row of 1st matrix.
          MOV DI,1401H      ;Make DI to point to first element of 2nd matrix.
          DEC CL            ;Decrement the row count.
          JNZ REPEAT3       ;Repeat multiplication and addition of next row
                            ;in 1st matrix with all column of 2nd matrix.
          HLT               ;Halt program execution.

CODE ENDS                   ;End of code segment.
END                         ;Assembly end.
```

Assembler listing for example program 22

```
;PROGRAM TO MULTIPLY TWO 3X3 MATRIX

0000                CODE SEGMENT              ;Start of code segment.

                             ASSUME CS:CODE   ;Assembler directive.
1000                         ORG  1000H       ;specify program starting address.

1000   BE 1301               MOV  SI,1301H    ;Set SI as pointer for first input matrix.
1003   BF 1401               MOV  DI,1401H    ;Set DI as pointer for second input matrix.
1006   BD 1501               MOV  BP,1501H    ;Set BP as pointer for product matrix.
1009   B1 03                 MOV  CL,03H      ;Set CL as count for elements in a row.
100B   B5 03                 MOV  CH,03H      ;Set CH as count for elements in a column.
100D   8A DD        REPEAT3: MOV  BL,CH       ;Copy the column count in BL-register.
```

```
100F   B2 00      REPEAT2: MOV  DL,00H     ;Initialize sum as zero.
1011   8A F5               MOV  DH,CH      ;Get the column count in DH.

1013   8A 04      REPEAT1: MOV  AL,[SI]    ;Get one element of the row in AL-register.
1015   F6 25               MUL  [DI]       ;Get product of row and column element in AL.
1017   02 D0               ADD  DL,AL      ;Add the product to sum.
1019   46                  INC  SI         ;Increment the first input matrix pointer.
101A   83 C7 03            ADD  DI,03H     ;Let DI point to next element of
                                           ;same column of 2nd matrix.
101D   FE CE               DEC  DH         ;Decrement the column count.
101F   75 F2               JNZ  REPEAT1    ;Repeat multiplication and addition until
                                           ;DH count is zero.

1021   88 56 00            MOV  [BP],DL    ;Store an element of product matrix in memory.
1024   45                  INC  BP         ;Increment the product matrix pointer.
1025   83 EE 03            SUB  SI,03H     ;Make SI to point to first element of the row.
1028   83 EF 09            SUB  DI,09H
102B   47                  INC  DI
102C   FE CB               DEC  BL         ;Decrement the column count.
102E   75 DF               JNZ  REPEAT2    ;Repeat multiplication and addition of a row
                                           ;in 1st matrix with next column of 2nd matrix.
1030   83 C6 03            ADD  SI,03H     ;Let SI point to first element of
                                           ;next row of 1st matrix.
1033   BF 1401             MOV  DI,1401H   ;Make DI to point to first element of 2nd matrix.
1036   FE C9               DEC  CL         ;Decrement the row count.
1038   75 D3               JNZ  REPEAT3    ;Repeat multiplication and addition of next row
                                           ;in 1st matrix with all column of 2nd matrix.
103A   F4                  HLT             ;Halt program execution.

103B              CODE ENDS                ;End of code segment.
                  END                      ;Assembly end.
```

Sample data

MATRIX 1 : 01 01 01 02 02 02 03 03 03

Memory address	Content
1301	01
1302	01
1303	01
1304	02
1305	02
1306	02
1307	03
1308	03
1309	03

MATRIX 2 : 04 04 04 05 05 05 06 06 06

Memory address	Content
1401	04
1402	04
1403	04
1404	05
1405	05
1406	05
1407	06
1408	06
1409	06

PRODUCT MATRIX : 0F 0F 0F 1E 1E 1E 2D 2D 2D

Memory address	Content
1501	0F
1502	0F
1503	0F
1504	1E
1505	1E
1506	1E
1507	2D
1508	2D
1509	2D

EXAMPLE PROGRAM 23 : BCD to Binary Conversion

Write an assembly language program to convert a BCD data (2-digit/8-bit) to binary.

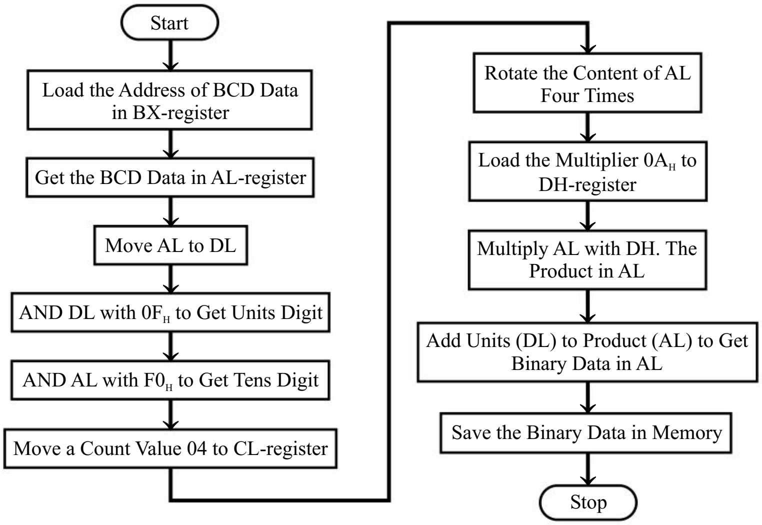
Problem Analysis

The 2-digit BCD data will have units digit and tens digit. When the tens digit (upper nibble) is multiplied by $0A_H$ and the product is added to units digit (lower nibble), the result will be in binary, because the microprocessor will perform binary arithmetic.

Algorithm

1. Load the address of BCD data in BX-register.
2. Get the BCD data in AL-register.
3. Copy the BCD data in DL-register.
4. Logically AND DL with $0F_H$ to mask upper nibble and get units digit in DL.
5. Logically AND AL with $F0_H$ to mask lower nibble.
6. Move the count value for rotation in CL-register.
7. Rotate the content of AL to move the upper nibble to lower nibble position.
8. Move $0A_H$ to DH-register.
9. Multiply AL with DH-register. The product will be in AL-register.
10. Add the units digit in DL-register to product in AL-register.
11. Save the binary data (AL) in memory.
12. Stop.

Flowchart


Start
Load the Address of BCD Data in BX-register
Get the BCD Data in AL-register
Move AL to DL
AND DL with 0F_H to Get Units Digit
AND AL with F0_H to Get Tens Digit
Move a Count Value 04 to CL-register
Rotate the Content of AL Four Times
Load the Multiplier 0A_H to DH-register
Multiply AL with DH. The Product in AL
Add Units (DL) to Product (AL) to Get Binary Data in AL
Save the Binary Data in Memory
Stop

Assembly language program

```
;PROGRAM TO CONVERT A BCD DATA TO BINARY DATA

CODE SEGMENT            ;Start of code segment.

        ASSUME CS:CODE  ;Assembler directive.
        ORG  1000H      ;specify program starting address.
        MOV  BX,1100H   ;Load the address of the data in BX-register.
        MOV  AL,[BX]    ;Get the BCD data in AL-register.
        MOV  DL,AL      ;Copy the data in DL-register.
        AND  DL,0FH     ;Mask upper nibble (tens digit).
        AND  AL,0F0H    ;Mask lower nibble (units digit).
        MOV  CL,4       ;Rotate the upper nibble to lower nibble position.
        ROR  AL,CL
        MOV  DH,0AH     ;Set multiplier as 0AH.
        MUL  DH         ;Multiply tens digit by 0AH,
                        ;the product will be in AL.
        ADD  AL,DL      ;Get sum of units digit and product in AL.
        MOV  [BX+1],AL  ;Save the binary data in memory.
        HLT             ;Halt program execution.
```

```
CODE ENDS               ;End of code segment.
END                     ;Assembly end.
```

Assembler listing for example program 23

```
;PROGRAM TO CONVERT A BCD DATA TO BINARY DATA

0000            CODE SEGMENT       ;Start of code segment.

                   ASSUME CS:CODE  ;Assembler directive.
1000               ORG  1000H      ;specify program starting address.

1000   BB 1100     MOV  BX,1100H   ;Load the address of the data in BX-register.
1003   8A 07       MOV  AL,[BX]    ;Get the BCD data in AL-register.
1005   8A D0       MOV  DL,AL      ;Copy the data in DL-register.
1007   80 E2 0F    AND  DL,0FH     ;Mask upper nibble (tens digit).
100A   24 F0       AND  AL,0F0H    ;Mask lower nibble (units digit).
100C   B1 04       MOV  CL,4       ;Rotate the upper nibble to lower nibble position.
100E   D2 C8       ROR  AL,CL
1010   B6 0A       MOV  DH,0AH     ;Set multiplier as 0AH.
1012   F6 E6       MUL  DH         ;Multiply tens digit by 0AH,
                                   ;the product will be in AL.
1014   02 C2       ADD  AL,DL      ;Get sum of units digit and product in AL.
1016   88 47 01    MOV  [BX+1],AL  ;Save the binary data in memory.
1019   F4          HLT             ;Halt program execution.

101A            CODE ENDS          ;End of code segment.
                END                ;Assembly end.
```

Sample Data

Input Data : 75_{10} Output Data : $4B_H$

Memory address	Content
1100	75
1101	4B

EXAMPLE PROGRAM 24 : Binary to BCD Conversion

Write an assembly language program to convert 8-bit binary data to BCD.

Problem Analysis

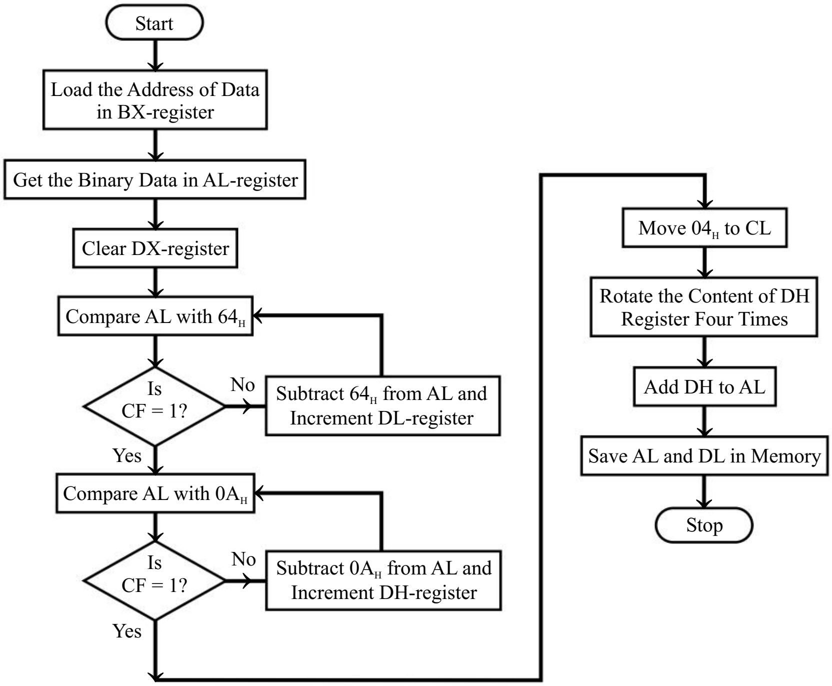
The maximum value of 8-bit binary is FF_H. The BCD equivalent of FF_H is 256_{10}. Hence, when an 8-bit binary is converted to BCD, the BCD data will have hundreds, tens and units digit. We can use two counters to count hundreds and tens. Initially the counters are cleared. First let us subtract all hundreds from the given data and for each subtraction hundreds counter is incremented by one. Then we can subtract all tens from the given data and for each subtraction tens counter is incremented by one. The remaining will be units. The tens and units can be combined as 2-digit BCD and stored in the memory. The hundreds can be separately stored in the memory.

Algorithm

1. Load the address of data in BX-register.
2. Get the binary data in AL-register.
3. Clear DX-register for storing hundreds and tens.
4. Compare AL with 64_H (100_{10}).
5. Check carry flag. If carry flag is set then go to step 9, otherwise go to next step.
6. Subtract 64_H (100_{10}) from AL-register.
7. Increment hundreds register (DL).
8. Go to step 4.

9. Compare AL with $0A_H$ (10_{10}).
10. Check carry flag. If carry flag is set then go to step 14, otherwise go to next step.
11. Subtract $0A_H$ (10_{10}) from AL-register.
12. Increment tens register (DH).
13. Go to step 9.
14. Move the count value 04_H for rotation in CL-register.
15. Rotate the content of DH four times.
16. Add DH to AL to combine tens and units as 2-digit BCD.
17. Save AL and DL in memory.
18. Stop.

Flowchart


Start
Load the Address of Data in BX-register
Get the Binary Data in AL-register
Clear DX-register
Compare AL with 64H
Is CF = 1?
No
Subtract 64H from AL and Increment DL-register
Yes
Compare AL with 0AH
Is CF = 1?
No
Subtract 0AH from AL and Increment DH-register
Yes
Move 04H to CL
Rotate the Content of DH Register Four Times
Add DH to AL
Save AL and DL in Memory
Stop

Assembly language program

```
;PROGRAM TO CONVERT A BINARY DATA TO BCD DATA

CODE SEGMENT          ;Start of code segment.

      ASSUME CS:CODE  ;Assembler directive.
      ORG  1000H      ;specify program starting address.

      MOV  BX,1100H   ;Load the address of the data in BX-register.
      MOV  AL,[BX]    ;Get the binary data in AL-register.
      MOV  DX,0000H   ;Clear DX for storing hundreds and tens.

HUND: CMP  AL,64H     ;Compare whether data is less than 100 (or 64H).
      JC   TEN        ;If the data is less than 100 then jump to TEN.
      SUB  AL,64H     ;If data greater than 100, subtract  hundred.
      INC  DL         ;Increment hundreds register.
      JMP  HUND        ;Repeat subtraction of hundred.

TEN:  CMP  AL,0AH      ;Compare whether data is less than 10 (or 0AH).
      JC   UNIT        ;If data is less than 10 then jump to UNIT.
      SUB  AL,0AH      ;If data greater than 10 then, subtract ten.
      INC  DH          ;Increment tens register.
      JMP  TEN         ;Repeat subtraction of ten.

UNIT: MOV  CL,4        ;Rotate tens digit to upper nibble position.
      ROL  DH,CL
      ADD  AL,DH       ;Combine tens and units digit.
      MOV  [BX+1],AL   ;Save tens and units in memory.
      MOV  [BX+2],DL   ;Save hundreds in memory.
      HLT              ;Halt program execution.

CODE ENDS              ;End of code segment.
END                    ;Assembly end.
```

Assembler listing for example program 24

```
;PROGRAM TO CONVERT A BINARY DATA TO BCD DATA

0000            CODE SEGMENT        ;Start of code segment.

                      ASSUME CS:CODE ;Assembler directive.
1000                  ORG 1000H      ;specify program starting address.

1000 BB 1100          MOV BX,1100H   ;Load the address of the data in BX-register.
1003 8A 07            MOV AL,[BX]    ;Get the binary data in AL-register.
1005 BA 0000          MOV DX,0000H   ;Clear DX for storing hundreds and tens.

1008 3C 64      HUND: CMP AL,64H     ;Compare whether data is less than 100 (or 64H).
100A 72 06            JC  TEN        ;If the data is less than 100 then jump to TEN.
100C 2C 64            SUB AL,64H     ;If data greater than 100, subtract hundred.
100E FE C2            INC DL         ;Increment hundreds register.
1010 EB F6            JMP HUND       ;Repeat subtraction of hundred.

1012 3C 0A      TEN:  CMP AL,0AH     ;Compare whether data is less than 10 (or 0AH).
1014 72 06            JC  UNIT       ;If data is less than 10 then jump to UNIT.
1016 2C 0A            SUB AL,0AH     ;If data greater than 10 then, subtract ten.
1018 FE C6            INC DH         ;Increment tens register.

101A EB F6            JMP TEN        ;Repeat subtraction of ten.

101C B1 04      UNIT: MOV CL,4       ;Rotate tens digit to upper nibble position.
101E D2 C6            ROL DH,CL
1020 02 C6            ADD AL,DH      ;Combine tens and units digit.
1022 88 47 01         MOV [BX+1],AL  ;Save tens and units in memory.
1025 88 57 02         MOV [BX+2],DL  ;Save hundreds in memory.
1028 F4               HLT            ;Halt program execution.

1029            CODE ENDS            ;End of code segment.
                END                  ;Assembly end.
```

Sample Data

Input Data : $E4_H$ Output Data : 0228_{10}

Memory address	Content
1100	E4
1101	28
1102	02

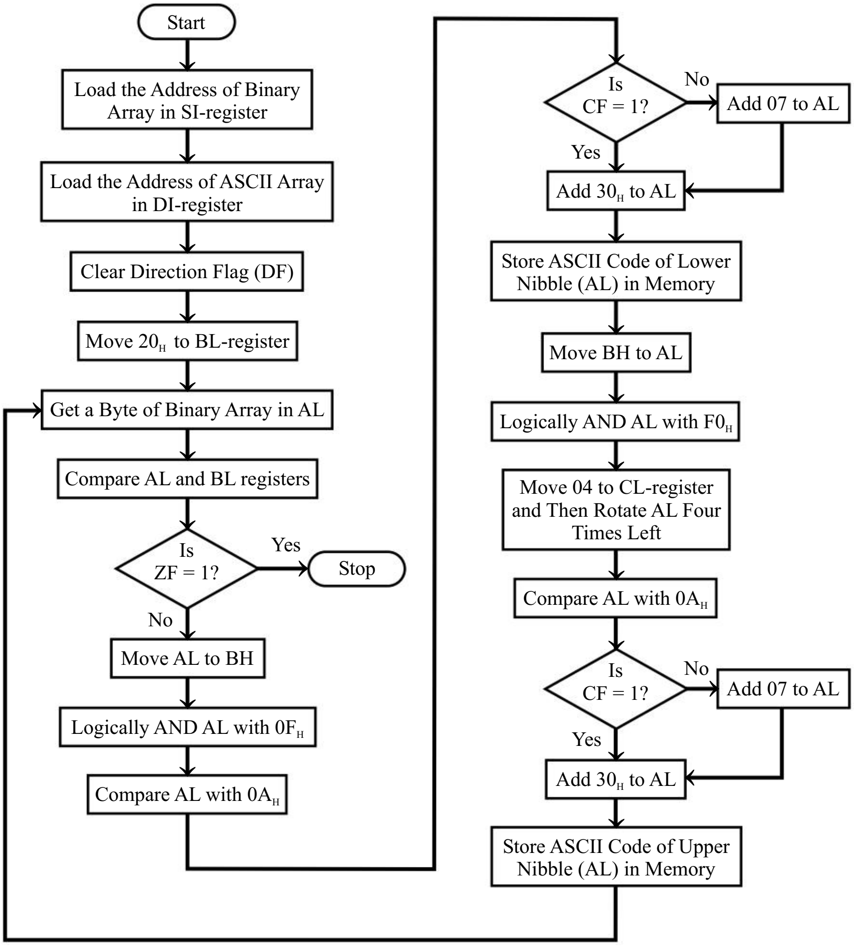
EXAMPLE PROGRAM 25 : Binary to ASCII Conversion

Write an assembly language program to convert an array of 8-bit binary data to ASCII code.

Problem Analysis

The 8-bit binary can be represented by 2-digit hexa. Each hexa digit can be converted to an 8-bit ASCII code (i.e., each nibble of binary data can be converted to an 8-bit ASCII code). The hexa digit 0 through 9 are represented by 30 to 39 in ASCII and the hexa digit A through F are represented by 41 to 46 in ASCII. Therefore, the 8-bit binary data is split into two nibbles : lower nibble and upper nibble. Then check each nibble whether it is less than $0A_H$ or not, if it is less than $0A_H$ then add 30_H to convert to ASCII or of it is greater than/equal to $0A_H$ then add 37_H to convert to ASCII.

Flowchart


Start
Load the Address of Binary Array in SI-register
Load the Address of ASCII Array in DI-register
Clear Direction Flag (DF)
Move 20H to BL-register
Get a Byte of Binary Array in AL
Compare AL and BL registers
Is ZF = 1?
Yes
Stop
No
Move AL to BH
Logically AND AL with 0FH
Compare AL with 0AH
Is CF = 1?
No
Add 07 to AL
Yes
Add 30H to AL
Store ASCII Code of Lower Nibble (AL) in Memory
Move BH to AL
Logically AND AL with F0H
Move 04 to CL-register and Then Rotate AL Four Times Left
Compare AL with 0AH
Is CF = 1?
No
Add 07 to AL
Yes
Add 30H to AL
Store ASCII Code of Upper Nibble (AL) in Memory

Algorithm

1. Set SI as pointer for binary array.
2. Set DI as pointer for ASCII array.
3. Clear **D**irection **F**lag (DF) for autoincrement of pointers.
4. Move the end character 20_H to BL-register.
5. Load a byte of binary array in AL-register.
6. Compare AL and BL.
7. Check zero flag. If zero flag is set then go to step 24, otherwise go to next step.
8. Save the byte in BH-register.
9. Logically AND AL with $0F_H$ to mask the upper nibble.
10. Compare AL with $0A_H$.
11. Check carry flag. If carry flag is set then go to step 13, otherwise go to next step.
12. Add 07_H to AL-register.
13. Add 30_H to AL-register.
14. Store AL-register (ASCII code for lower nibble) in memory.
15. Move the saved byte in BH register to AL-register.
16. Logically AND AL-register with $F0_H$ to mask the lower nibble.
17. Rotate the upper nibble in AL-register to lower nibble position.
18. Compare AL with $0A_H$.
19. Check carry flag. If carry flag is set then go to step 21, otherwise go to next step.
20. Add 07_H to AL-register.
21. Add 30_H to AL-register.
22. Store AL-register (ASCII code for upper nibble) in memory.
23. Jump to step 5.
24. Stop.

Assembly language program

```
;PROGRAM TO CONVERT AN ARRAY OF BINARY DATA TO ASCII DATA

CODE SEGMENT          ;Start of code segment.

       ASSUME CS:CODE ;Assembler directive.
       ORG  1000H     ;specify program starting address.
       MOV  SI,1100H  ;Set SI as pointer for binary array.
       MOV  DI,1400H  ;Set DI as pointer for ASCII array.
       CLD            ;Clear DF for autoincrement of SI and DI.
       MOV  BL,20H    ;Load the end character in BL-register.

NEXT:  LODSB          ;Load a byte of binary data in AL-register.
       CMP  AL,BL     ;Check for end of string.
       JE   EXIT      ;If zero flag is set, then go to EXIT.

       MOV  BH,AL     ;Save the byte in BH-register.
       AND  AL,0FH    ;Mask the upper nibble of binary data.
       CMP  AL,0AH    ;Check whether lower nibble less than 0AH.
       JL   SKIP1     ;If lower nibble less than 0AH, then add 30H.
       ADD  AL,07H    ;If lower nibble is greater than 0AH, then add 37H.
SKIP1: ADD  AL,30H
       STOSB          ;Store ASCII code of lower nibble in memory.

       MOV  AL,BH     ;Get the data in AL-register.
       AND  AL,0F0H   ;Mask the lower nibble.
       MOV  CL,04H    ;Rotate upper nibble to lower nibble position.
       ROL  AL,CL
       CMP  AL,0AH    ;Check whether upper nibble is less than 0AH.
       JL   SKIP2     ;If upper nibble less than 0AH, then add 30H.
       ADD  AL,07H    ;If upper nibble greater than 0AH, then add 37H.
```

```
SKIP2: ADD  AL,30H
       STOSB           ;Store ASCII code of upper nibble in memory.
       JMP  NEXT       ;Jump to NEXT to convert next byte.
EXIT:  HLT             ;Halt program execution.

CODE ENDS              ;End of code segment.
END                    ;Assembly end.
```

Assembler listing for example program 25

```
;PROGRAM TO CONVERT AN ARRAY OF BINARY DATA TO ASCII DATA

0000               CODE SEGMENT           ;Start of code segment.

                          ASSUME CS:CODE  ;Assembler directive.
1000                      ORG  1000H      ;specify program starting address.
1000    BE 1100           MOV  SI,1100H   ;Set SI as pointer for binary array.
1003    BF 1400           MOV  DI,1400H   ;Set DI as pointer for ASCII array.
1006    FC                CLD             ;Clear DF for autoincrement of SI and DI.
1007    B3 20             MOV  BL,20H     ;Load the end character in BL-register.

1009    AC         NEXT:  LODSB           ;Load a byte of binary data in AL-register.
100A    3A C3             CMP  AL,BL      ;Check for end of string.
100C    74 20             JE   EXIT       ;If zero flag is set, then go to EXIT.

100E    8A F8             MOV  BH,AL      ;Save the byte in BH-register.
1010    24 0F             AND  AL,0FH     ;Mask the upper nibble of binary data.
1012    3C 0A             CMP  AL,0AH     ;Check whether lower nibble less than 0AH.
1014    7C 02             JL   SKIP1      ;If lower nibble less than 0AH, then add 30H.
1016    04 07             ADD  AL,07H     ;If lower nibble greater than 0AH, then add 37H.
1018    04 30     SKIP1:  ADD  AL,30H
101A    AA                STOSB           ;Store ASCII code of lower nibble in memory.

101B    8A C7             MOV  AL,BH      ;Get the data in AL-register.
101D    24 F0             AND  AL,0F0H    ;Mask the lower nibble.
101F    B1 04             MOV  CL,04H     ;Rotate upper nibble to lower nibble position.
1021    D2 C0             ROL  AL,CL
1023    3C 0A             CMP  AL,0AH     ;Check whether upper nibble is less than 0AH.
1025    7C 02             JL   SKIP2      ;If upper nibble less than 0AH, then add 30H.
1027    04 07             ADD  AL,07H     ;If upper nibble greater than 0AH, then add 37H.
1029    04 30      SKIP2: ADD  AL,30H
102B    AA                STOSB           ;Store ASCII code of upper nibble in memory.
102C    EB DB             JMP  NEXT       ;Jump to NEXT to convert next byte.
102E    F4         EXIT:  HLT             ;Halt program execution.

102F               CODE ENDS              ;End of code segment.
                   END                    ;Assembly end.
```

Sample Data

Input Data : $4E_H$
15_H
87_H
$C0_H$
20_H

Memory address	Content
1100	4E
1101	15
1102	87
1103	C0
1104	20

Output Data : 45
34
35
31
37
38
30
43

Memory address	Content
1400	45
1401	34
1402	35
1403	31
1404	37
1405	38
1406	30
1407	43

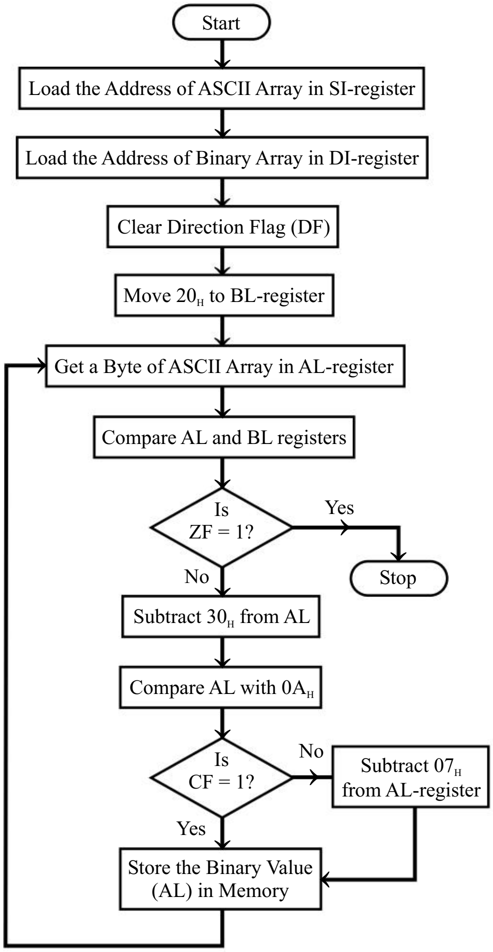
EXAMPLE PROGRAM 26 : ASCII to Binary Conversion

Write an assembly language program to convert an array of ASCII character to binary array.

Problem Analysis

The hexa digit 0 through 9 are represented by 30_H to 39_H in ASCII. Hence, if the ASCII code is in the range 30_H to 39_H, then we can subtract 30_H to get the binary value. The hexa digit A through F are represented by 41_H to 46_H in ASCII. Hence, if the ASCII code is in the range 41_H to 46_H then we can subtract 37_H to get the binary value.

Flowchart


Start
Load the Address of ASCII Array in SI-register
Load the Address of Binary Array in DI-register
Clear Direction Flag (DF)
Move 20H to BL-register
Get a Byte of ASCII Array in AL-register
Compare AL and BL registers
Is ZF = 1?
Yes
Stop
No
Subtract 30H from AL
Compare AL with 0AH
Is CF = 1?
No
Subtract 07H from AL-register
Yes
Store the Binary Value (AL) in Memory

Algorithm

1. Set SI as pointer for ASCII array.
2. Set DI as pointer for binary array.
3. Clear direction flag (DF)for autoincrement of pointers.
4. Move the end character 20_H to BL.
5. Get a byte of ASCII array in AL-register.

6. Compare AL and BL registers.
7. Check zero flag. If zero flag is set then go to step 14, otherwise go to next step.
8. Subtract 30_H from AL-register.
9. Compare AL with $0A_H$.
10. Check carry flag. If carry flag is set then go to step 12, otherwise go to next step.
11. Subtract 07_H from AL-register.
12. Store the binary value (AL) in memory.
13. Go to step 5.
14. Stop.

Assembly language program

```
;PROGRAM TO CONVERT AN ARRAY OF ASCII DATA TO BINARY DATA

CODE SEGMENT          ;Start of code segment.

       ASSUME CS:CODE ;Assembler directive.
       ORG  1000H     ;specify program starting address.

       MOV  SI,1100H  ;Set SI as pointer for ASCII array.
       MOV  DI,1400H  ;Set DI as pointer for binary array.
       CLD            ;Clear DF for autoincrement of SI and DI.
       MOV  BL,20H    ;Load the end character in BL-register.

NEXT:  LODSB          ;Load a byte of ASCII array in AL-register.
       CMP  AL,BL     ;Check for end of string.
       JE   EXIT      ;If zero flag is set, then go to EXIT.
       SUB  AL,30H    ;If zero flag is not set,
                      ;then subtract 30H from AL.
       CMP  AL,0AH    ;Check whether AL is greater than 0AH.
       JC   STORE     ;If AL is less than 0AH, then go to STORE.
       SUB  AL,07H    ;If AL is greater than or equal to 0AH,
                      ;then subtract 07H from AL.
STORE: STOSB          ;Store the binary value in memory.
       JMP  NEXT      ;Jump to convert next byte.

EXIT:  HLT            ;Halt program execution.

CODE ENDS             ;End of code segment.
END                   ;Assembly end.
```

Assembler listing for example program 26

```
;PROGRAM TO CONVERT AN ARRAY OF ASCII DATA TO BINARY DATA

0000            CODE SEGMENT          ;Start of code segment.

                      ASSUME CS:CODE  ;Assembler directive.
1000                  ORG  1000H      ;specify program starting address.
1000   BE 1100        MOV  SI,1100H   ;Set SI as pointer for ASCII array.
1003   BF 1400        MOV  DI,1400H   ;Set DI as pointer for binary array.
1006   FC             CLD             ;Clear DF for autoincrement of SI and DI.
1007   B3 20          MOV  BL,20H     ;Load the end character in BL-register.

1009   AC       NEXT: LODSB           ;Load a byte of ASCII array in AL-register.
100A   3A C3          CMP  AL,BL      ;Check for end of string.
100C   74 0B          JE   EXIT       ;If zero flag is set, then go to EXIT.
100E   2C 30          SUB  AL,30H     ;If zero flag is not set,
                                      ;then subtract 30H from AL.
1010   3C 0A          CMP  AL,0AH     ;Check whether AL is greater than 0AH.
1012   72 02          JC   STORE      ;If AL is less than 0AH, then go to STORE.
```

```
1014    2C 07           SUB  AL,07H     ;If AL is greater than or equal to 0AH,
                                        ;then subtract 07H from AL.
1016    AA      STORE: STOSB            ;Store the binary value in memory.
1017    EB F0           JMP  NEXT       ;Jump to convert next byte.

1019    F4      EXIT:  HLT              ;Halt program execution.

101A            CODE  ENDS              ;End of code segment.
                END                     ;Assembly end.
```

Sample Data

Input Data : 42 37 46 39 38 20

Memory address	Content
1100	42
1101	37
1102	46
1103	39
1104	38
1105	20

Output Data : 0B 07 0F 09 08

Memory address	Content
1400	0B
1401	07
1402	0F
1403	09
1404	08

EXAMPLE PROGRAM 27 : Program to Display the ASCII Code of the Key Pressed

Write an 8086 assembly language program to read the PC (Personal Computer) keyboard and display the ASCII code of the key pressed in the PC monitor.

Problem Analysis

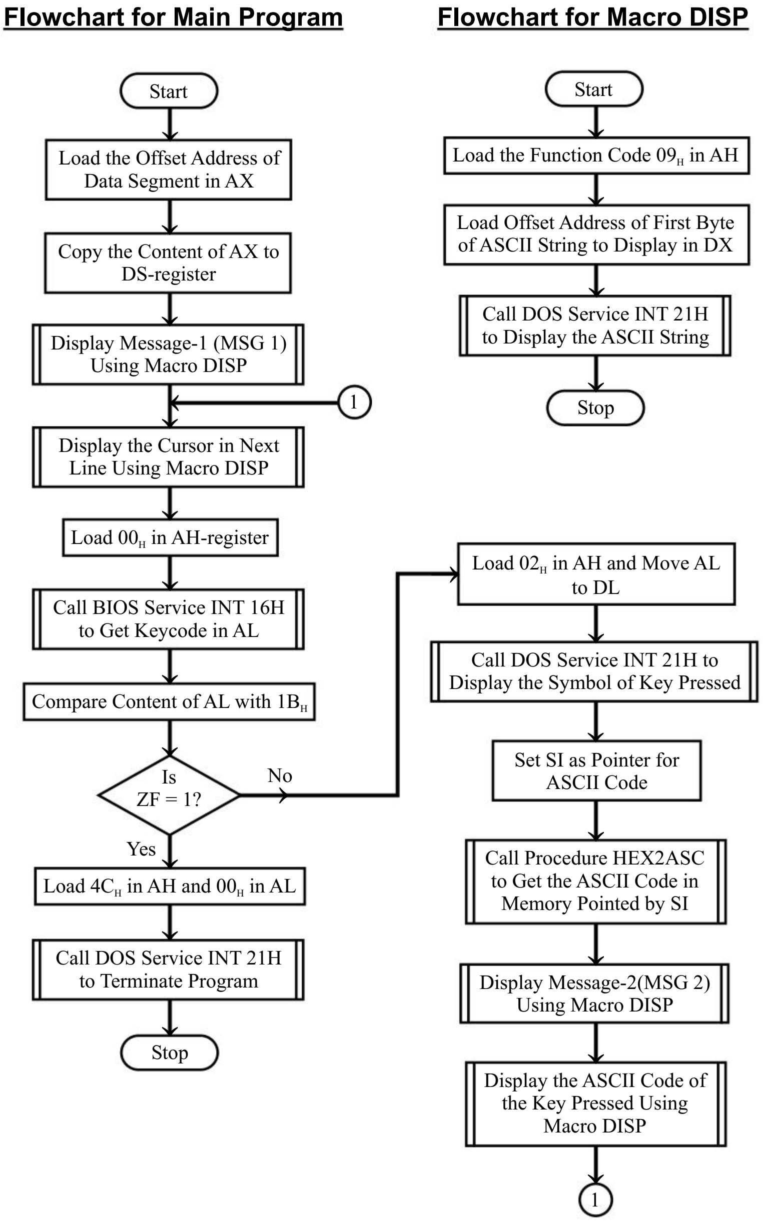
This assembly language program can be developed using BIOS and DOS interrupts. The BIOS interrupt INT 16H with function code 00_H can be used to read the ASCII code of the key pressed. Using the received ASCII value and the DOS interrupt INT 21H with function code 02_H the symbol/character representing the key which is being pressed can be displayed.

In order to display the ASCII value of the key, the ASCII value has to be considered as hexcode and then the ASCII value of each nibble of the hexcode has to be determined. A separate procedure can be written to convert the hexcode to ASCII. (The logic of converting hex to ASCII has been discussed in example program 25.) The ASCII values of lower and upper nibbles can be stored as ASCII string and then displayed on the PC monitor using the DOS interrupt INT 21H with function code 09_H.

Apart from displaying the key symbol and ASCII value, some message can also be displayed for the user. Since the program involves display of codes/messages a number of times, a macro can be written for display of ASCII string on PC the monitor.

One of the PC keys can be used to terminate the program. Here the ESC key is used to terminate the program. When the ESC key is pressed, the DOS interrupt INT 21H with function code $4C_H$ can be initiated to terminate the program and return the control to the command prompt.

Flowchart



Flowchart for Main Program

Flowchart for Macro DISP

Start
Load the Offset Address of Data Segment in AX
Copy the Content of AX to DS-register
Display Message-1 (MSG 1) Using Macro DISP
1
Display the Cursor in Next Line Using Macro DISP
Load 00$_H$ in AH-register
Call BIOS Service INT 16H to Get Keycode in AL
Compare Content of AL with 1B$_H$
Is ZF = 1?
No
Yes
Load 4C$_H$ in AH and 00$_H$ in AL
Call DOS Service INT 21H to Terminate Program
Stop
Start
Load the Function Code 09$_H$ in AH
Load Offset Address of First Byte of ASCII String to Display in DX
Call DOS Service INT 21H to Display the ASCII String
Stop
Load 02$_H$ in AH and Move AL to DL
Call DOS Service INT 21H to Display the Symbol of Key Pressed
Set SI as Pointer for ASCII Code
Call Procedure HEX2ASC to Get the ASCII Code in Memory Pointed by SI
Display Message-2(MSG 2) Using Macro DISP
Display the ASCII Code of the Key Pressed Using Macro DISP
1

Note : For flowchart of the procedure HEX2ASC please refer to Example program 25.

Algorithm

i) Macro DISP to display ASCII string

1. Load function code 09_H in AH-register.
2. Load the offset address of the first byte of ASCII string to be displayed in DX-register.
3. Initiate DOS interrupt INT 21H.

ii) Procedure for hex to ASCII conversion

For algorithm of this procedure please refer to Example program 25.

iii) Main program

1. Initialize DS-register.
2. Display the message "Press any key to test and ESC to exist" using macro DISP.
3. Move the cursor to next line using macro DISP.
4. Load the function code 00_H in AH-register and initiate BIOS interrupt INT 16H to get the ASCII value of key pressed in AL-register.
5. Compare the content of AL with ASCII value of ESC key to check whether ESC key is pressed. If ZF = 1, go to step 11, otherwise go to next step.
6. Load the function code 02_H in AH-register and load the ASCII value of key pressed in DL-register and then initiated DOS interrupt INT 21H to display the character/symbol of the key pressed.
7. Set SI as memory pointer for ASCII code.
8. Call procedure HEX2ASC to determine the ASCII codes.
9. Display the message "Normal ASCII character code = " using the macro display and then display the ASCII code using the same macro.
10. Go to step 3.
11. Load the function code $4C_H$ in AH-register and initiate the DOS interrupt INT 21H to terminate the program.
12. Stop.

Assembly language program

```
;PROGRAM TO DISPLAY THE ASCII CODE OF THE KEY PRESSED

;user defined macro to display the string

DISP MACRO MSG              ;Start of macro DISP.

        MOV  AH, 09H        ;Move the function code in AH.
        MOV  DX,OFFSET MSG  ;Load the address of message in DX.
        INT  21H            ;Call DOS service for display.

ENDM                        ;End of macro DISP.

CODE SEGMENT                ;Start of code segment.

        ASSUME CS:CODE      ;Assembler directive.
        ASSUME DS:DATA      ;Assembler directive.

        MOV  AX,DATA        ;Initialize DS to DATA.
        MOV  DS,AX
        DISP MSG1           ;Display MSG1 using macro DISP.

RDKEY:  DISP NL             ;Display cursor in next line.
        MOV  AH,00H         ;Move the function code in AH.
        INT  16H            ;Call BIOS service to read key code.

        CMP  AL,ESC         ;Check whether ESC key is pressed.
        JE   OVER           ;If yes, jump to OVER, otherwise write
                            ;character to standard output device.
        MOV  AH,02H         ;Move the function code in AH.
        MOV  DL,AL          ;Load hexcode of key pressed in DL.
        INT  21H            ;Call DOS service for display.
```

```
         MOV  SI,OFFSET ASCI  ;Get address for saving ASCII code.
         CALL HEX2ASC         ;Call procedure hex to ASCII conversion.
         DISP MSG2            ;Display MSG2 using macro DISP.
         DISP ASCI            ;Display ASCII code using macro DISP.
         JMP  RDKEY           ;Jump to RDKEY to read next key code.

OVER:    MOV  AH,4CH          ;Move the function code in AH.
         MOV  AL,00H
         INT  21H             ;Return to command prompt.

HEX2ASC PROC NEAR             ;Start of procedure HEX2ASC.

         MOV  BL,AL           ;Save the key code in BL-register.
         AND  AL,0F0H         ;Mask the lower nibble of key code.
         MOV  CL,04H          ;Rotate upper nibble to
         ROL  AL,CL           ;lower nibble position.
         MOV  DL,0AH          ;Check whether upper nibble is
         CMP  AL,DL           ;less than 0AH.
         JL   SKIP1           ;If upper nibble is less than 0AH,
         ADD  AL,07H          ;then add 30H or if upper nibble is
SKIP1:   ADD  AL,30H          ;greater than 0AH, then add 37H.
         MOV  [SI],AL         ;Store ASCII code of upper nibble.

         MOV  AL,BL           ;Get the key code in AL-register.
         AND  AL,0FH          ;Mask the upper nibble.
         CMP  AL,DL           ;Check whether lower nibble is less than 0AH.
         JL   SKIP2           ;If lower nibble is less than 0AH,
         ADD  AL,07H          ;then add 30H or if lower nibble is
SKIP2:   ADD  AL,30H          ;greater than 0AH, then add 37H.
         MOV  [SI+1],AL       ;Store ASCII code of lower nibble.
         MOV  AL,'$'          ;Append end of string.
         MOV  [SI+2],AL
         RET                  ;Return to main program.

HEX2ASC ENDP                  ;End of procedure HEX2ASC.

CODE ENDS                     ;End of code segment.

DATA SEGMENT                  ;start of data segment.


         CR   EQU 0DH         ;ASCII for carriage return.
         LF   EQU 0AH         ;ASCII for line feed.
         ESC  EQU 1BH         ;ASCII for escape.
         ASCI DB 8 DUP(0)     ;Assembler directive.

         MSG1 DB   'Press any key to test and esc to exit ','$'.
         MSG2 DB   ', Normal ASCII character code = ','$'.
         NL   DB   CR,LF,'$'.

DATA ENDS                     ;End of data segment.
END                           ;Assembly end.
```

Assembler listing for example program 27

```
                              ;PROGRAM TO DISPLAY THE ASCII CODE OF THE KEY PRESSED

                              ;user defined macro to display the string

                              DISP MACRO MSG              ;Start of macro DISP.

                                    MOV   AH, 09H         ;Move the function code in AH.
                                    MOV   DX,OFFSET MSG   ;Load the address of message in DX.
                                    INT   21H             ;Call DOS service for display.
```

```
                       ENDM                   ;End of macro DISP.

0000                   CODE SEGMENT           ;Start of code segment.

                          ASSUME CS:CODE      ;Assembler directive.
                          ASSUME DS:DATA      ;Assembler directive.

0000  B8 – R              MOV  AX,DATA        ;Initialize DS to DATA.
0003  8E D8               MOV  DS,AX
                          DISPMSG1            ;Display MSG1 using macro DISP.
0005  B4 09     +         MOV  AH, 09H        ;Move the function code in AH.
0007  BA 0008 R +         MOV  DX,OFFSET MSG1 ;Load the address of message in DX.
000A  CD 21     +         INT  21H            ;Call DOS service for display.

000C              RDKEY:  DISP NL             ;Display cursor in next line.
000C  B4 09     +         MOV  AH, 09H        ;Move the function code in AH.
000E  BA 0050 R +         MOV  DX,OFFSET NL   ;Load the address of message in DX.
0011  CD 21     +         INT  21H            ;Call DOS service for display.
0013  B4 00               MOV  AH,00H         ;Move the function code in AH.
0015  CD 16               INT  16H            ;Call BIOS service to read key code.

0017  3C 1B 90 90         CMP  AL,ESC         ;Check whether ESC key is pressed.
001B  74 1C               JE   OVER           ;If yes, jump to OVER, otherwise write
                                              ;character to standard output device.
001D  B4 02               MOV  AH,02H         ;Move the function code in AH.
001F  8A D0               MOV  DL,AL          ;Load hexcode of key pressed in DL.
0021  CD 21               INT  21H            ;Call DOS service for display.

0023  BE 0000 R           MOV  SI,OFFSET ASCI ;Get address for saving ASCII code.
0026  E8 003F R           CALL HEX2ASC        ;Call procedure hex to ASCII conversion.
                          DISP MSG2           ;Display MSG2 using macro DISP.
0029  B4 09     +         MOV  AH, 09H        ;Move the function code in AH.
002B  BA 002F R +         MOV  DX,OFFSET MSG2 ;Load the address of message in DX.
002E  CD 21     +         INT  21H            ;Call DOS service for display.
                          DISP ASCI           ;Display ASCII code using macro DISP.
0030  B4 09     +         MOV  AH, 09H        ;Move the function code in AH.
0032  BA 0000 R +         MOV  DX,OFFSET ASCI ;Load the address of message in DX.
0035  CD 21     +         INT  21H            ;Call DOS service for display.
0037  EB D3               JMP  RDKEY          ;Jump to RDKEY to read next key code.

0039  B4 4C       OVER:   MOV  AH,4CH         ;Move the function code in AH.
003B  B0 00               MOV  AL,00H
003D  CD 21               INT  21H            ;Return to command prompt.

003F               HEX2ASCPROC  NEAR          ;Start of procedure HEX2ASC.

003F  8A D8               MOV  BL,AL          ;Save the key code in BL-register.
0041  24 F0               AND  AL,0F0H        ;Mask the lower nibble of key code.
0043  B1 04               MOV  CL,04H         ;Rotate upper nibble to
0045  D2 C0               ROL  AL,CL          ;lower nibble position.
0047  B2 0A               MOV  DL,0AH         ;Check whether upper nibble is
0049  3A C2               CMP  AL,DL          ;less than 0AH.
004B  7C 02               JL   SKIP1          ;If upper nibble is less than 0AH,
004D  04 07               ADD  AL,07H         ;then add 30H or if upper nibble is
004F  04 30       SKIP1:  ADD  AL,30H         ;greater than 0AH, then add 37H.
0051  88 04               MOV  [SI],AL        ;Store ASCII code of upper nibble.

0053  8A C3               MOV  AL,BL          ;Get the key code in AL-register.
0055  24 0F               AND  AL,0FH         ;Mask the upper nibble.
0057  3A C2               CMP  AL,DL          ;Check lower nibble is less than 0AH.
0059  7C 02               JL   SKIP2          ;If lower nibble is less than 0AH,
005B  04 07               ADD  AL,07H         ;then add 30H or if lower nibble is
```

```
005D  04 30         SKIP2: ADD   AL,30H         ;greater than 0AH, then add 37H.
005F  88 44 01             MOV   [SI+1],AL      ;Store ASCII code of lower nibble.
0062  B0 24                MOV   AL,'$'         ;Append end of string.
0064  88 44 02             MOV   [SI+2],AL
0067  C3                   RET                  ;Return to main program.

0068                HEX2ASC ENDP                ;End of procedure HEX2ASC.

0068                CODE ENDS                   ;End of code segment.

0000                DATA SEGMENT                ;start of data segment.

= 000D                    CR        EQU 0DH     ;ASCII for carriage return.
= 000A                    LF        EQU 0AH     ;ASCII for line feed.
= 001B                    ESC       EQU 1BH     ;ASCII for escape.
0000   08  [              ASCI      DB 8 DUP(0) ;Assembler directive.
              00
                  ]
0008  50 72 65 73 73 20  MSG1   DB  'Press any key to test and esc to exit ','$'.
      61 6E 79 20 6B 65
      79 20 74 6F 20 74
      65 73 74 20 61 6E
      64 20 65 73 63 20
      74 6F 20 65 78 69
      74 20 24
002F  2C 20 4E 6F 72 6D  MSG2    DB  ', Normal ASCII character code = ','$'.
      61 6C 20 41 53 43
      49 49 20 63 68 61
      72 61 63 74 65 72
      20 63 6F 64 65 20
      3D 20 24
0050  0D 0A 24             NL DB CR,LF,'$'

0053                     DATA ENDS              ;End of data segment.
                         END                    ;Assembly end.
```

EXAMPLE PROGRAM 28 : Program to Find the Length of a String

Write an 8086 assembly language program to determine the length of ASCII string. Use PC keyboard to input the string and display the length on PC monitor.

Problem Analysis

The ASCII string is input through PC keyboard and so DOS interrupt INT 21H with function code 01_H can be used to read the key code. On counting the number of input characters the length can be determined. The carriage return (Enter key) can be used to terminate the input string. The input string can also be displayed on PC monitor.

The count value representing the length of string will be in hex. In order to display the count value in PC monitor it has to be converted to decimal and ASCII value of each decimal digit has to be determined and stored in memory as ASCII string. Then the DOS interrupt INT 21H with function code 09_H can be used to display the length in decimal.

A separate procedure can be written to convert the count value in hex to decimal and then to ASCII. In this procedure the hex value is divided by ten ($0A_H$). Now the quotient is ten's digit and remainder is unit's digit. (Here it is assumed that the count does not exceeds 99_{10}.) Then the ASCII value of zero can be added to ten's digit and unit's digit to get their respective ASCII values.

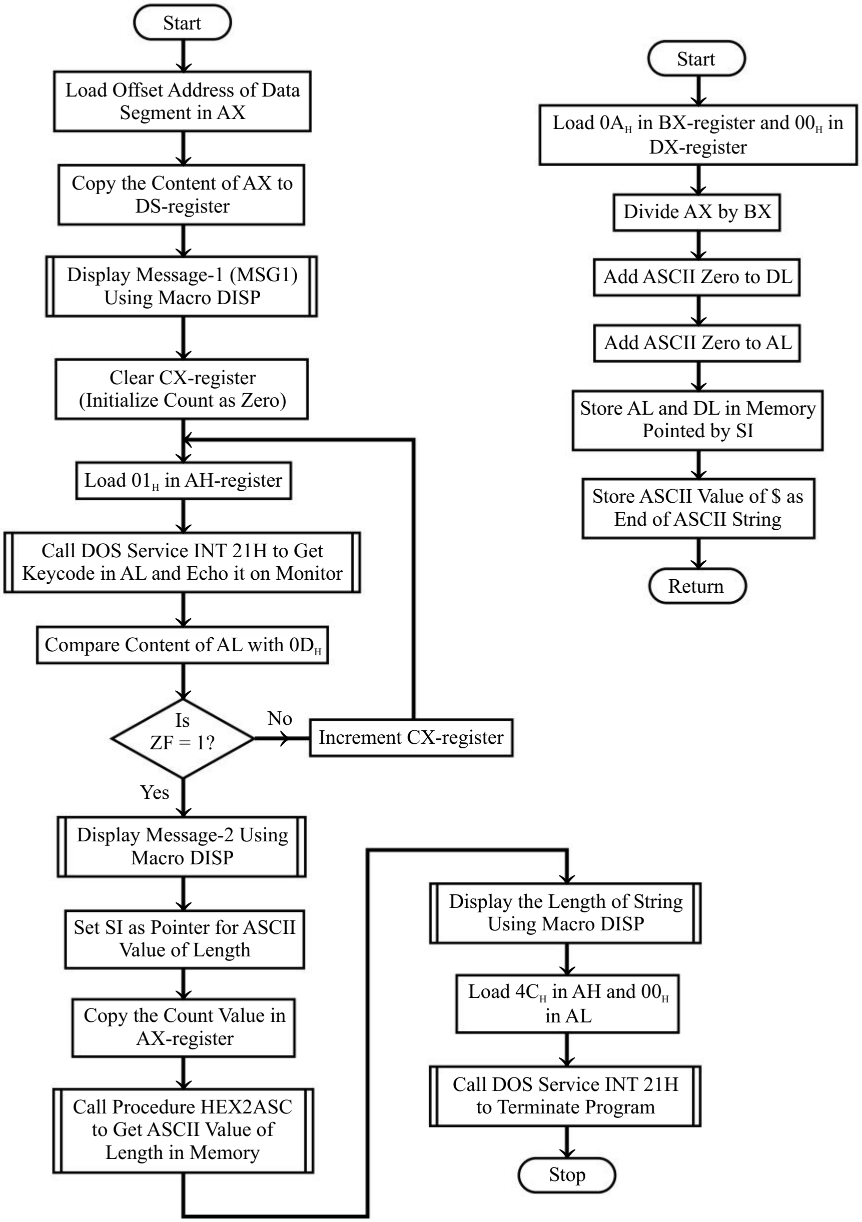
Apart from displaying the input string and its length some message can also be displayed for the user. Since the program involves display of messages a number of times, a macro can be written for display of ASCII string on PC monitor.

After displaying the length of string the DOS interrupt INT 21H with function code $4C_H$ can be initiated to terminate the program execution and return the control to the command prompt.

Flowchart

Flowchart for Main Program

Flowchart for Procedure HEX2ASC


Start
Load Offset Address of Data Segment in AX
Copy the Content of AX to DS-register
Display Message-1 (MSG1) Using Macro DISP
Clear CX-register (Initialize Count as Zero)
Load 01_H in AH-register
Call DOS Service INT 21H to Get Keycode in AL and Echo it on Monitor
Compare Content of AL with $0D_H$
Is ZF = 1?
No
Increment CX-register
Yes
Display Message-2 Using Macro DISP
Set SI as Pointer for ASCII Value of Length
Copy the Count Value in AX-register
Call Procedure HEX2ASC to Get ASCII Value of Length in Memory
Display the Length of String Using Macro DISP
Load $4C_H$ in AH and 00_H in AL
Call DOS Service INT 21H to Terminate Program
Stop
Start
Load $0A_H$ in BX-register and 00_H in DX-register
Divide AX by BX
Add ASCII Zero to DL
Add ASCII Zero to AL
Store AL and DL in Memory Pointed by SI
Store ASCII Value of $ as End of ASCII String
Return

Algorithm

i) Main program

1. Initialize DS-register
2. Display the message "ENTER THE STRING.INPUT = " using macro DISP.
3. Let CX-register be used as counter. Initialize count as zero.
4. Load the function code 01_H in AH-register and initiate DOS interrupt INT 21H to get the ASCII value of key pressed in AL and echo (display) the key symbol/character in the PC monitor.
5. Compare the content of AL with ASCII value of ENTER key to check whether ENTER key is pressed. If ZF =1, go to step 8, otherwise go to next step.
6. Increment the counter (CX-register).
7. Go to step 4.
8. Display the message " THE LENGTH OF STRING = " using the macro DISP.
9. Set SI as memory pointer for ASCII value of length.
10. Copy the count value in AX-register.
11. Call procedure HEX2ASC to get the ASCII value of count in memory pointed by SI.
12. Display the length of string using the macro DISP.
13. Load the function code $4C_H$ in AH-register and initiate the DOS interrupt INT 21H to terminate the program.
14. Stop.

ii) Procedure HEX2ASC

1. Load the divisor ($0A_H$) in BX-register.
2. Divide the hex value of length in AX with $0A_H$ to get units in DX and tens in AX.
3. Add the ASCII value of zero to DL-register to get ASCII value of units.
4. Add the ASCII value of zero to AL-register to get ASCII value of tens.
5. Store the ASCII values in memory pointed by SI.
6. Return.

Note : Refer to Example program 27 for algorithm of macro DISP.

Assembly language program

```
;PROGRAM TO FIND THE LENGTH OF STRING
;User defined macro to display message

DISP MACRO MSG                ;Start of macro.

        MOV  AH, 09H          ;Move function code in AH.
        MOV  DX,OFFSET MSG    ;Get address of string to display in DX.
        INT  21H              ;Call DOS service for display.

ENDM                          ;End of macro.

DATA SEGMENT                  ;Start of data segment.

        CR   EQU 0DH          ;ASCII for carriage return.
        LF   EQU 0AH          ;ASCII for line feed.
        LEN  DB 04 DUP(0)     ;Assembler directive.
        MSG1 DB  'ENTER THE STRING.INPUT = ','$'.
        MSG2 DB CR,LF,  'THE LENGTH OF STRING = ','$'.

DATA ENDS                     ;End of data segment.

CODE SEGMENT                  ;Start of code segment.

        ASSUME CS:CODE        ;Assembler directive.
        ASSUME DS:DATA        ;Assembler directive.

        MOV  AX,DATA          ;Initialize DS to data.
        MOV  DS,AX
        DISP MSG1             ;Display MSG1 using macro DISP.
        MOV  CX,00H          ;Initialize count to zero.

RDKEY:  MOV  AH,01H          ;Move function code in AH.
        INT  21H             ;Call DOS service to read keycode.

        CMP  AL,CR           ;Compare keycode with carriage return.
        JE    AHEAD          ;If key is carriage return, go to AHEAD.

        INC  CX              ;Increment the counter.
        JMP  RDKEY           ;Jump to RDKEY to get next keycode.

AHEAD:  DISP MSG2            ;Display MSG2 using macro disp.
        MOV  SI,OFFSET LEN   ;Get the address for saving the length.
        MOV  AX,CX           ;Get the count in AX-register.
        CALL HEX2ASC         ;Call procedure HEX2ASC.
        DISP LEN             ;Display length of string using macro.

        MOV  AH,4CH          ;Move function code in AH.
        MOV  AL,00H
        INT  21H             ;Return to command prompt.

HEX2ASC PROC NEAR            ;Start of procedure HEX2ASC.

        MOV  BX,0AH          ;Load the divisor in BX-register.
        MOV  DX,00           ;Clear DX-register.
        DIV  BX              ;Divide the count value by 0AH to get
                             ;unit digit in DL and ten’s digit in AL.
        ADD  DL,’0'          ;Convert the unit digit to ASCII.
        ADD  AL,’0'          ;Convert the ten’s digit to ASCII.
        MOV  [SI],AL         ;Store the ASCII values in memory.
        MOV  [SI+1],DL
        MOV  AL,’$’          ;Append end of string.
        MOV  [SI+2],AL
        RET

HEX2ASC ENDP                 ;End of procedure HEX2ASC.

CODE ENDS                    ;End of code segment.
END                          ;Assembly end.
```

Assembler listing for example program 28

```
                        ;PROGRAM TO FIND THE LENGTH OF STRING
                        ;User defined macro to display message

                        DISP MACRO MSG             ;Start of macro.

                              MOV  AH, 09H         ;Move function code in AH.
                              MOV  DX,OFFSET MSG   ;Get address of string to display in DX.
                              INT  21H             ;Call DOS service for display.
                        ENDM                       ;End of macro.

0000                    DATA SEGMENT               ;start of data segment.

= 000D                        CR   EQU 0DH         ;ASCII for carriage return.
= 000A                        LF   EQU 0AH         ;ASCII for line feed.
0000      04  [               LEN  DB 04 DUP(0)    ;Assembler directive.
                00
                   ]
0004 45 4E 54 45 52 20        MSG1 DB  ‘ENTER THE STRING.INPUT = ‘,’$’.
     54 48 45 20 53 54
     52 49 4E 47 2E 49
     4E 50 55 54 20 3D
     20 24
```

```
001E 0D 0A 54 48 45 20        MSG2    DB CR,LF,  'THE LENGTH OF STRING = ‘,’$’.
     4C 45 4E 47 54 48
     20 4F 46 20 53 54
     52 49 4E 47 20 3D
     20 24

0038               DATA ENDS                     ;End of data segment.

0000               CODE SEGMENT                  ;Start of code segment.

                           ASSUME CS:CODE        ;Assembler directive.
                           ASSUME DS:DATA        ;Assembler directive.

0000  B8 –    R            MOV  AX,DATA          ;Initialize DS to data.
0003  8E D8                MOV  DS,AX
                           DISP MSG1             ;Display MSG1 using macro DISP.
0005  B4 09      +         MOV  AH, 09H          ;Move function code in AH.
0007  BA 0004 R  +         MOV  DX,OFFSET MSG1   ;Get address of string to display in DX.
000A  CD 21      +         INT  21H              ;Call DOS service for display.
000C  B9 0000              MOV  CX,00H           ;Initialize count to zero.

000F  B4 01        RDKEY:  MOV  AH,01H           ;Move function code in AH.
0011  CD 21                INT  21H              ;Call DOS service to read keycode.

0013  3C 0D                CMP  AL,CR            ;Compare keycode with carriage return.
0015  74 03                JE   AHEAD            ;If key is carriage return, go to AHEAD.

0017  41                   INC  CX               ;Increment the counter.
0018  EB F5                JMP  RDKEY            ;Jump to RDKEY to get next keycode.

001A               AHEAD:  DISP MSG2             ;Display MSG2 using macro disp.
001A  B4 09      +         MOV  AH, 09H          ;Move function code in AH.
001C  BA 001E  R +         MOV  DX,OFFSET MSG2   ;Get address of string to display in DX.
001F  CD 21      +         INT  21H              ;Call DOS service for display.
0021  BE 0000  R           MOV  SI,OFFSET LEN    ;Get the address for saving the length.
0024  8B C1                MOV  AX,CX            ;Get the count in AX-register.
0026  E8 0036 R            CALL HEX2ASC          ;Call procedure HEX2ASC.
                           DISP LEN              ;Display length of string using macro.
0029  B4 09     +          MOV  AH, 09H          ;Move function code in AH.
002B  BA 0000  R +         MOV  DX,OFFSET LEN    ;Get address of string to display in DX.
002E  CD 21     +          INT  21H              ;Call DOS service for display.

0030  B4 4C                MOV  AH,4CH           ;Move function code in AH.
0032  B0 00                MOV  AL,00H
0034  CD 21                INT  21H              ;Return to command prompt.

0036                HEX2ASC  PROC NEAR           ;Start of procedure HEX2ASC.

0036  BB 000A              MOV  BX,0AH           ;Load the divisor in BX-register.
0039  BA 0000              MOV  DX,00            ;Clear DX-register.
003C  F7 F3                DIV  BX               ;Divide the count value by 0AH to get
                                                 ;unit digit in DL and ten’s digit in AL.
003E  80 C2 30             ADD  DL,'0'           ;Convert the unit's digit to ASCII.
0041  04 30                ADD  AL,'0'           ;Convert the ten’s digit to ASCII.
0043  88 04                MOV  [SI],AL          ;Store the ASCII values in memory.
0045  88 54 01             MOV  [SI+1],DL
0048  B0 24                MOV  AL,’$’           ;Append end of string.
004A  88 44 02             MOV  [SI+2],AL
004D  C3                   RET
004E                HEX2ASC ENDP                 ;End of procedure HEX2ASC.

004E                CODE ENDS                    ;End of code segment.
                    END                          ;Assembly end.
```

EXAMPLE PROGRAM 29 : Program to Find Palindrome

Write an 8086 assembly language program to verify whether an input string is a palindrome or not. Use PC keyboard to input the string and display the result on the PC monitor.

Problem Analysis

The input ASCII string can be read using DOS interrupt INT 21H with function code 01_H and stored in memory. Then the input string can be arranged in the reverse order to get reversed string and store the reversed string in another memory location.

Compare the input string and reversed string byte by byte to verify whether it is a palindrome. If the input string is byte by byte same as reversed string then it is a palindrome, otherwise it is not a palindrome.

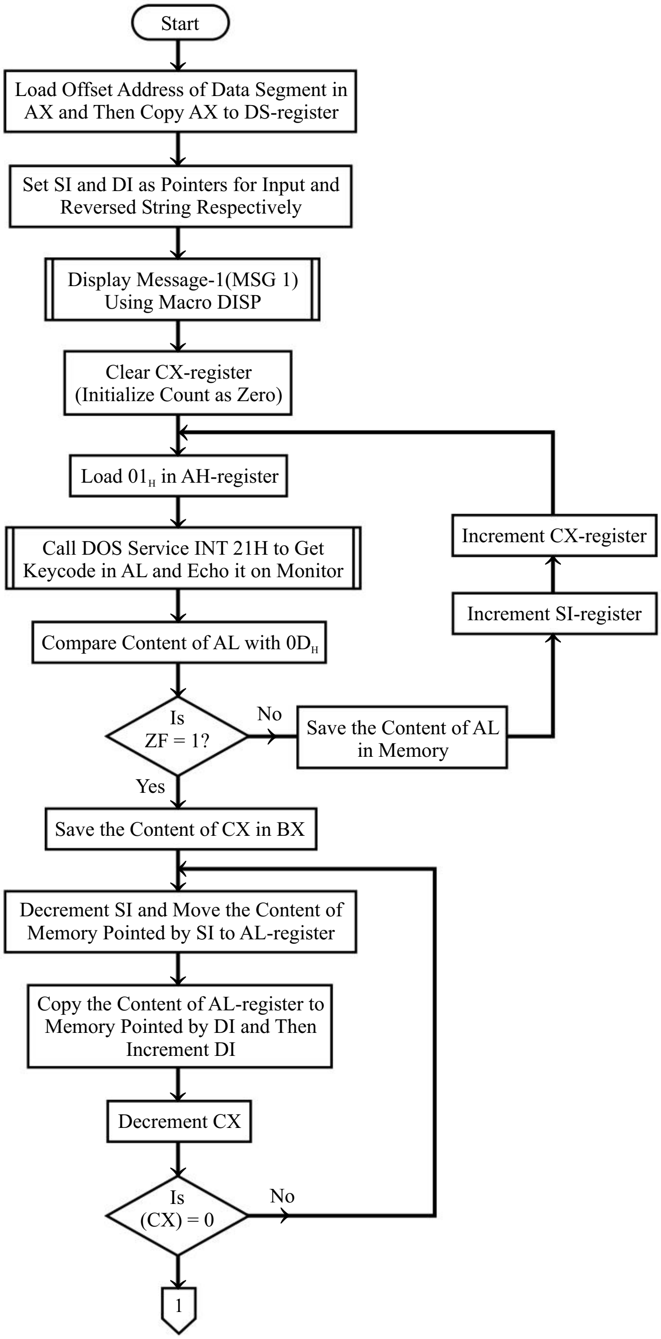
For the convenience of the user, the input string and reversed string can also be displayed on the PC monitor. The result can be displayed as a message on the PC monitor. Since the program involves display of messages a number of times, a macro can be written for display of ASCII string on the PC monitor. After displaying the result the DOS interrupt INT 21H with function code $4C_H$ can be initiated to terminate the program execution and return the control to the command prompt.

Algorithm

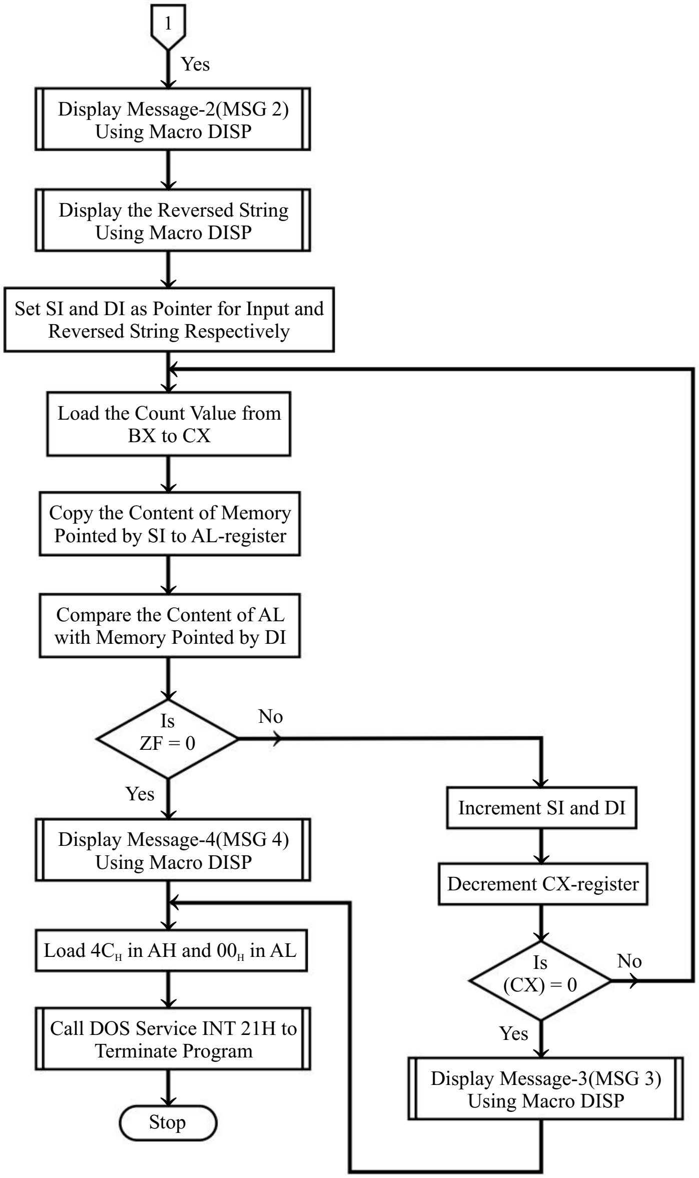
1. Initialize DS-register.
2. Set SI as pointer for input string and DI as pointer for reversed string.
3. Display the message "ENTER THE STRING. INPUT STRING = " using the macro DISP.
4. Let CX-register be used as counter for number of bytes in the input string. Initialize count as zero.
5. Load the function code 01_H in AH-register and initiate DOS interrupt INT 21H to get the ASCII value of key pressed in AL and echo (display) the key symbol/character in the PC monitor.
6. Compare the content of AL with ASCII value of ENTER key to check whether ENTER key is pressed. If ZF = 1, go to step 10, otherwise go to next step.
7. Save the content of AL-register in memory pointed by SI, and increment SI-register.
8. Increment the counter (CX-register).
9. Go to step 5.
10. Save the count value in BX-register.
11. Decrement SI register and copy a byte of input string in AL-register.
12. Copy the content of AL-register to the memory pointed by DI and increment DI-register.
13. Decrement CX-register. Check whether content of CX is zero and if it is true, go to next step, otherwise go to step 11.
14. Display the message "REVERSE OF STRING = " using the macro DISP.
15. Display the reversed string using the macro DISP.
16. Again set SI as pointer for input string and DI as pointer for reversed string.
17. Load the count value from BX-register to CX-register.
18. Get a byte of input string in AL and compare with corresponding byte of reverse string in memory.
19. Check zero flag. If ZF = 0, then go to step 24, otherwise go to next step.
20. Increment SI and DI-register.
21. Decrement CX-register. Check whether content of CX is zero and if it is true, go to next step, otherwise go to step 18.
22. Display the message "INPUT STRING IS A PALINDROME" using macro DISP.
23. Go to step 25.
24. Display the message "INPUT STRING IS NOT A PALINDROME" using macro DISP.
25. Load function code $4C_H$ in AH-register and initiate the DOS interrupt INT 21H to terminate the program.
26. Stop.

Note : For algorithm and flowchart of macro DISP refer to Example program 27.

Flowchart


Start
Load Offset Address of Data Segment in AX and Then Copy AX to DS-register
Set SI and DI as Pointers for Input and Reversed String Respectively
Display Message-1(MSG 1) Using Macro DISP
Clear CX-register (Initialize Count as Zero)
Load 01$_H$ in AH-register
Call DOS Service INT 21H to Get Keycode in AL and Echo it on Monitor
Compare Content of AL with 0D$_H$
Is ZF = 1?
No
Save the Content of AL in Memory
Increment SI-register
Increment CX-register
Yes
Save the Content of CX in BX
Decrement SI and Move the Content of Memory Pointed by SI to AL-register
Copy the Content of AL-register to Memory Pointed by DI and Then Increment DI
Decrement CX
Is (CX) = 0
No
1

Flowchart continued ....


1
Yes
Display Message-2(MSG 2) Using Macro DISP
Display the Reversed String Using Macro DISP
Set SI and DI as Pointer for Input and Reversed String Respectively
Load the Count Value from BX to CX
Copy the Content of Memory Pointed by SI to AL-register
Compare the Content of AL with Memory Pointed by DI
Is ZF = 0
No
Yes
Increment SI and DI
Display Message-4(MSG 4) Using Macro DISP
Decrement CX-register
Load 4C$_H$ in AH and 00$_H$ in AL
Is (CX) = 0
No
Yes
Call DOS Service INT 21H to Terminate Program
Display Message-3(MSG 3) Using Macro DISP
Stop

Assembly language program

```
;PROGRAM TO FIND PALINDROME
;Macro to display string
DISP MACRO MSG              ;Start of macro DISP.
        MOV  AH,09H         ;Move function code to AH-register.
        MOV  DX,OFFSET MSG  ;Move address of string to display in DX.
        INT  21H            ;Call DOS service for display.
ENDM                        ;End of macro DISP.
;Define code segment
CODE SEGMENT                ;Start of code segment.

        ASSUME CS:CODE      ;Assembler directive.
        ASSUME DS:DATA      ;Assembler directive.
        MOV  AX,DATA        ;Initialize DS to DATA.
        MOV  DS,AX
        MOV  SI,OFFSET INS  ;Load address of input string in SI.
        MOV  DI,OFFSET RES  ;Load address of reverse string in DI.
        DISP MSG1           ;Display MSG1 using macro DISP.
        MOV  CX,00H         ;Initialize the count.
RDCHAR: MOV  AH,01H         ;Move the function code in AH.
        INT  21H            ;Call DOS service to read keyboard.
        CMP  AL,CR          ;Check for carriage return.
        JE   AHEAD          ;If key entered is carriage return,
                            ;then go to AHEAD.
        MOV  [SI],AL        ;Save character in memory pointed by SI.
        INC  SI             ;Increment the pointer.
        INC  CX             ;Increment the counter.
        JMP  RDCHAR         ;Jump to RDCHAR to get next character.
AHEAD:  MOV  BX,CX          ;Save the count in BX.
REVERS: DEC  SI             ;Copy characters from memory pointed
        MOV  AL,[SI]        ;by SI to memory pointed by DI
        MOV  [DI],AL        ;in reverse order.
        INC  DI
        LOOP REVERS
        MOV  AL,’$’         ;Put end of string.
        MOV  [DI],AL
        DISP MSG2           ;Display the MSG2 using macro DISP.
        DISP RES            ;Display reversed string using macro DISP.
        MOV  SI,OFFSET INS  ;Get address of input string in SI.
        MOV  DI,OFFSET RES  ;Get address of reversed string in DI.
        MOV  CX,BX          ;Initialize the counter.
CHECK:  MOV  AL,[SI]        ;Get a character of input string in AL.
        CMP  AL,[DI]        ;Compare with character of reversed string.
        JNE  FALSE          ;If not equal jump to FALSE.
        INC  SI             ;Increment the pointers.
        INC  DI
        LOOP CHECK          ;Repeat comparison until counter expires.
        DISP MSG3           ;Display MSG3 using macro DISP.
        JMP  EXIT
FALSE:  DISP MSG4           ;Display MSG4 using macro DISP.
EXIT:   MOV  AH,4CH         ;Move function code to AH-register.
        MOV  AL,00H
        INT  21H            ;Return to command prompt.
CODE ENDS                   ;End of code segment.
;Define data segment
DATA SEGMENT                ;start of data segment.
        CR   EQU 0DH        ;ASCII code for carriage return.
        LF   EQU 0AH        ;ASCII code for line feed.
        INS  DB 40 DUP(0)   ;Reserve 40 locations for input string.
        RES  DB 40 DUP(0)   ;Reserve 40 locations for reverse string.
        MSG1 DB‘ENTER THE STRING.INPUT STRING= ‘,’$’.
        MSG2 DB CR,LF,’REVERSE OF STRING = ‘,’$’.
        MSG3 DB CR,LF,’INPUT STRING IS A PALINDROME’,’$’.
        MSG4 DB CR,LF,’INPUT STRING IS NOT A PALINDROME’,’$’.
DATA ENDS                  ;End of data segment.
END                        ;Assembly end.
```

Assembler listing for example program 29

```
                ;PROGRAM TO FIND PALINDROME
                        ;Macro to display string

                         DISP MACRO MSG        ;Start of macro DISP.

                          MOV  AH,09H          ;Move function code to AH-register.
                          MOV  DX,OFFSET MSG   ;Move address of string to display in DX.
                          INT  21H             ;Call DOS service for display.
                          ENDM                 ;End of macro DISP.
                ;Define code segment
0000            CODE SEGMENT                   ;Start of code segment.
                          ASSUME CS:CODE       ;Assembler directive.
                          ASSUME DS:DATA       ;Assembler directive.
0000 B8 –  R              MOV  AX,DATA         ;Initialize DS to DATA.
0003 8E D8                MOV  DS,AX
0005 BE 0000 R            MOV  SI,OFFSET INS   ;Load address of input string in SI.
0008 BF 0028 R            MOV  DI,OFFSET RES   ;Load address of reverse string in DI.
                          DISP MSG1            ;Display MSG1 using macro DISP.
000B B4 09     +          MOV  AH,09H          ;Move function code to AH-register.
000D BA 0050 R +          MOV  DX,OFFSET MSG1  ;Move address of string to display in DX.
0010 CD 21     +          INT  21H             ;Call DOS service for display.
0012 B9 0000              MOV  CX,00H          ;Initialize the count.
0015 B4 01       RDCHAR:  MOV  AH,01H          ;Move the function code in AH.
0017 CD 21                INT  21H             ;Call DOS service to read keyboard.
0019 3C 0D 90 90          CMP  AL,CR           ;Check for carriage return.
001D 74 06                JE   AHEAD           ;If key entered is carriage return,
                                               ;then go to AHEAD.
001F 88 04                MOV  [SI],AL         ;Save character in memory pointed by SI.
0021 46                   INC  SI              ;Increment the pointer.
0022 41                   INC  CX              ;Increment the counter.
0023 EB F0                JMP  RDCHAR          ;Jump to RDCHAR to get next character.
0025 8B D9       AHEAD:   MOV  BX,CX           ;Save the count in BX.
0027 4E          REVERS:  DEC  SI              ;Copy characters from memory pointed
0028 8A 04                MOV  AL,[SI]         ;by SI to memory pointed by DI
002A 88 05                MOV  [DI],AL         ;in reverse order.
002C 47                   INC  DI
002D E2 F8                LOOP REVERS
002F B0 24                MOV  AL,’$’          ;Put end of string.
0031 88 05                MOV  [DI],AL
                          DISP MSG2            ;Display the MSG2 using macro DISP.
0033 B4 09     +          MOV  AH,09H          ;Move function code to AH-register.
0035 BA 0063 R +          MOV  DX,OFFSET MSG2  ;Move address of string to display in DX.
0038 CD 21     +          INT  21H             ;Call DOS service for display.
                          DISP RES             ;Display reversed string using macro DISP.
003A B4 09     +          MOV  AH,09H          ;Move function code to AH-register.
003C BA 0028 R +          MOV  DX,OFFSET RES   ;Move address of string to display in DX.
003F CD 21     +          INT  21H             ;Call DOS service for display.
0041 BE 0000 R            MOV  SI,OFFSET INS   ;Get address of input string in SI.
0044 BF 0028 R            MOV  DI,OFFSET RES   ;Get address of reversed string in DI.
0047 8B CB                MOV  CX,BX           ;Initialize the counter.
0049 8A 04         CHECK: MOV  AL,[SI]         ;Get a character of input string in AL.
004B 3A 05                CMP  AL,[DI]         ;Compare with character of reversed string .
004D 75 0E                JNE  FALSE           ;If not equal jump to FALSE.
004F 46                   INC  SI              ;Increment the pointers.
0050 47                   INC  DI
0051 E2 F6                LOOP CHECK           ;Repeat comparison until counter expires.
                          DISP MSG3            ;Display MSG3 using macro DISP.
0053 B4 09     +          MOV  AH,09H          ;Move function code to AH-register.
```

```
0055 BA 007A R +          MOV  DX,OFFSET MSG3 ;Move address of string to display in DX.
0058 CD 21     +          INT  21H            ;Call DOS service for display.
005A EB 08 90             JMP  EXIT
005D              FALSE:  DISP MSG4           ;Display MSG4 using macro DISP.
005D B4 09     +          MOV  AH,09H         ;Move function code to AH-register.
005F BA 0099 R +          MOV  DX,OFFSET MSG4 ;Move address of string to display in DX.
0062 CD 21     +          INT  21H            ;Call DOS service for display.
0064 B4 4C        EXIT:   MOV  AH,4CH         ;Move function code to AH-register.
0066 B0 00                MOV  AL,00H
0068 CD 21                INT  21H            ;Return to command prompt.
006A              CODE ENDS                   ;End of code segment.
                     ;Define data segment
0000               DATA SEGMENT               ;start of data segment.
= 000D                    CR   EQU 0DH        ;ASCII code for carriage return.
= 000A                    LF   EQU 0AH        ;ASCII code for line feed.
0000 28 [                 INS  DB 40 DUP(0)   ;Reserve 40 locations for input string.
        00
          ]
0028 28 [                 RES  DB 40 DUP(0)   ;Reserve 40 locations for reverse string.
        00
          ]
0050 45 4E 54 45 52 20        MSG1   DB‘ENTER THE STRING.INPUT STRING= ‘,’$’.
     54 48 45 20 53 54
     52 49 4E 47 2E 49
     4E 50 55 54 20 53
     54 52 49 4E 47 3D
     20 24
0070 0D 0A 52 45 56 45        MSG2   DB CR,LF,’REVERSE OF STRING = ‘,’$’.
     52 53 45 20 4F 46
     20 53 54 52 49 4E
     47 20 3D 20 24
0087 0D 0A 49 4E 50 55        MSG3   DB CR,LF,’INPUT STRING IS A PALINDROME’,’$’.
     54 20 53 54 52 49
     4E 47 20 49 53 20
     41 20 50 41 4C 49
     4E 44 52 4F 4D 45
     24
00A6 0D 0A 49 4E 50 55        MSG4   DB CR,LF,’INPUT STRING IS NOT A PALINDROME’,’$’.
     54 20 53 54 52 49
     4E 47 20 49 53 20
     4E 4F 54 20 41 20
     50 41 4C 49 4E 44
     52 4F 4D 45 24
00C9                   DATA ENDS              ;End of data segment.
                       END                    ;Assembly end.
```

EXAMPLE PROGRAM 30 : Program to Verify Password

Write an 8086 assembly language program to read a password through the PC keyboard and validate the user. Use the PC monitor to display the validity of the password.

Problem Analysis

The actual password can be stored in the memory as an ASCII array. The user entered password through the PC keyboard can be read by using BIOS service INT 16H with function code 00_H. A-register can be used as the counter to count the number of characters received and the received characters can be stored in the memory. The carriage return/enter key can be used to terminate the password.

First the count value is checked with the number of characters in the stored password. If it is not equal then an invalid password message can be displayed on the PC screen and request the user either to enter the correct password or to enter the ESC key to exit.

If the count is equal to the number of characters in the stored password, then the user entered password is checked with the stored password byte by byte. If the passwords are equal then an acceptance message can be displayed and the control can be returned to DOS prompt. If they are not equal then an invalid message can be displayed and request the user either to enter the correct password or to enter ESC key to exit.

In this program separate procedures has been written to clear the PC monitor screen and position the cursor at the desired location using the BIOS services.

Algorithm

1. Call procedure CLRS to clear the PC monitor screen.
2. Call procedure POS to position the cursor.
3. Initialize DS-register.
4. Load 09_H in AH and 0450_H in DX, and then call DOS service INT 21H to display the message "ENTER THE PASSWORD :".
5. Set SI as memory pointer to store user entered password.
6. Let CX be counter for number of characters in user entered password.Initialize CX as zero.
7. Load 00_H in AH and call BIOS service INT 16H to get the keycode in AL-register.
6. Compare AL with $0D_H$(ASCII value of ENTER key).
9. If ZF = 1, go to step 13, otherwise go to next step.
10. Load 02_H in AH and ASCII value of "*" in DL, and then call DOS service INT 21H to display the symbol "*".
11. Increment the pointer(SI) and count(CL).
12. Go to step 7.
13. Compare the count (CX-register) with 07_H (Here the stored password has 7 characters).
14. If ZF = 0, then go to step 20, otherwise go to next step.
15. Set SI as pointer for user entered password and DI as pointer for stored password.
16. Load a character of entered password in AL and compare with corresponding character stored password.
17. If ZF = 0, then go to step 25, otherwise go to next step.
16. Increment the pointer (SI and DI).
19. Decrement CX-register and check whether CX is zero.
20. If content of CX is zero then go to next step, otherwise go to step 16.
21. Call procedure CLRS to clear the PC monitor screen.
22. Call procedure POS to position the cursor.
23. Load 09_H in AH-register and 0600_H in DX-register and then call DOS service INT 21H to display the message "ENTRY ACCEPTED".
24. Go to step 32.
25. Call procedure CLRS to clear the PC monitor screen.
26. Call procedure POS to position the cursor.
27. Load 09_H in AH and 0550_H in DX, and then call DOS service INT 21H to display the message "INCORRECT PASSWORD".
26. Load 09_H in AH and 0580_H in DX, and then call DOS service INT 21H to display the message "PRESS ANY KEY TO TRY AGAIN OR PRESS ESC TO EXIT."
29. Load 00_H in AH and call BIOS service INT 16H to get the keycode in AL.
30. Compare AL with $1B_H$ (ASCII value of ESC key).
31. If ZF = 0, then go to step 1, otherwise go to next step.
32. Load $4C_H$ in AH-register and call DOS service INT 21H to terminate the program and return the control to DOS prompt.
33. Stop.

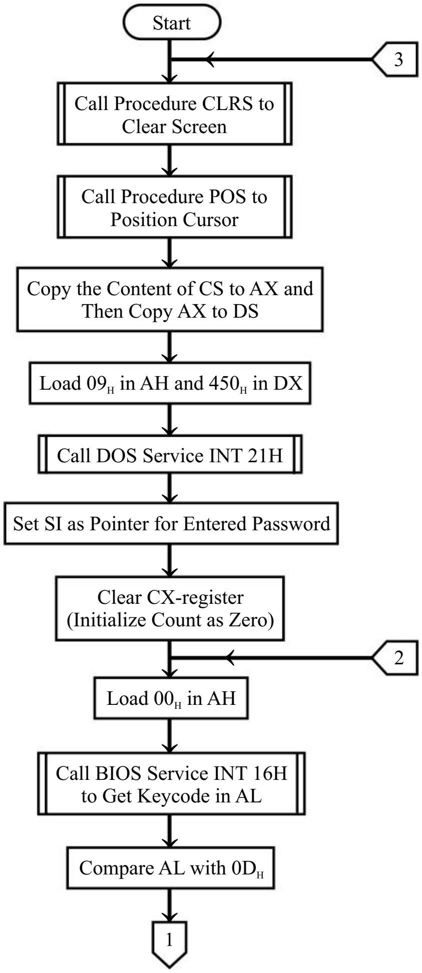
Algorithm for procedure CLRS

1. Load the function code 07_H in AH-register.
2. Load the number of lines to scroll down in AL-register.
3. Load the blanked area attribute (07_H) in BL-register.
4. Load the x and y coordinates of upper left corner in CL and CH registers.
5. Load the x and y coordinates of lower right corner in DL and DH.
6. Call BIOS video service INT 10H to clear the screen.
7. Return.

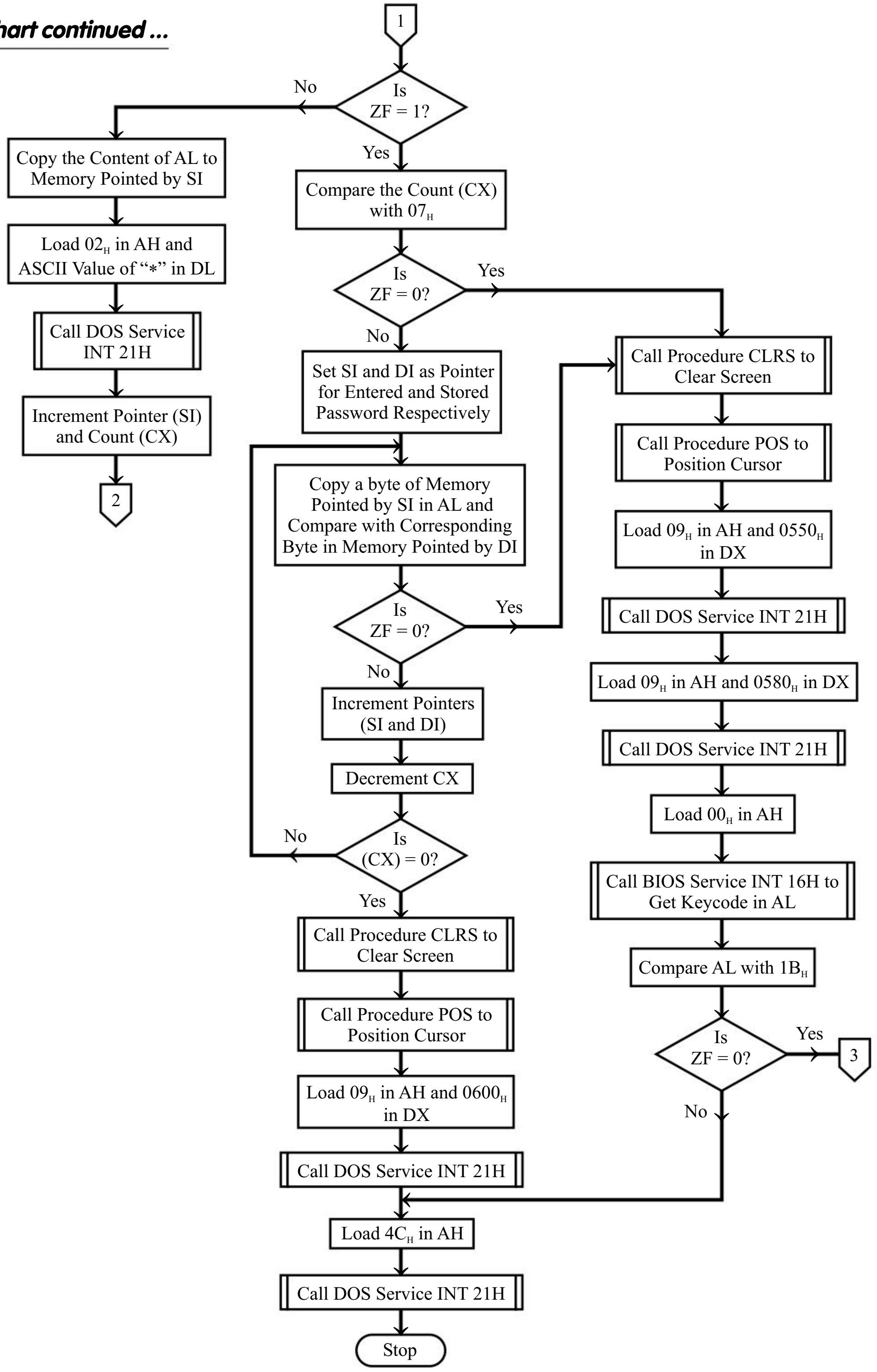
Algorithm for procedure POS

1. Load the function code 02_H in AH-register.
2. Load the video page (00_H) in BH-register.
3. Load the x and y coordinates of video page in DL and DH.
4. Call BIOS video service INT 10H to position the cursor.
5. Return.

Flowchart


Start
3
Call Procedure CLRS to Clear Screen
Call Procedure POS to Position Cursor
Copy the Content of CS to AX and Then Copy AX to DS
Load 09H in AH and 450H in DX
Call DOS Service INT 21H
Set SI as Pointer for Entered Password
Clear CX-register (Initialize Count as Zero)
2
Load 00H in AH
Call BIOS Service INT 16H to Get Keycode in AL
Compare AL with 0DH
1

Flowchart continued ...


1
Is ZF = 1?
No
Yes
Copy the Content of AL to Memory Pointed by SI
Load 02_H in AH and ASCII Value of "*" in DL
Call DOS Service INT 21H
Increment Pointer (SI) and Count (CX)
2
Compare the Count (CX) with 07_H
Is ZF = 0?
Yes
No
Set SI and DI as Pointer for Entered and Stored Password Respectively
Copy a byte of Memory Pointed by SI in AL and Compare with Corresponding Byte in Memory Pointed by DI
Is ZF = 0?
Yes
No
Increment Pointers (SI and DI)
Decrement CX
Is (CX) = 0?
No
Yes
Call Procedure CLRS to Clear Screen
Call Procedure POS to Position Cursor
Load 09_H in AH and 0600_H in DX
Call DOS Service INT 21H
Load $4C_H$ in AH
Call DOS Service INT 21H
Stop
Call Procedure CLRS to Clear Screen
Call Procedure POS to Position Cursor
Load 09_H in AH and 0550_H in DX
Call DOS Service INT 21H
Load 09_H in AH and 0580_H in DX
Call DOS Service INT 21H
Load 00_H in AH
Call BIOS Service INT 16H to Get Keycode in AL
Compare AL with $1B_H$
Is ZF = 0?
Yes
3
No

Assembly language program

```
        ;PROGRAM TO READ THE PASSWORD AND VALIDATE THE USER

CODE SEGMENT            ;Start of code segment.

        ASSUME CS:CODE ;Assembler directive.
        ASSUME DS:CODE ;Assembler directive.
        CR   EQU 0DH    ;Assign ASCII value for carriage return.

AGAIN:  CALL CLRS       ;Clear the screen.
        CALL POS        ;Position the cursor.

        MOV  AX,CS      ;Initialize DS to code segment.
        MOV  DS,AX

        MOV  AH,09H     ;Load the function code in AH.
        MOV  DX,450H    ;Load address of string to display in DX.
        INT  21H        ;Call DOS service for display.

        MOV  SI,400H    ;Initialize the pointer.
        MOV  CX,0000H   ;Initialize count in CX-register as zero.

L1:     MOV  AH,00H     ;Load the function code in AH.
        INT  16H        ;Call BIOS service to get keyboard character in AL.
        CMP  AL,CR      ;Compare the keycode with carriage return.
        JE   CHECK      ;If equal go to CHECK.
        MOV  [SI],AL    ;Store the keyboard character in memory.
        MOV  AH,02H     ;Load the function code in AH.
        MOV  DL,'*'     ;Load the ASCII value of '*' in DL.
        INT  21H        ;Call DOS service for display.
        INC  SI         ;Increment pointer.
        INC  CX         ;Increment the count.
        JMP  L1         ;Jump to L1 to get next character.

CHECK:  CMP  CX,0007H   ;Compare count with 07H.
        JNE  REPEAT     ;If count is not equal to 07H,
                        ;then go to REPEAT.
        MOV  SI,400H    ;Initialize the pointers.
        MOV  DI,500H
L2:     MOV  AL,[SI]    ;Get a character of entered password in AL.
        CMP  AL,[DI]    ;Compare entered character with
                        ;corresponding character of stored password.
        JNE  REPEAT     ;If not equal jump to REPEAT.
        INC  SI         ;Increment the pointers.
        INC  DI
        LOOP L2         ;Repeat comparison until counter expires.

        CALL CLRS       ;Clear the screen.
        CALL POS        ;Position the cursor.
        MOV  AH,09H     ;Load the function code in AH.
        MOV  DX,600H    ;Load address of string to display in DX.
        INT  21H        ;Call DOS service for display.
        JMP   EXIT      ;Go to exit.

REPEAT: CALL CLRS       ;Clear the screen.
        CALL POS        ;Position the cursor.

        MOV  AH,09H     ;Load the function code in AH.
        MOV  DX,550H    ;Load address of string to display in DX.
        INT  21H        ;Call DOS service for display.

        MOV  AH,09H     ;Load the function code in AH.
        MOV  DX,580H   ;Load address of string to display in DX.
        INT  21H       ;Call DOS service for display.

        MOV  AH,00H    ;Load the function code in AH.
        INT  16H       ;Call BIOS service to get keyboard
                       ;Character in AL.
        CMP  AL,1BH    ;Check for esc key.
        JNE  AGAIN     ;If key pressed is not esc, then go to AGAIN.
        JMP  EXIT      ;If key pressed is esc, then go to EXIT.

POS PROC NEAR          ;Start of procedure to position the cursor.

        MOV  AH,02H    ;Load the function code in AH.
        MOV  BH,00H    ;Load the video page in BH.
        MOV  DX,0000H  ;Load the x and y coordinates in DL and DH.
        INT  10H       ;Call BIOS service for cursor positioning.
        RET            ;Return to main program.

POS ENDP               ;End of procedure to position the cursor.

CLRS PROC NEAR         ;Start of procedure to clear the screen.

        MOV  AH,07H    ;Load the function code in AH.
        MOV  AL,00H    ;Load number of lines to scroll down in AL.
        MOV  BH,07H    ;Load the blanked area attribute in BL.
        MOV  CX,0000H  ;Load the x and y coordinates of
                       ;upper left corner in CL and CH.
        MOV  DX,184FH  ;Load the x and y coordinates
                       ;of lower right corner in DL and DH.
        INT  10H       ;Call BIOS video service for clearing screen.
        RET            ;Return to main program.

CLRS ENDP              ;End of procedure to clear the screen.
        ORG  450H
        DB ‘ENTER THE PASSWORD : ‘,’$’.
        ORG  500H
        DB ‘WELCOME’,’$’.
        ORG  550H
        DB ‘INCORRECT PASSWORD.  ‘,’$’.
        ORG  580H
        DB ‘PRESS ANY KEY TO TRY AGAIN OR PRESS ESC TO EXIT ‘,’$’.
        ORG  600H
        DB ‘ENTRY ACCEPTED’,’$’.
EXIT:   MOV  AH,4CH    ;Load the function code in AH.
        INT  21H       ;Call DOS service to return to command prompt.

CODE ENDS              ;End of code segment.
END                    ;Assembly end.
```

Assembler listing for example program 30

```
          ;PROGRAM TO READ THE PASSWORD AND VALIDATE THE USER

0000               CODE SEGMENT                ;Start of code segment.

                            ASSUME CS:CODE  ;Assembler directive.
                            ASSUME DS:CODE  ;Assembler directive.
=000D                       CR   EQU 0DH    ;Assign ASCII value for carriage return.

0000   E8 0079 R   AGAIN:   CALL CLRS       ;Clear the screen.
0003   E8 006F R            CALL POS        ;Position the cursor.

0006   8C C8                MOV  AX,CS      ;Initialize DS to code segment.
0008   8E D8                MOV  DS,AX
```

```
000A  B4 09                  MOV  AH,09H     ;Load the function code in AH.
000C  BA 0450                MOV  DX,450H    ;Load address of string to display in DX.
000F  CD 21                  INT  21H        ;Call DOS service for display.

0011  BE 0400                MOV  SI,400H    ;Initialize the pointer.
0014  B9 0000                MOV  CX,0000H   ;Initialize count in CX-register as zero.

0017  B4 00        L1:       MOV  AH,00H     ;Load the function code in AH.
0019  CD 16                  INT  16H        ;Call BIOS service to get keyboard character in AL .
001B  3C 0D                  CMP  AL,CR      ;Compare the keycode with carriage return.
001D  74 0C                  JE   CHECK      ;If equal go to CHECK.
001F  88 04                  MOV  [SI],AL    ;Store the keyboard character in memory.
0021  B4 02                  MOV  AH,02H     ;Load the function code in AH.
0023  B2 2A                  MOV  DL,’*’     ;Load the ASCII value of ‘*’ in DL.
0025  CD 21                  INT  21H        ;Call DOS service for display.
0027  46                     INC  SI         ;Increment pointer.
0028  41                     INC  CX         ;Increment the count.
0029  EB EC                  JMP  L1         ;Jump to L1 to get next character.

002B  83 F9 07     CHECK:    CMP  CX,0007H   ;Compare count with 07H.
002E  75 20                  JNE  REPEAT     ;If count is not equal to 07H,
                                             ;then go to REPEAT.
0030  BE 0400                MOV  SI,400H    ;Initialize the pointers.
0033  BF 0500                MOV  DI,500H
0036  8A 04        L2:       MOV  AL,[SI]    ;Get a character of entered password in AL.
0038  3A 05                  CMP  AL,[DI]    ;Compare entered character with
                                             ;corresponding character of stored password.
003A  75 14                  JNE  REPEAT     ;If not equal jump to REPEAT.
003C  46                     INC  SI         ;Increment the pointers.
003D  47                     INC  DI
003E  E2 F6                  LOOP L2         ;Repeat comparison until counter expires.

0040  E8 0079 R              CALL CLRS       ;Clear the screen.
0043  E8 006F R              CALL POS        ;Position the cursor.
0046  B4 09                  MOV  AH,09H     ;Load the function code in AH.
0048  BA 0600                MOV  DX,600H    ;Load address of string to display in DX.
004B  CD 21                  INT  21H        ;Call DOS service for display.
004D  E9 060F R              JMP  EXIT       ;Go to exit.

0050  E8 0079 R     REPEAT:  CALL CLRS       ;Clear the screen.
0053  E8 006F R              CALL POS        ;Position the cursor.

0056  B4 09                  MOV  AH,09H     ;Load the function code in AH.
0058  BA 0550                MOV  DX,550H    ;Load address of string to display in DX.
005B  CD 21                  INT  21H        ;Call DOS service for display.

005D  B4 09                  MOV  AH,09H     ;Load the function code in AH.
005F  BA 0580                MOV  DX,580H    ;Load address of string to display in DX.
0062  CD 21                  INT  21H        ;Call DOS service for display.

0064  B4 00                  MOV  AH,00H     ;Load the function code in AH.
0066  CD 16                  INT  16H        ;Call BIOS service to get keyboard
                                             ;character in AL.
0068  3C 1B                  CMP  AL,1BH     ;Check for esc key.
006A  75 94                  JNE  AGAIN      ;If key pressed is not esc, then go to AGAIN.
006C  E9 060F R              JMP  EXIT       ;If key pressed is esc,then go to EXIT.
006F               POS PROC NEAR             ;Start of procedure to position the cursor.

006F  B4 02                  MOV  AH,02H     ;Load the function code in AH.
```

```
0071  B7 00               MOV  BH,00H   ;Load the video page in BH.
0073  BA 0000             MOV  DX,0000H ;Load the x and y coordinates in DL and DH.
0076  CD 10               INT  10H      ;Call BIOS service for cursor positioning.
0078  C3                  RET           ;Return to main program.
0079                      POS  ENDP     ;End of procedure to position the cursor.
0079              CLRS PROC NEAR        ;Start of procedure to clear the  screen.

0079  B4 07               MOV  AH,07H   ;Load the function code in AH.
007B  B0 00               MOV  AL,00H   ;Load number of lines to scroll down in AL.
007D  B7 07               MOV  BH,07H   ;Load the blanked area attribute in BL.
007F  B9 0000             MOV  CX,0000H ;Load the x and y coordinates of
                                        ;upper left corner in CL and CH.
0082  BA 184F             MOV  DX,184FH ;Load the x and y coordinates
                                        ;of lower right corner in DL and DH.
0085  CD 10               INT  10H      ;Call BIOS video service for clearing screen.
0087  C3                  RET           ;Return to main program.
0088              CLRS ENDP             ;End of procedure to clear the screen.

0450                      ORG  450H
0450  45 4E 54 45 52 20   DB ‘ENTER THE PASSWORD : ‘,’$’.
      54 48 45 20 50 41
      53 53 57 4F 52 44
      20 3A 20 24

0500                      ORG  500H
0500  57 45 4C 43 4F 4D   DB ‘WELCOME’,’$’.
      45 24

0550                      ORG  550H
0550  49 4E 43 4F 52 52   DB ‘INCORRECT PASSWORD.  ‘,’$’.
      45 43 54 20 50 41
      53 53 57 4F 52 44
      2E 20 20 24

0580                      ORG  580H
0580  50 52 45 53 53 20   DB ‘PRESS ANY KEY TO TRY AGAIN OR PRESS ESC TO EXIT ‘,’$’.
      41 4E 59 20 4B 45
      59 20 54 4F 20 54
      52 59 20 41 47 41
      49 4E 20 4F 52 20
      50 52 45 53 53 20
      45 53 43 20 54 4F
      20 45 58 49 54 20
      24

0600                      ORG  600H
0600  45 4E 54 52 59 20   DB ‘ENTRY ACCEPTED’,’$’.
      41 43 43 45 50 54
      45 44 24

060F  B4 4C       EXIT:   MOV  AH,4CH   ;Load the function code in AH.
0611  CD 21               INT  21H      ;Call DOS service to return to command prompt.

0613              CODE ENDS             ;End of code segment.
                  END                   ;Assembly end.
```

6.10 SHORT QUESTIONS AND ANSWERS

6.1 What is meant by a program?

A program is a set of instructions written to perform a certain task.

6.2 What is assembler, interpreter and compiler?

Assembler : It is a software that converts assembly language program codes to machine language codes.

Compiler : It is a software that converts the programs written in high level language to machine language.

Interpreter : It is similar to a compiler but it converts the instructions one by one.

6.3 What is the need for an assembler?

The 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.

6.4 What are the advantages of an assembler?

The advantages of the 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 easier to alter the program and reassemble it.
4. The assembler identifies the syntax errors.
5. The assembler can reserve memory locations for data or result.
6. The assembler provides the list file for documentation.

6.5 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 informations like start and end of a program, values of variables used in the program, storage locations for input and output data, etc.

6.6 List some of the assembler directives of a typical 8085/8086 assembler.

Some of the assembler directives of a typical 8085/8086 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.

6.7 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.

6.8 What is expanding the macro?

While assembling a program, the assembler replaces the instructions represented by a macro in the place where the macro is called. This process is called expanding the macro.

6.9 What is the disadvantage in a macro?

The disadvantage in macro is that, if it is expanded or used a number of times in a program then the program may occupy more memory.

6.10 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.

6.11 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.

6.12 What is a list?

A 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.

6.13 What are the types of linked data structures?

The different types of linked data structures are linear linked listed, linked lists with multiple pointers, circular linked lists and trees.

6.14 Whar is an array?

An array is a series of data of the same type stored in successive memory lacations. Each value in the array is referred to as an element of the array.

6.15 What is a flowchart?

A flowchart is a graphical representation of the operation flow of the program. It is the graphical (pictorial) form of an algorithm.

6.16 List the symbols used for drawing a flowchart.

The following symbols are used for drawing a flowchart :

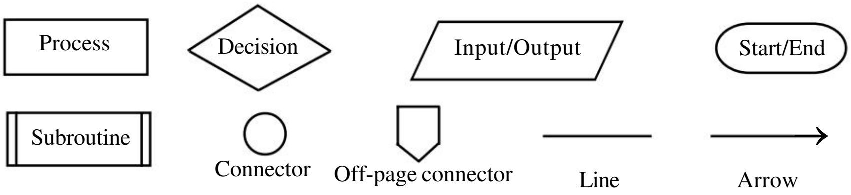


Fig. Q.6.16 : Symbols used in flowcharts.

6.17 *What is development system? What are its components?*

A development system is a system used by a microprocessor-based system designer to design and test the software and hardware aspects of a new system under development.

The components of development system are a microcomputer with standard accessories, emulator and program development tools like editor, assembler, linker, locator, debugger, simulator, etc.

6.18 *Write a short note on assembly language program development tools.*

The program development tools include the 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.

6.19 *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 it as a file.

6.20 *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.

6.21 *What is a two-pass assembler?*

A two-pass assembler is an assembler in which the source codes are processed twice. In the first pass the assembler assigns addresses to all the labels and attaches values to all the variables used in the program. In the second pass it converts the source code into machine code.

6.22 *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 the one-pass assembler issues an error message if it encounters a label or variable that is defined at a later part of a program.

6.23 *What is linker and locator?*

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 specific addresses to the object codes to be loaded into memory.

6.24 *What is debugging?*

A process of locating and correcting an error using a debugger is known as debugging.

6.25 *What is a debugger?*

A debugger is a software used to locate and troubleshoot the errors in a program.

6.26 *What is simulator?*

The simulator is a program which can be run on the development system 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 the result.
- Break-point execution of a program.
- Single-step execution of a program.
- Display the content of a register/memory.

6.27 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.

6.28 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.

6.29 How can one access DOS services?

The steps involved in accessing DOS services are :

1. Load a DOS function number in AH-register. If there is a subfunction, then its code is loaded in AL-register.
2. Load the other registers as indicated in the DOS service formats.
3. Prepare buffers, ASCIIZ (ASCII string terminated by zero) and control blocks, if necessary.
4. Set the location of the Disk Transfer Area if necessary.
5. Invoke DOS service INT 21H.
6. The DOS service will return the required parameters in the specified registers.

6.30 How does one access BIOS services?

The steps involved in accessing the BIOS services are :

1. Load a BIOS function number in the AH-register. If there is a subfunction, then its code is loaded in AL-register.
2. Load the other register as indicated in the BIOS service formats.
3. Prepare buffers, ASCIIZ (ASCII string terminated by zero) and control blocks if necessary.
4. Invoke BIOS call.
5. The BIOS service will return the required parameters in the specified register.

CHAPTER 7

PERIPHERAL DEVICES AND INTERFACING

7.1 PROGRAMMABLE PERIPHERAL DEVICES

The 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 various functions it has to perform. The format of the control word will be specified by the manufacturer of the peripheral devices.

INTEL have 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 systems are Parallel peripheral interface-8255, Serial communication interface-8251, Keyboard/Display controller-8279, Programmable Timer 8253/8254 and DMA controller-8237. A brief discussion about these devices and their interfacing with 8086 processor are presented in this chapter.

The 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 processor and another system. The 8279 is used to relieve the processor from time consuming routine activities like keyboard scanning and display refreshing. The programmable timers are used to maintain various timings and to initiate time-based activities. The DMA controllers are used to achieve very fast data transfer between memory and IO devices by bypassing the processor.

7.2 PARALLEL DATA COMMUNICATION INTERFACE

In microprocessor-based systems, digital informations 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) are 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 scheme are adopted to transfer data between various devices inside the system.

Basically, the microprocessor-based system has been fabricated on a PCB (**P**rinted **C**ircuit **B**oard) in which a bus is formed with 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 0.5 m), we require high current signals to drive the data for longer distance. In such cases data are transmitted bit by bit through a single data line.

7.2.1 Parallel Data Transfer Schemes

The 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, 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.

The 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 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. **D**irect **M**emory **A**ccess (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 are 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.

The programmed data transfer scheme can be further classified into the following three types:

1. Synchronous data transfer scheme.
2. Asynchronous data transfer scheme.
3. 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 instructions during the **HOLD** period.

The DMA data transfer is used for a large block of data transfer between the 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:

a) Cycle stealing DMA or Single transfer mode DMA.
b) Block or Burst mode DMA.
c) Demand transfer mode DMA.

Figure 7.1 shows the various types of data transfer scheme. All the data transfer schemes discussed above require 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.

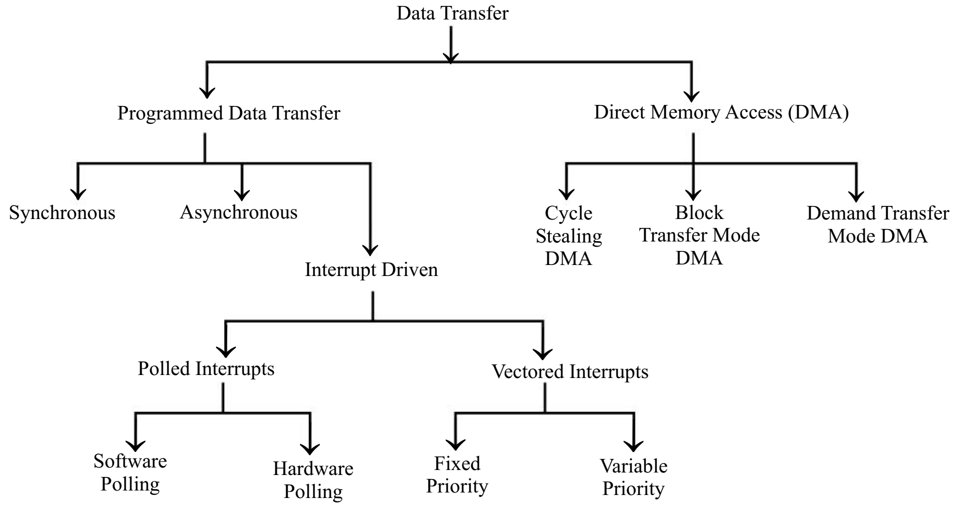


Fig. 7.1 : Types of data transfer schemes.

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 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. 7.2. The mode-0 input/output in 8255 is an example of synchronous data transfer.

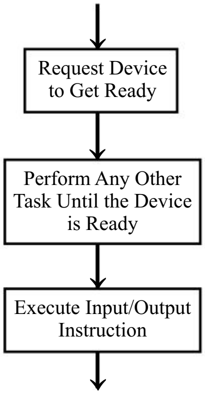

Request Device to Get Ready
Perform Any Other Task Until the Device is Ready
Execute Input/Output Instruction

Fig. 7.2 : Synchronous data transfer scheme.

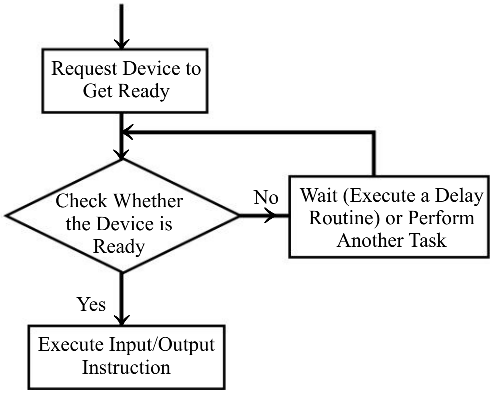

Request Device to Get Ready
Check Whether the Device is Ready
No
Wait (Execute a Delay Routine) or Perform Another Task
Yes
Execute Input/Output Instruction

Fig. 7.3 : Asynchronous data transfer scheme.

Asynchronous Data Transfer Scheme

The asynchronous data transfer scheme is employed when the speed of the processor and IO device do 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. 7.3. 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 call an **I**nterrupt **S**ervice **S**ubroutine (ISS) to service the interrupting device. At the end of ISS, the processor status is retrieved from stack and the processor starts executing its main program. The sequence of operations for an interrupt driven data transfer scheme is shown in Fig. 7.4. (For detailed discussion on interrupt driven data transfer scheme, please refer to Chapter-5.)

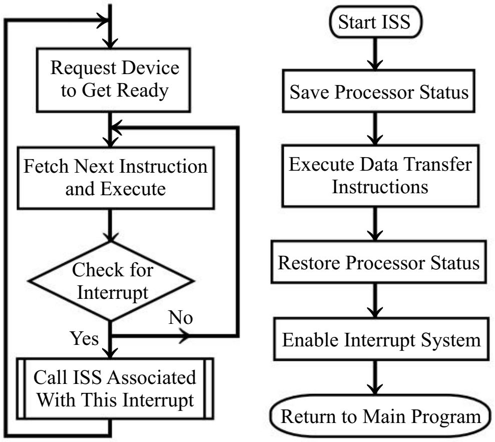

Request Device to Get Ready
Fetch Next Instruction and Execute
Check for Interrupt
No
Yes
Call ISS Associated With This Interrupt
Start ISS
Save Processor Status
Execute Data Transfer Instructions
Restore Processor Status
Enable Interrupt System
Return to Main Program

Fig. a : Main program execution sequence.

Fig. b : ISS execution sequence.

Fig. 7.4 : Interrupt driven data transfer scheme.

7.2.2 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 functions :

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 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

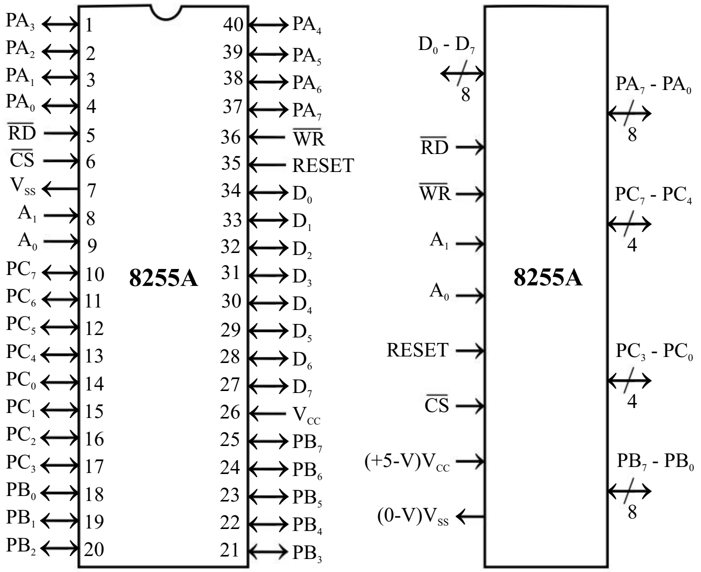
The pin description of 8255 is shown in Fig. 7.5. It has 40 pins and requires a single +5-V supply. The internal block diagram of 8255 is shown in Fig. 7.6.

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 port-B, port-C lower and its control circuit. The read/ write control logic requires six control signals. These signals are given below:

$\overline{\text{RD}}$ (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.

$\overline{\text{WR}}$ (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.



Pin	Description
D_0 - D_7	Data Lines
RESET	Reset Input
$\overline{CS}$	Chip Select
$\overline{RD}$	Read Control
$\overline{WR}$	Write Control
A_0, A_1	Internal Address
PA_7 - PA_0	Port-A Pins
PB_7 - PB_0	Port-B Pins
PC_7 - PC_0	Port-C Pins
V_{CC}	+5-V
V_{SS}	0-V (GND)

Fig. 7.5 : Pin description of 8255.

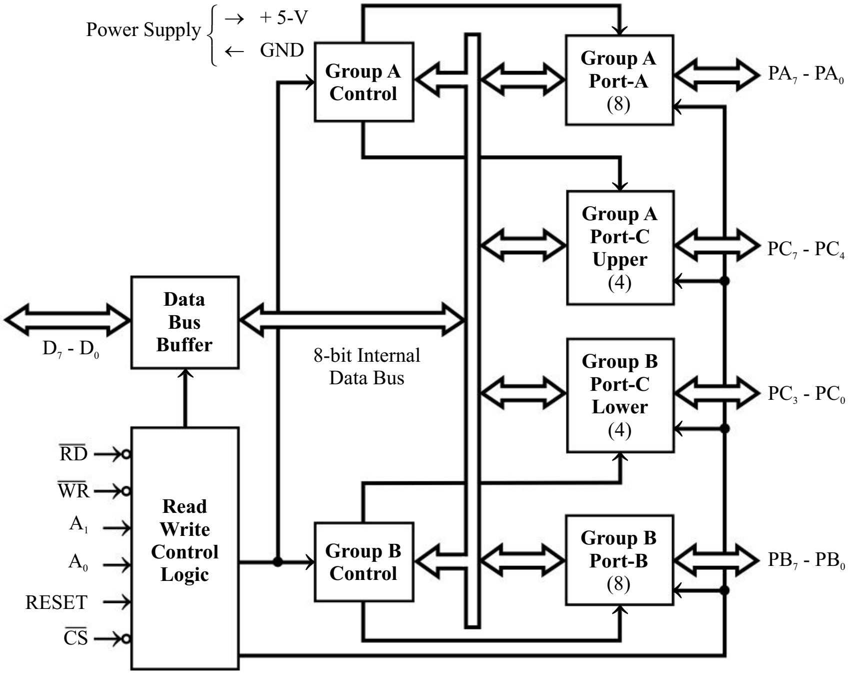


Fig. 7.6 : Internal block diagram of 8255.

$\overline{CS}$, 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. A_0 and A_1 selects any one of the 4 internal devices as shown in Table-7.1. 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 - 7.1

Internal address		Device selected
A_1	A_0	
0	0	Port-A
0	1	Port-B
1	0	Port-C
1	1	Control Register

Interfacing of 8255 with 8086 Processor

A simple schematic for interfacing the 8255 with 8086 processor is shown in Fig. 7.7. The 8255 can be either memory-mapped or IO-mapped in the system. In the schematic shown in Fig. 7.7, the 8255 is IO-mapped in the system with even addresses. The chip select signals for IO-mapped devices are generated by using a 3-to-8 decoder. The address lines A_5, A_6 and A_7 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_0 and the control signal $M/\overline{IO}$ are used as enable for decoder.

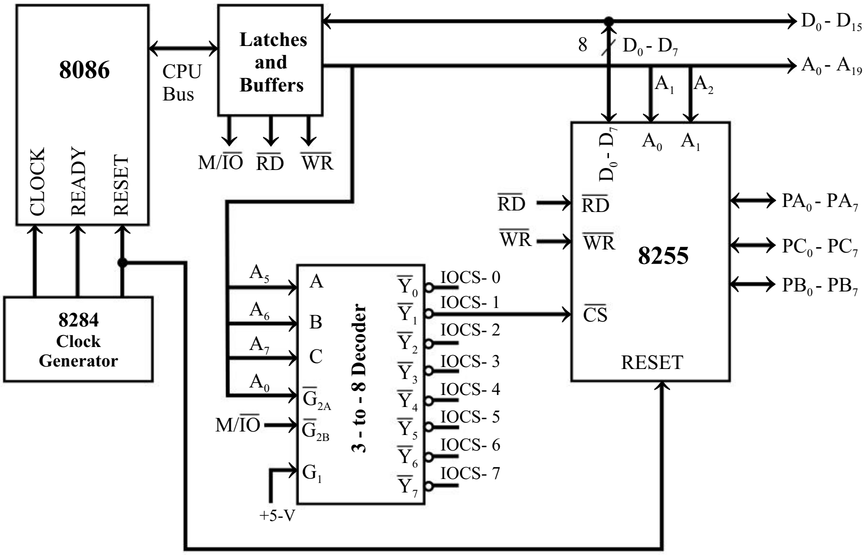


Fig. 7.7 : Interfacing of 8255 with 8086 processor.

The address line A_1 of 8086 is connected to A_0 of 8255 and A_2 of 8086 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-7.2. The data lines D_0-D_7 are connected to D_0-D_7 of the processor to achieve parallel data transfer.

TABLE - 7.2 : IO ADDRESSES OF 8255

Internal device	Binary address								Hexa address
	Decoder input			Input to address pins of 8255				Decoder enable	
	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
Port-A	0	0	1	x	x	0	0	0	20
Port-B	0	0	1	x	x	0	1	0	22
Port-C	0	0	1	x	x	1	0	0	24
Control Register	0	0	1	x	x	1	1	0	26

Note : Don't care "x" is considered as zero.

In the schematic shown in Fig. 7.7, 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 **M**ode **S**et control **W**ord (MSW) and **B**it **S**et/**R**eset (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 the IO functions in various modes. If bit $B_7 = 0$, then the bits B_6- B_0 determine whether the pin of port-C is to be set or reset.

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. 7.8 and the format of bit set/reset control word is shown in Fig. 7.9. The various functions (assignments) of port-C pins during the different operating modes of ports A and B are listed in Table-7.3.

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 8086 processor through NMI/INTR pin for a read/write operation. In status check technique, the 8086 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.

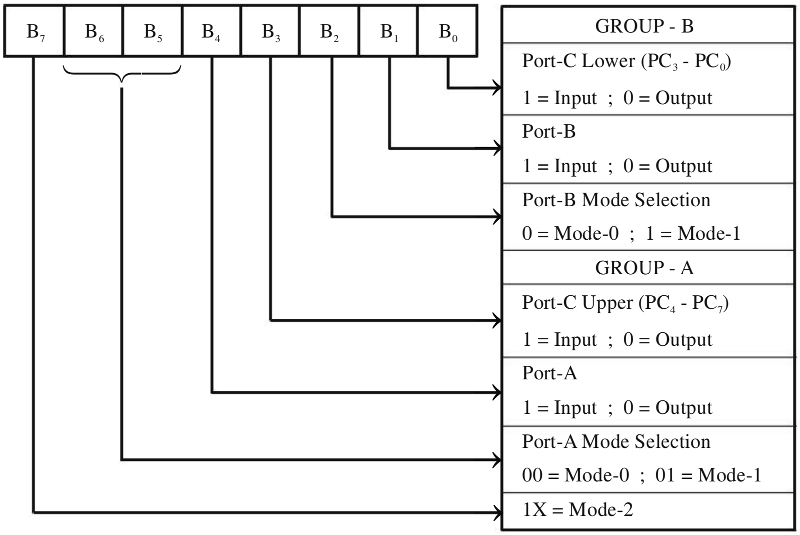
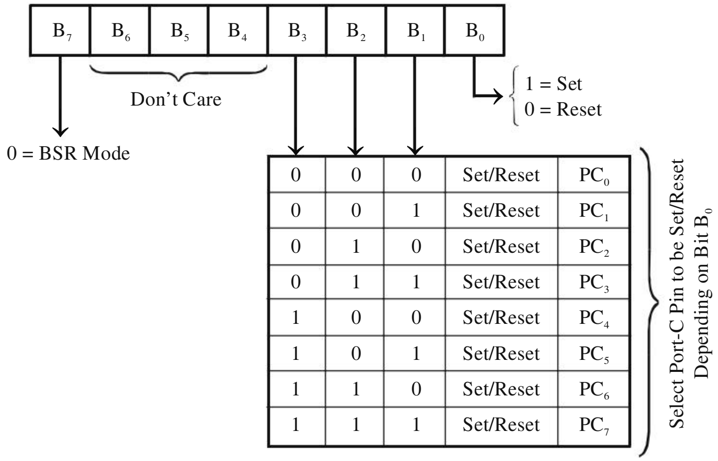


Fig. 7.8 : Format of IO mode set control word of 8255.



0	0	0	Set/Reset	PC_0
0	0	1	Set/Reset	PC_1
0	1	0	Set/Reset	PC_2
0	1	1	Set/Reset	PC_3
1	0	0	Set/Reset	PC_4
1	0	1	Set/Reset	PC_5
1	1	0	Set/Reset	PC_6
1	1	1	Set/Reset	PC_7

Fig. 7.9 : Format of Bit Set/Reset control word of 8255.

TABLE - 7.3 : PORT-C PIN ASSIGNMENTS

Functions of ports A and B	PC_7	PC_6	PC_5	PC_4	PC_3	PC_2	PC_1	PC_0
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_A	$\overline{\text{STB}}_A$	INTR_A	$\overline{\text{STB}}_B$	IBF_B	INTR_B
Ports A and B in mode-1 Output ports	$\overline{\text{OBF}}_A$	$\overline{\text{ACK}}_A$	IO	IO	INTR_A	$\overline{\text{ACK}}_B$	$\overline{\text{OBF}}_B$	INTR_B
Port-A in mode-2 Port-B in mode-0	$\overline{\text{OBF}}_A$	$\overline{\text{ACK}}_A$	IBF_A	$\overline{\text{STB}}_A$	INTR_A	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.		

The 8255 has two internal flip-flops as interrupt enables (INTE_A and INTE_B) 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_A is enabled by setting PC_4 to **high** and INTE_B is enabled by setting PC_2 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) the 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. 7.10.

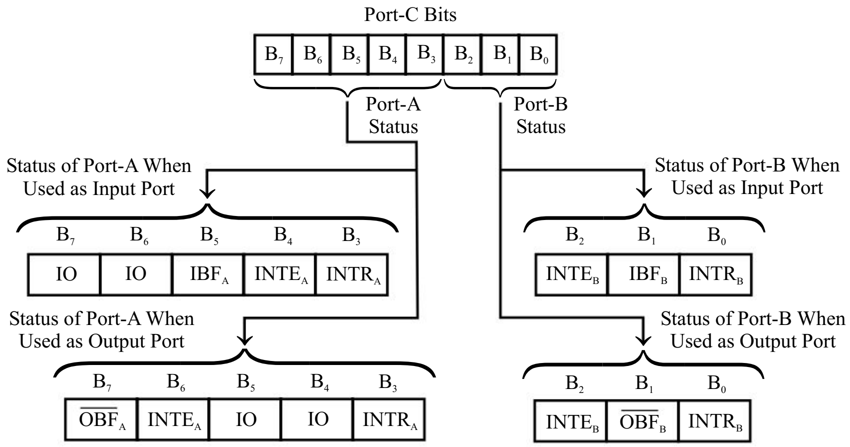


Fig. 7.10 : Format of status word of 8255 for handshake input and output operation.

8255 handshake input port (Mode-1)

The signals used for data transfer between input device and microprocessor using port-A of 8255 as handshake input port (Mode-1) are shown in Fig. 7.11.

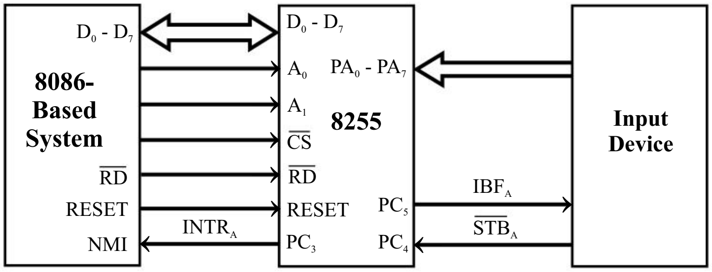


Fig. 7.11 : Port-A of 8255 as handshake input port (Mode-1).

1. The input device checks IBF$_A$ signal, if it is **low** then the input device places the data on the port lines PA_0-PA_7 and asserts $\overline{STB}_A$ **low** and after a delay time $\overline{STB}_A$ is asserted **high.**
2. When $\overline{STB}_A$ is **low** the 8255 asserts IBF signal **high** and at the rising edge of $\overline{STB}_A$ the data is latched to the port and INTR$_A$ is set **high.**
3. When INTR$_A$ 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 $\overline{RD}$ **low** and then **high.**
4. When $\overline{RD}$ is **low**, the INTR$_A$ is resetted (asserted **low**) by 8255 and at the rising edge of $\overline{RD}$, the 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_0, PC_1 and PC_2 are used.

8255 handshake output port (Mode-1)

The signals used for data transfer between output device and microprocessor using port-A of 8255 as handshake output port (Mode-1) are shown in Fig. 7.12.

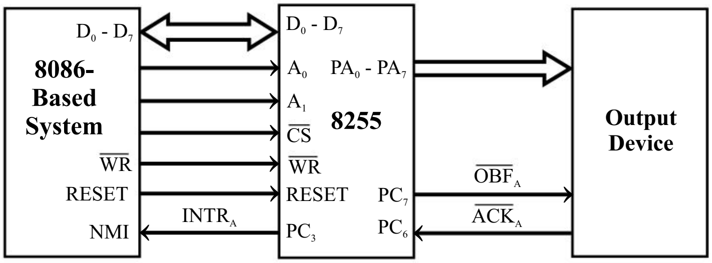


Fig. 7.12 : Port-A of 8255 as handshake output port (Mode-1).

1. When the port is empty, the processor writes a byte in the port.
2. For writing a data to the 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 IO device and microprocessor using port-A of 8255 as bidirectional port (Mode-2) are shown in Fig. 7.13.

> *Note : Only port-A can work in mode-2.*

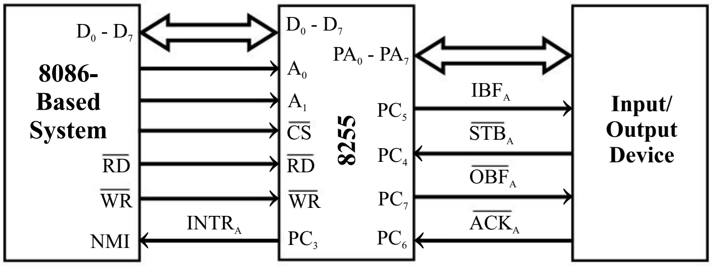


Fig. 7.13 : Port-A of 8255 as bidirectional port (Mode-2).

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 the 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 the mode-1 output port.

7.2.3 DMA Data Transfer Scheme

Normally the data transfer from memory to IO device or IO device to memory can be acheived only through the 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, the data cannot be directly transferred between memory and IO device, even though they are connected to a common bus. The above process is inevitable because the processor cannot simultaneously select two devices. Hence a scheme called **D**irect **M**emory **A**ccess (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 amounts of data between the memory and IO device in a short time.

For direct data transfer between IO device and memory a dedicated hardware device called **D**irect **M**emory **A**ccess 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. 7.14. In the system shown in Fig. 7.14, the DMA controller has one channel, which serves for one IO device. In an 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 simplicity let us consider a one-channel DMA controller.

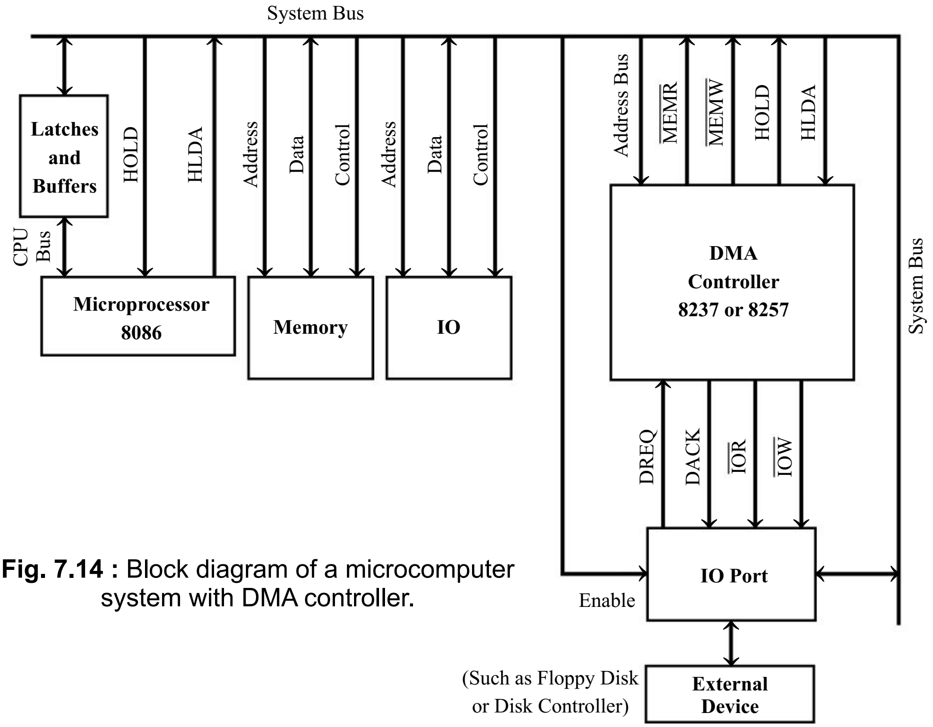


Fig. 7.14 : Block diagram of a microcomputer system with DMA controller.

The DMA controller can work as a slave or as a master. In the slave mode, the microprocessor loads the address register with the starting address of the memory, loads the count register with the number of bytes to be transferred and loads the control register with the control information.

For performing DMA operation the processor has to initialize or program the IO device and DMA controller. Consider an example of transferring a 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 the 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 **D**MA **req**uest 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 ack**nowledge 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 reading memory locations. 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 writing data to the 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.

7.2.4 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 high speed data transfer between memory and IO device. The 8237 has four channels and so it can be used to provide DMA to four IO devices. When more than four devices require DMA, then a number of 8237 can be connected in cascade to increase the DMA channels.

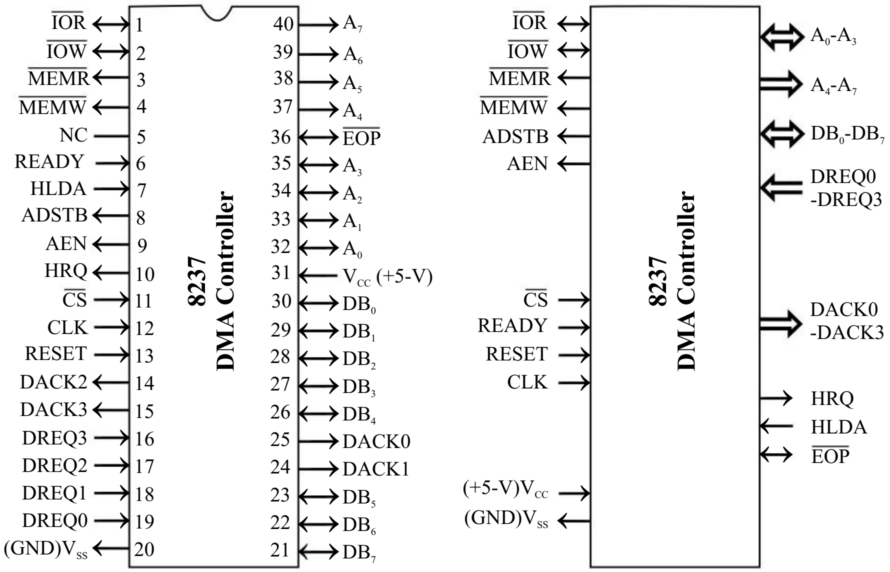


Fig. 7.15 : Pin configuration of 8237.

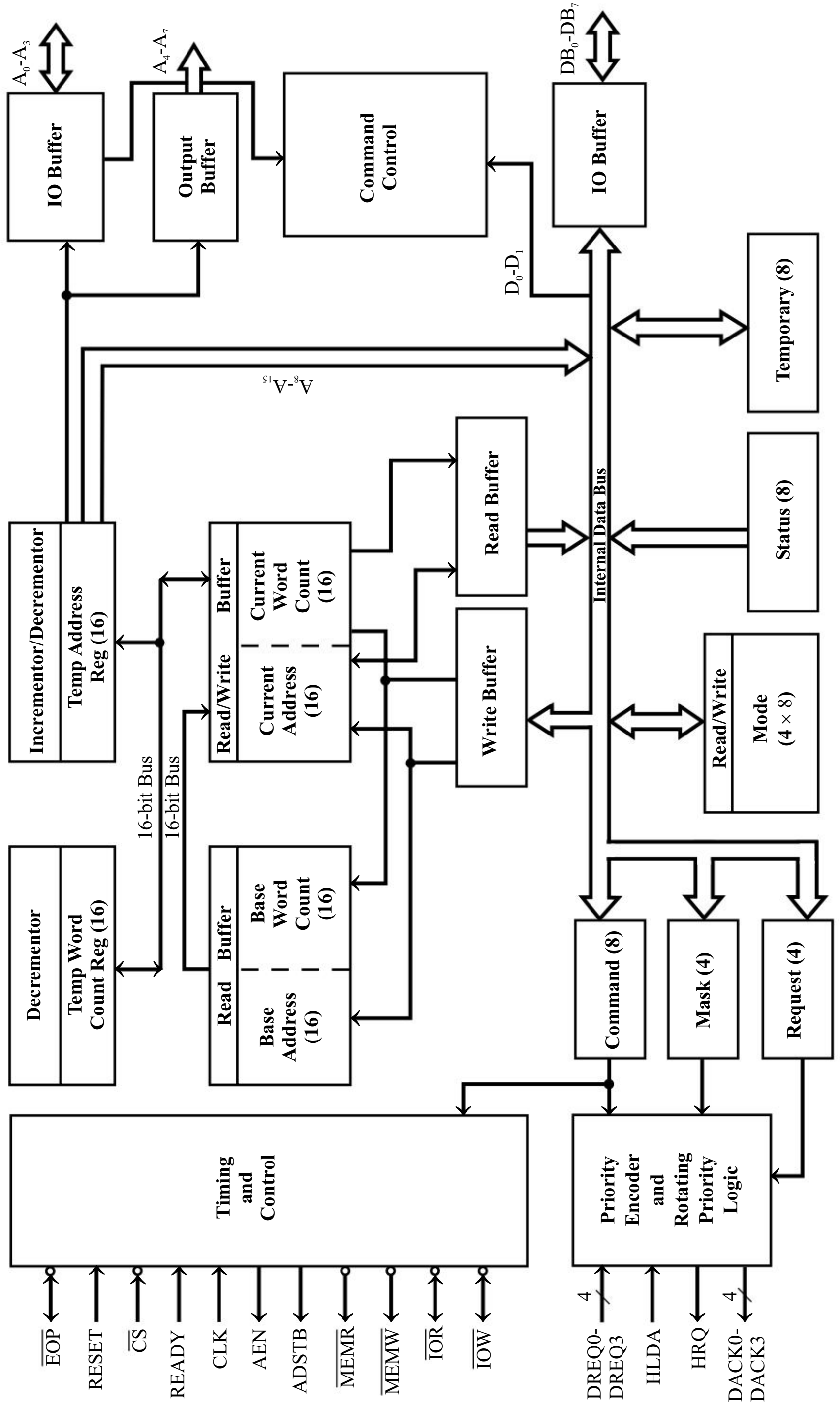


Fig. 7.16 : Functional block diagram of an 8237.

For each DMA channel, a set of registers has been dedicated to store the memory address and the count value for the 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 available in a **D**ual **I**n-**L**ine **P**ackage (DIP). The pin configuration of 8237 is shown in Fig. 7.15. A brief description about the pins and signals of 8237 are listed in Table-7.4. The functional block diagram of 8237 is shown in Fig. 7.16.

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.

TABLE - 7.4 : PIN DESCRIPTION OF 8237

Pin	Description
CLK	Clock input to 8237. Maximum clock frequency is 5 MHz. In 8086/8088 system, the processor clock is inverted and applied to the CLK of 8237.
$\overline{\text{CS}}$	Logic **low** chip select signal. It is input signal to select the 8237 during the programming mode.
RESET	Reset input to 8237. Connected to system reset, when the RESET signal goes **high** the command, status, request and temporary registers are cleared. It also clears the 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 **low**, the 8237 enters a wait state. This is used to get extra time in the 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 the 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 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 output signals from 8237 to the IO devices to inform the acceptance of DMA request. These outputs are programmable as either active **high** or active **low** signals.

Table - 7.4 continued...

Pin	Description
DB_7 to DB_0	Data bus lines. These pins are used for data transfer between the processor and DMA controller during the 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 the input control signal for reading the DMA controller during the programming mode and the output control signal for reading the IO device during DMA (memory) write cycle.
$\overline{\text{IOW}}$	Bidirectional IO write control signal. It is an input control signal for writing the DMA controller during the programming mode and output control signal for writing IO device during the DMA (memory) read cycle.
$\overline{\text{EOP}}$	End of process. It is a bidirectional active low signal. It is used either as input to terminate a DMA process or as output to inform the end of the DMA transfer to the processor. This output can be used as interrupt to terminate the DMA.
A_3 to A_0	Four bidirectional address lines. Used as input address during the 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_7 to A_4	Four unidirectional address lines. Used to output the memory address bits A_7 to A_4 during DMA mode.
AEN	Address enable output signal. It is used to enable the address latch connected to the DB_7 - DB_0 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_7 to DB_0 lines by 8237 during the 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-7.5. The processor can read or write into these registers. But with certain registers the processor can perform only read operation and with certain registers the processor can perform only write operation. The internal registers are selected by a 4-bit address supplied through A_0 - A_3 lines of 8237. The addresses of the internal registers and the operations (read/write) that can be performed on these registers are listed in Table-7.6.

TABLE - 7.5 : 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

The 16-bit internal registers of 8237 are read/write through an 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 the 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 the 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 base address register.

Current word count (CWC) register

It holds the count value of the number of bytes to be transferred by the DMA. Initially the count value is loaded to the CWC register from the base count register. After each byte transfer by the 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 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 BA-register is loaded in CA-register. If auto initialization is enabled in the mode register, then the content of BA-register is reloaded in the CA-register at the end of the DMA process.

TABLE - 7.6 : ADDRESS OF INTERNAL REGISTERS OF 8237

Name of the register	Operation performed	Binary address			
		A_3	A_2	A_1	A_0
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

Base word count (BWC) register

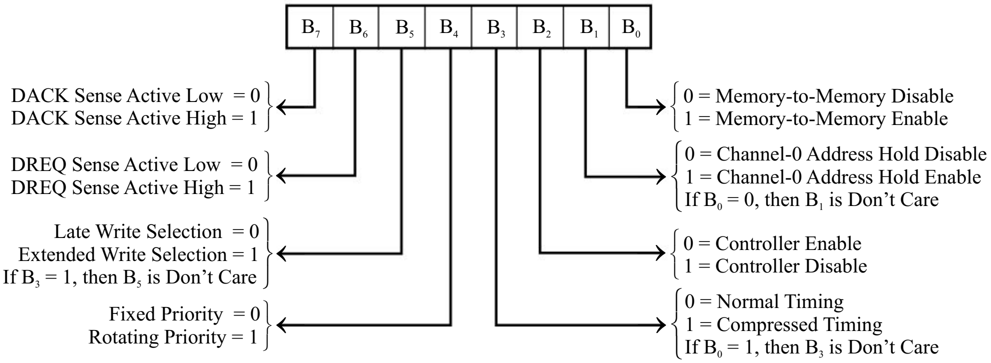
It is used to hold the count value for the number of bytes to be transferred by DMA. During the start of DMA process, the content of BWC register is loaded in CWC register. If auto initialization is enabled in mode register then the content of BWC register is reloaded in CWC register at the end of DMA 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. 7.17. During memory-to-memory DMA transfer, the channel-0 registers are used to hold the source address and the channel-1 registers are used to hold the destination address. The data transfer takes place via the temporary register in 8237. The number of bytes transferred is determined by the channel-1 count register.



B7 B6 B5 B4 B3 B2 B1 B0
DACK Sense Active Low = 0
DACK Sense Active High = 1
DREQ Sense Active Low = 0
DREQ Sense Active High = 1
Late Write Selection = 0
Extended Write Selection = 1
If B3 = 1, then B5 is Don't Care
Fixed Priority = 0
Rotating Priority = 1
0 = Memory-to-Memory Disable
1 = Memory-to-Memory Enable
0 = Channel-0 Address Hold Disable
1 = Channel-0 Address Hold Enable
If B0 = 0, then B1 is Don't Care
0 = Controller Enable
1 = Controller Disable
0 = Normal Timing
1 = Compressed Timing
If B0 = 1, then B3 is Don't Care

Fig. 7.17 : Format of control word to be loaded in command register.

The bit B_2 is used to turn ON/OFF the entire controller by the 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.

The bit B_4 is used to select fixed/rotating priority for DMA channels. In fixed priority, the 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 the 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 the rotating priority scheme the priorities of the channels from the highest to the lowest will be channel-0, channel-1, channel-3 and channel-2. Alternately, if the 8237 is programmed for fixed priority, then for the same situation after servicing channel-2, the priorities of the DMA channels from highest to lowest will be channel-0, channel-1, channel-2 and channel-3.

The bit B_5 is used to extend the timing of the write pulse when the IO devices require wider write pulse. This is possible only in normal timing. The bit B_6 and B_7 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 auto initialization.

The format of the control word to be loaded in the mode register is shown in Fig. 7.18. The control word of all the four mode registers are sent to the same internal address, but 8237 identifies the control word of a channel from the bits B_0 and B_1. The bits B_2 and B_3 are used to program the read/write/verify transfer. In read transfer the data is transferred from memory to the IO device. In write transfer the data is transferred from the IO device to the memory. Verification operations generate the DMA addresses without generating the DMA memory and IO control signals.

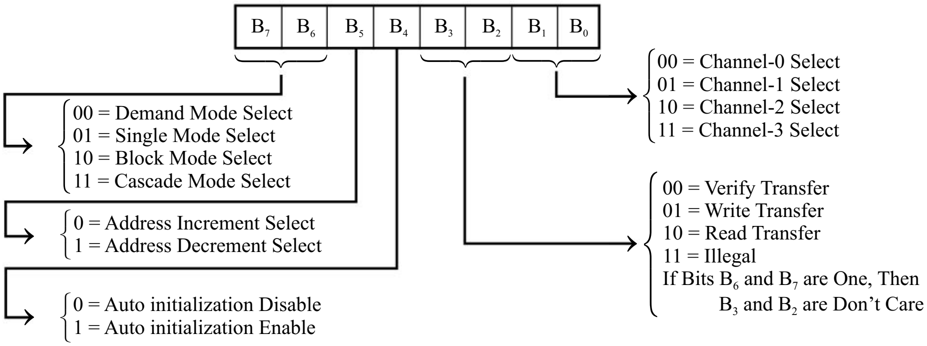


Fig. 7.18 : Format of control word to be loaded in the mode register.

The bit B_4 is used to enable/disable auto initialization of DMA channels. When it is enabled, the memory address and count value from base registers are loaded in the current registers after completion of DMA process, which are used to repeat the DMA process between the IO device and same block of memory.

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 the $\overline{\text{EOP}}$ pin of 8237 or until the DREQ input becomes inactive.

In single transfer mode, the 8237 releases the bus to the processor by deactivating the HOLD signal after transfer of each byte by the DMA. If the DREQ pin is held active, then 8237 will make a request for the DMA to the processor through the 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 the count register and then release the bus to the processor by deactivating the HOLD signal. In cascaded operation, the **H**old **Req**uest pin (HRQ) of one 8237 will be connected to the 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, then this channel cannot perform read/write/verify transfer.

Request register

It is used to request a DMA transfer via software. The format of the control word to be loaded in the request register is shown in Fig. 7.19. The bit B_0 and B_1 selects the channel in which DMA transfer is required and the bit B_2 is used to set/reset the DMA request.

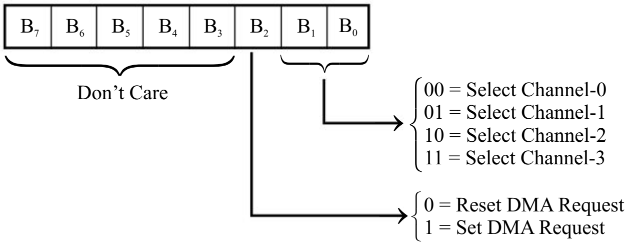


Fig. 7.19 : Format of control word to be loaded in a request register.

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 have to be unmasked by sending a control word to the mask register.

The mask register has two internal addresses. One address is used to set/reset the single mask bit (i.e., to mask/unmask one channel at a time) and the other address is used to set/reset all the mask bits (i.e., to mask/unmask all the channels). The format of the two control words for mask register are shown in Fig. 7.20.

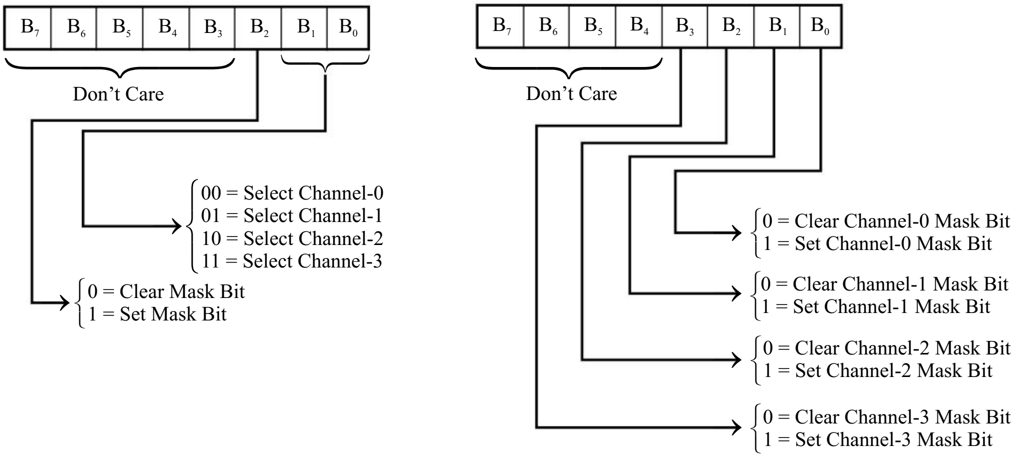

B7 B6 B5 B4 B3 B2 B1 B0
Don't Care
00 = Select Channel-0
01 = Select Channel-1
10 = Select Channel-2
11 = Select Channel-3
0 = Clear Mask Bit
1 = Set Mask Bit
B7 B6 B5 B4 B3 B2 B1 B0
Don't Care
0 = Clear Channel-0 Mask Bit
1 = Set Channel-0 Mask Bit
0 = Clear Channel-1 Mask Bit
1 = Set Channel-1 Mask Bit
0 = Clear Channel-2 Mask Bit
1 = Set Channel-2 Mask Bit
0 = Clear Channel-3 Mask Bit
1 = Set Channel-3 Mask Bit

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. 7.20 : 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 has reached their **T**erminal **C**ount (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. 7.21.

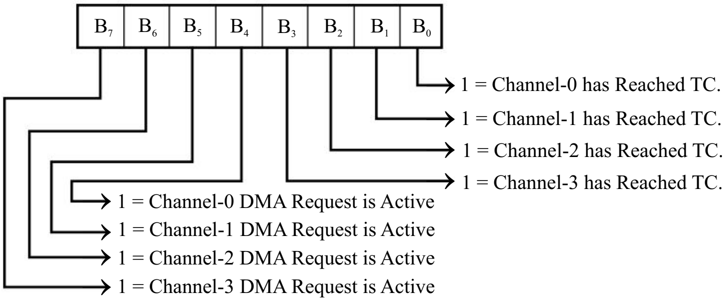

B_7
B_6
B_5
B_4
B_3
B_2
B_1
B_0
1 = Channel-0 has Reached TC.
1 = Channel-1 has Reached TC.
1 = Channel-2 has Reached TC.
1 = Channel-3 has Reached TC.
1 = Channel-0 DMA Request is Active
1 = Channel-1 DMA Request is Active
1 = Channel-2 DMA Request is Active
1 = Channel-3 DMA Request is Active

Fig. 7.21 : 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-7.6 for internal addresses of these commands.) We need not worry about the data sent to these port during write operation because the 8237 will ignore the data. The function 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 the read/write operation of the 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 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, clears first-last flip-flop and clears all internal register, except the mask register.
2. Send a control word to the command register to inform priority of DMA channels, normal/compressed timings, polarity of the DREQ and polarity of the DACK signals.
3. Write a mode word to the mode register of each channel to inform the DMA mode and the type of DMA transfer.
4. Send a "clear first-last flip-flop" software command to reset it to zero.
5. After ensuring that the 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 high byte.
6. Then write the 16-bit count value in the count register of each channel, by sending the low byte first and then the 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 values, because after each write operation it automatically toggles to keep track of low byte and high byte.
7. Finally send "the clear mask register" software command to enable all DMA channels. Now 8237 is ready to perform the DMA process.

Interfacing 8237 with 8086 Processor

A simple schematic for interfacing the 8237 with 8086 processor is shown in Fig. 7.22. The 8237 can be either memory-mapped or IO-mapped in the system. In the schematic shown in Fig. 7.22, the 8237 is IO-mapped in the system with even addresses. The chip select signals for IO-mapped devices are generated by using a 3-to-8 decoder. The address lines A_5, A_6 and A_7 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_0 and the control signal $M/\overline{IO}$ are used as enables for the decoder.

The address lines A_1 - A_4 of 8086 are connected to A_0 - A_3 of 8237 through a latch enabled by ALE to provide the internal addresses. The intermediate latch is necessary because the address lines A_0 - A_3 of 8237 should be connected to A_1 to A_4 of the address bus during the programming mode and they are connected to A_0 - A_3 of the address bus during the DMA mode. The IO addresses of the internal register of 8237 are listed in Table-7.7.

The 8237 supplies only lower 16 bits (A_0-A_{15}) of memory address during the DMA mode. The upper 4 bits of the memory address (A_{16}-A_{19}) should be generated by some means. For this a latch is provided in the system to hold the upper four bits of the DMA memory address. This latch is enabled by the chip select signal IOCS-7 and the IO address of this latch is also listed in Table-7.7. The processor can load the upper four bits of the DMA memory address in this latch while initializing the DMA controller.

The DB_0-DB_7 lines of 8237 are connected to the data bus lines D_0-D_7 for data transfer with the processor during the 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 the DMA mode on the external

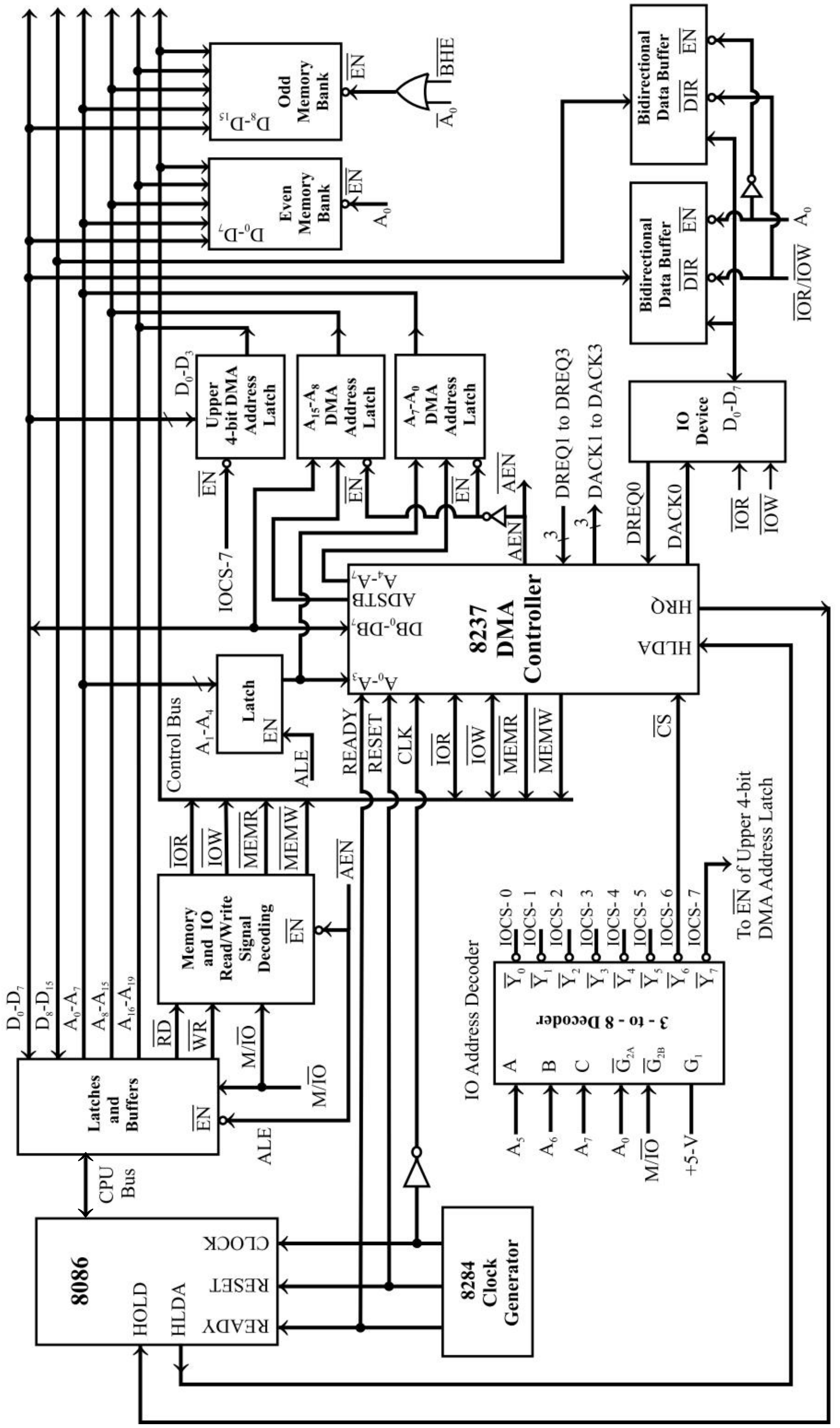


Fig. 7.22 : Interfacing of 8237 with 8086 processor.

latches. In the schematic shown in Fig. 7.22, two 8-bit latches are provided to hold the lower 16-bit memory address during DMA mode. During the 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 M/$\overline{IO}$ of the 8086 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-4, Fig. 4.17 for the logic circuit to generate separate read and write signals for memory and IO devices.)

The RESET, READY and clock signal for 8237 are provided by 8284 clock generator. The clock used for the processor should be inverted and supplied to 8237 for proper operation. The HRQ output of 8237 is connected to the HOLD input of 8086 in order to make a HOLD request to the processor. The HLDA output of 8086 is connected to HLDA input of 8237 in order to receive the acknowledge signal from the processor once the HOLD request is accepted.

TABLE - 7.7 : IO ADDRESSES OF 8237

Name of the internal register of 8237	Binary address								Hexa address
	Decoder input			Input to address pins of 8237				Decoder enable	
	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
Channel-0 Base and Current address register	1	1	0	0	0	0	0	0	C0
Channel-0 Base and Current word count register	1	1	0	0	0	0	1	0	C2
Channel-1 Base and Current address register	1	1	0	0	0	1	0	0	C4
Channel-1 Base and Current word count register	1	1	0	0	0	1	1	0	C6
Channel-2 Base and Current address register	1	1	0	0	1	0	0	0	C8
Channel-2 Base and Current word count register	1	1	0	0	1	0	1	0	CA
Channel-3 Base and Current address register	1	1	0	0	1	1	0	0	CC
Channel-3 Base and Current word count register	1	1	0	0	1	1	1	0	CE
Status/Command register	1	1	0	1	0	0	0	0	D0
Request register	1	1	0	1	0	0	1	0	D2
Write single mask register bit	1	1	0	1	0	1	0	0	D4
Mode register	1	1	0	1	0	1	1	0	D6
Clear first-last flip-flop	1	1	0	1	1	0	0	0	D8
Temporary register/Master clear	1	1	0	1	1	0	1	0	DA
Clear mask register	1	1	0	1	1	1	0	0	DC
Write all mask register bits	1	1	0	1	1	1	1	0	DE

Name of the external device	Binary address								Hexa address
	Decoder input			Unused address lines					
	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
Upper 4-bit DMA address latch	1	1	1	x	x	x	x	x	E0

Note : Here don't care "x" is considered as zero.

The IO device has to read/write data from/to both the even memory bank and the odd memory bank. In order to provide this facility two bidirectional data buffers are provided in the system. These buffers act as switches to connect the D_0-D_7 of the IO device to D_0-D_7 of the system bus for even addresses and to D_8-D_{15} of the system bus for odd addresses. The address A_0 is used to enable these buffers.

The buffer which is enabled by A_0 is used for read/write data from/to the even memory bank via the low order data bus. The buffer which is enabled by $\bar{A}_0$ is used for read/write data from/to the odd memory bank via the high order data bus. The direction control for the bidirectional buffer can be provided by using either $\overline{\text{IOR}}$ or $\overline{\text{IOW}}$ signal.

DMA Operation in 8086 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 the DMA request input from the IO devices. When the 8237 deducts a valid DMA request, it performs the following activity :

1) When the 8237 receives a DMA request from a peripheral it sends a hold request to the 8086 processor (provided the channel should be enabled and there should not be any pending higher priority DMA request).
2) When the 8086 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 8086 sends an acknowledge signal to the 8237.
3) On receiving an acknowledge from the 8086, the 8237 will send an acknowledge to the peripheral device which requested DMA.
4) The 8237 asserts AEN **high**, which enable DMA memory address latches and disables the processor address latch.
5) Then the 8237 outputs the low byte address on the A_0-A_7 lines and high byte address on the DB_0 to DB_7 lines. Also the control signal ADSTB is asserted **high** to latch this address into the 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, steps 4,5 and 6 are repeated again and again until the count is zero. In demand transfer mode, 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 deactivates the hold request to the processor after one-byte of transfer by the DMA.

7.2.5 DMA Controller - INTEL 8257

The DMA controller-8257 has been developed for 8085/8086/8088 microprocessor-based systems. It is a device dedicated to perform a high speed data transfer between memory and IO device. The 8257 has four channels and so it can be used to provide DMA to four IO devices. It cannot be connected in cascade like the 8237 and it has less features than the 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 the number of bytes to be read/write 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 **D**ual **I**n-line **P**ackage (DIP). The pin configuration of the 8257 is shown in Fig. 7.23. A brief description about the pins and signals of the 8257 are listed in Table -7.8.

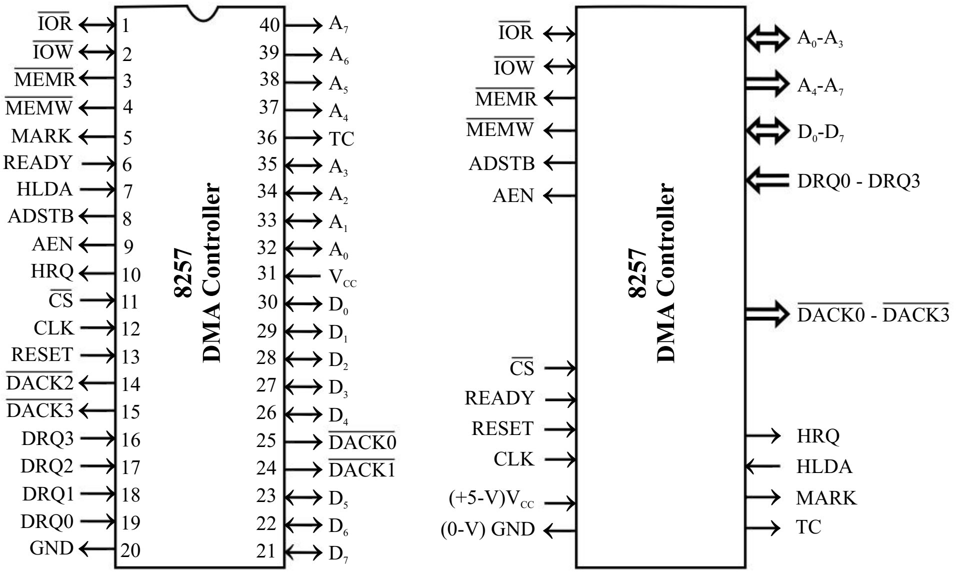


Fig. 7.23 : Pin configuration of an 8257.

TABLE - 7.8 : PIN DESCRIPTION OF 8257

Pin	Description
CLK	Clock input to 8257. The maximum clock frequency is 5 MHz. In an 8086/8088 system, the processor clock signal is inverted and applied to the CLK of 8257.
$\overline{\text{CS}}$	Logic **low** chip select signal. It is the input signal to select 8257 during the programming mode.
RESET	Reset input to 8257. Connected to the system reset, when the RESET signal goes **high** 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 **low**, the 8257 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 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 sent by the processor to inform the acceptance of a 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 **low** output signals from 8257 to the IO devices to inform the acceptance of a DMA request.

Table - 7.8 continued...

Pin	Description
D_0 - D_7	Data bus lines. These pins are used for data transfer between the processor and the DMA controller during the programming mode. During the DMA mode, these lines are used as multiplexed high order address and data lines.
$\overline{IOR}$	Bidirectional IO read control signal. It is the input control signal for reading DMA controller during the programming mode and the output control signal for reading IO device during the DMA (memory) write cycle.
$\overline{IOW}$	Bidirectional IO write control signal. It is the input control signal for writing the DMA controller during the programming mode and output control signal for writing the IO device during the 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 the DMA mode.
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 - D_0 lines by 8257 during the DMA mode into an external latch.
$\overline{MEMR}$	Memory read control signal. It is an output control signal issued during the DMA read operation.
$\overline{MEMW}$	Memory write control signal. It is an output control signal issued during the DMA write operation.

Features of 8257

- It has four independent DMA channels to service four IO devices.
- Each channel is independently programmable to transfer up to 64 kb of data by DMA.
- Each channel can independently perform read transfer, write transfer and verify transfer.

Functional Block Diagram of 8257

The functional block diagram of 8257 is shown in Fig. 7.24. The functional blocks of 8257 are the data bus buffer, read/write logic, control logic and four numbers of the DMA channels.

Each channel has two programmable 16-bit registers. One register is used to program the starting address of the memory location for DMA data transfer and another register is used to program a 14-bit count value and a 2-bit code for the 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 the count register is shown in Fig. 7.25(a).

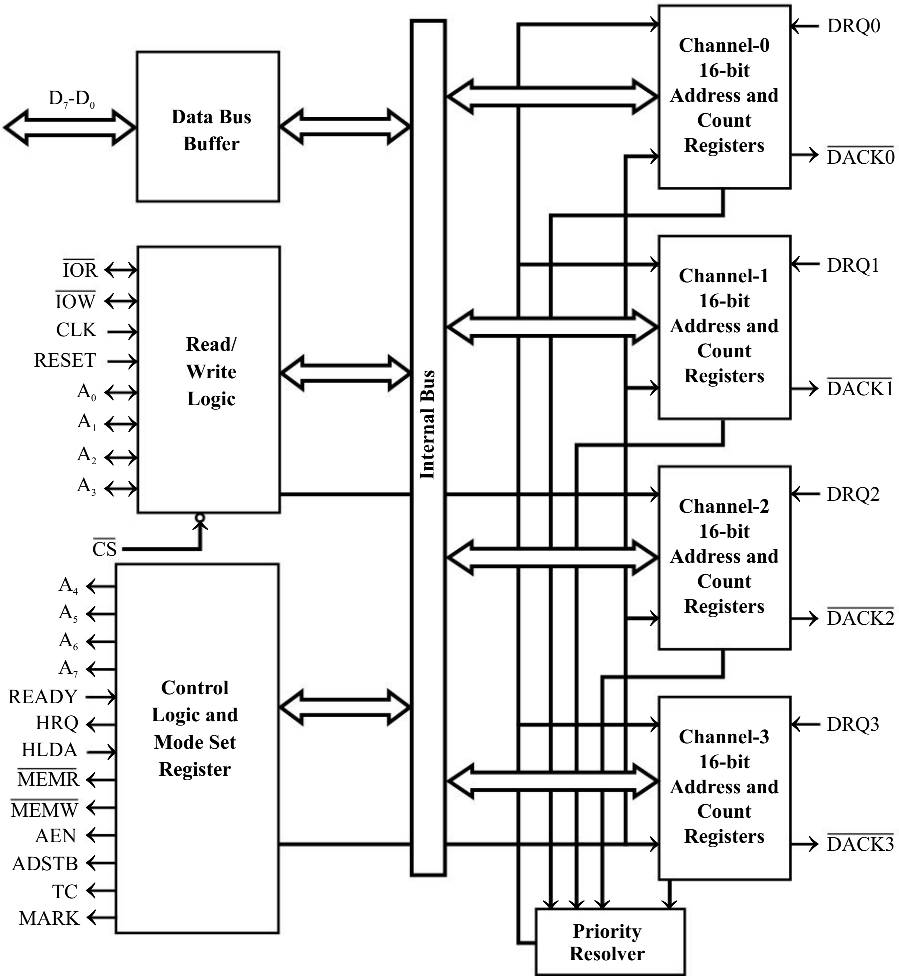

D7-D0
Data Bus Buffer
IOR
IOW
CLK
RESET
A0
A1
A2
A3
Read/ Write Logic
CS
A4
A5
A6
A7
READY
HRQ
HLDA
MEMR
MEMW
AEN
ADSTB
TC
MARK
Control Logic and Mode Set Register
Internal Bus
Channel-0 16-bit Address and Count Registers
DRQ0
DACK0
Channel-1 16-bit Address and Count Registers
DRQ1
DACK1
Channel-2 16-bit Address and Count Registers
DRQ2
DACK2
Channel-3 16-bit Address and Count Registers
DRQ3
DACK3
Priority Resolver

Fig. 7.24 : Functional block diagram of DMA controller 8257.

In read transfer, the data is transferred from the memory to the IO device. In write transfer, the data is transferred from the 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 the slave/programming mode by sending a 4-bit address to 8257 through A_0 - A_3 lines. The internal addresses of the registers of 8257 are listed in Table-7.9.

TABLE - 7.9 : 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

While programming 16-bit register the low byte has to be send first and then the high byte. Internally, the loading of low byte and high byte into 16-bit register are taken care 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 the mode set register of the 8257 is shown in Fig. 7.25(b). The bits B_0, B_1, B_2 and B_3 of the mode set register are used to enable/disable channel-0, 1,2 and 3 respectively. A one in these bit position will enable a particular channel and a zero will disable it.

In the mode set register, if bit B_4 is set to one, then the channels will have rotating priority and if it is zero then the channels will have fixed priority. In rotating priority, after servicing a channel its priority is made as lowest. In fixed priority, channel-0 has the highest priority and channel-2 has the lowest priority.

In mode set register, if bit B_5 is set to one, then the timing of the write signals ($\overline{\text{MEMW}}$ and $\overline{\text{IOW}}$) will be extended and if bit B_6 is set to one then the DMA operation is stopped at the terminal count. Bit B_7 is used to select the auto load feature for DMA channel-2. When bit B_7 is set to one, the content of channel-3 count and address registers are loaded in channel-2 count and address registers respectively whenever channel-2 reaches terminal count. Therefore, when this mode is activated the number of channels available for DMA reduces from four to three.

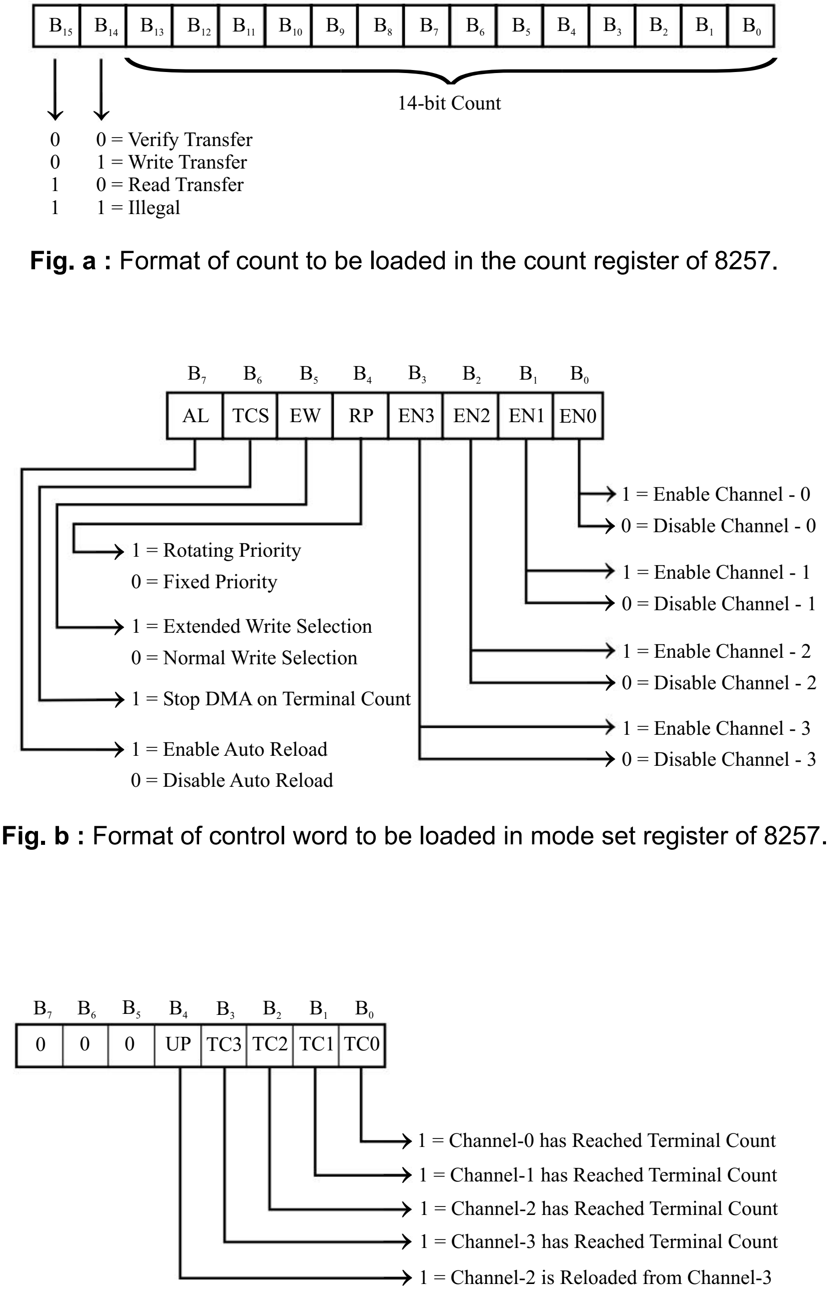


Fig. a : Format of count to be loaded in the count register of 8257.

Fig. b : Format of control word to be loaded in mode set register of 8257.

B7 B6 B5 B4 B3 B2 B1 B0
0 0 0 UP TC3 TC2 TC1 TC0
1 = Channel-0 has Reached Terminal Count
1 = Channel-1 has Reached Terminal Count
1 = Channel-2 has Reached Terminal Count
1 = Channel-3 has Reached Terminal Count
1 = Channel-2 is Reloaded from Channel-3

Fig. c : Status register of 8257.

Fig. 7.25 : Format of registers of 8257.

The format of status register of 8257 is shown in Fig. 7.25(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 the status register indicate the terminal count status of channels-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 the microprocessor. Bit B_4 of the 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.

Note : Interfacing of 8257 with 8086 processor will be similar to that of interfacing 8237 with 8086.

DMA Operation in 8086 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 the DMA request input from the IO devices. Whenever a DMA request is made by an IO device, the DMA operation is performed. The various steps of the DMA operation are as follows :

1. When a peripheral device require a DMA, it will assert the 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 (**H**OLD **Req**uest) as **high.**
3. When the 8086 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 the HLDA signal as **high.**
4. When the 8257 receives an acknowledge signal from 8086, the 8257 will send an acknowledge signal to the peripheral which requested the DMA, by asserting the $\overline{\text{DACK}}$ 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 the peripheral to the memory.
7. After performing one-byte transfer steps 5 and 6 are repeated again and again, until the terminal count (i.e., until the count reaches zero).

7.3 SERIAL DATA COMMUNICATION INTERFACE

7.3.1 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 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.

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 the processor and commercial radio stations are examples of simplex transmission.

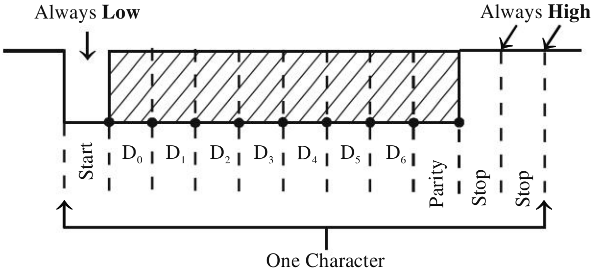


Fig. 7.26 : Bit format used for sending asynchronous serial data.

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 a 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 blocks 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 identifies its end. Since each data is individually identified, data can be sent at any time. Figure 7.26 shows the bit format often used for transmitting 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 systems 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 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 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 **mo**dulator-**dem**odulator.

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 are transmitted as voltage, the commonly used standard is known as RS-232C. It was developed by **E**lectronics **I**ndustries **A**ssociation (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-7.10. 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 the RS-232C serial bus is terminated on a 25-pin connector.)

The voltage levels for all RS-232C signals are :

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**).

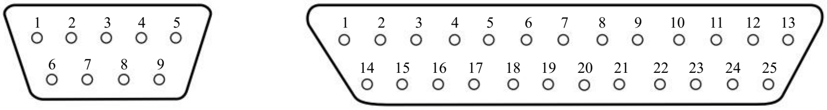
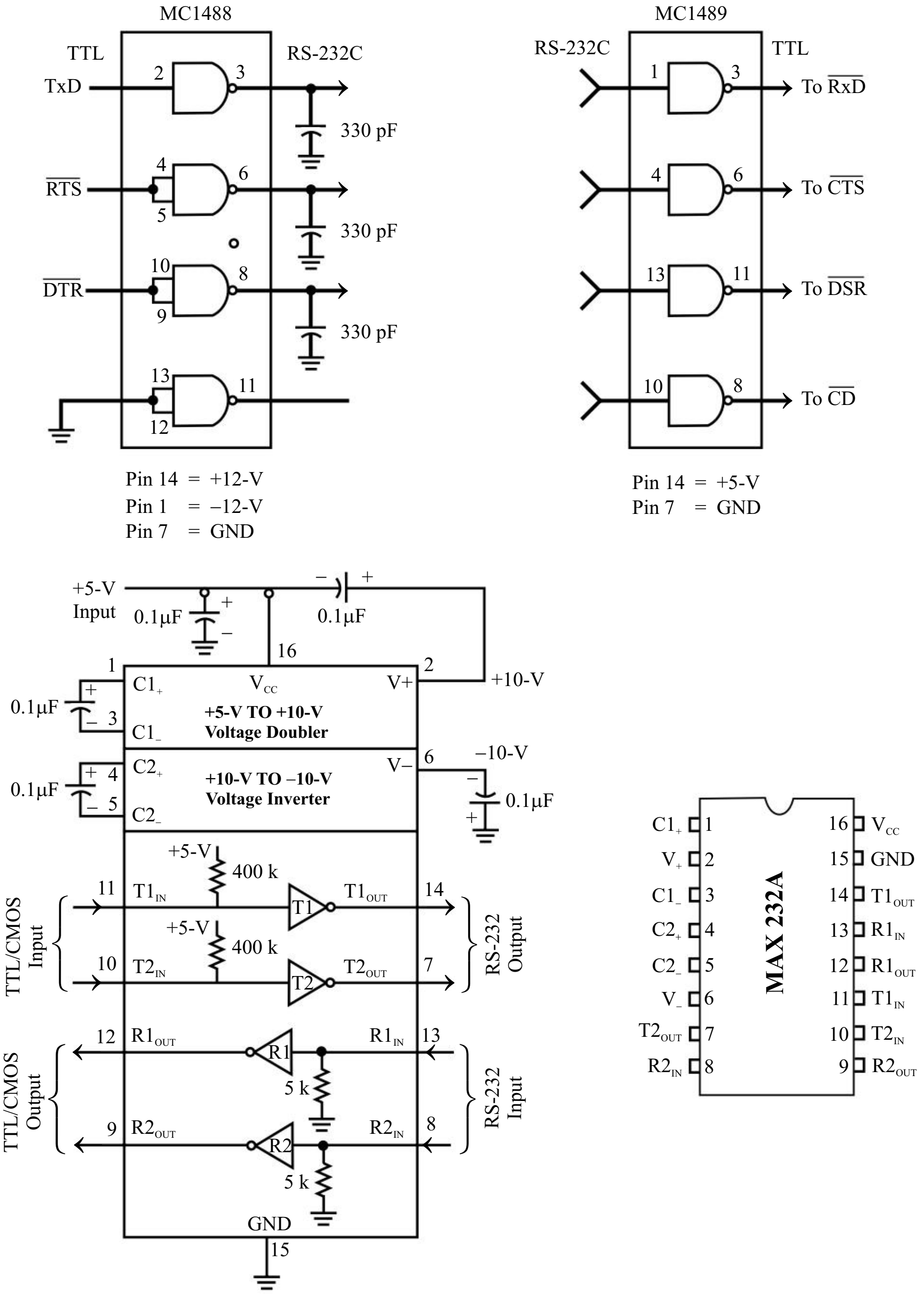


Fig. a : 9-pin D-type connector. **Fig. b :** 25-pin D-type connector.

Fig. 7.27 : Connectors used for terminating RS-232C bus.

TABLE - 7.10 : RS-232C PIN NAMES AND SIGNAL DESCRIPTION

Pin number	Common name	RS-232C name	Description	Signal direction on DCE
1	-	AA	Protective Ground	-
2	TxD	BA	Transmitted Data	IN
3	RxD	BB	Received Data	OUT
4	$\overline{RTS}$	CA	Request to send	IN
5	$\overline{CTS}$	CB	Clear to send	OUT
6	$\overline{DSR}$	CC	Data Set ready	OUT
7	GND	AB	Signal ground (Common return)	-
8	$\overline{CD}$	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	$\overline{DTR}$	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	-



Typical circuit connection of MAX 232A

> ***Note*** : *1. For MAX 232 all capacitors should be 1 mF.*
>
> *2. The voltage rating of all capacitors should be above 10-V.*

Fig. 7.28 : TTL to RS-232C and RS-232C to TTL signal conversion.

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.

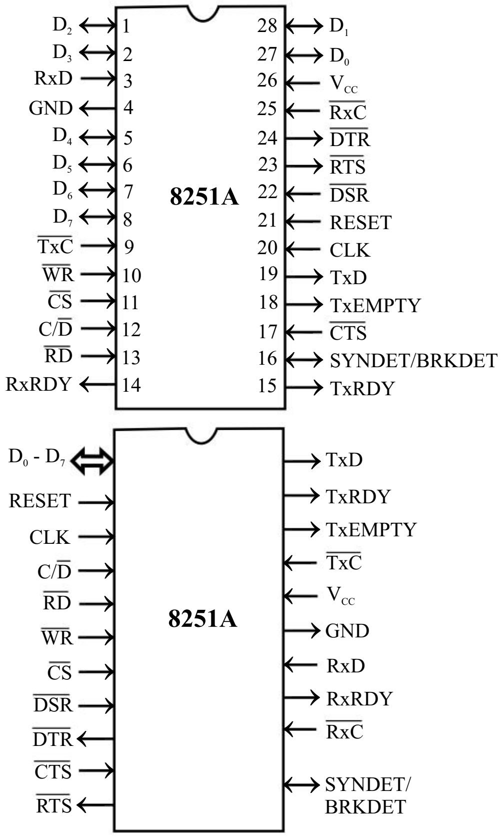
(Max 232 is equivalent to a combination of MC1488 and MC1489 in single IC.)

The signal level conversion using the above converters are shown in Fig. 7.28.

7.3.2 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. 7.29.

The functional block diagram of 8251A is shown in Fig. 7.30. The block diagram shows five sections, they are read/write control logic, transmitter, receiver, data bus buffer and modem control.



Pin	Description
D_0-D_7	Parallel data
C/$\overline{D}$	Control register or Data buffer select
$\overline{RD}$	Read control
$\overline{WR}$	Write control
$\overline{CS}$	Chip Select
CLK	Clock pulse (TTL)
RESET	Reset
$\overline{TxC}$	Transmitter Clock
TxD	Transmitter Data
$\overline{RxC}$	Receiver Clock
RxD	Receiver Data
RxRDY	Receiver Ready
TxRDY	Transmitter Ready
$\overline{DSR}$	Data Set Ready
$\overline{DTR}$	Data Terminal Ready
SYNDET/ BRKDET	Synchronous Detect / Break Detect
$\overline{RTS}$	Request To Send Data
$\overline{CTS}$	Clear To Send Data
TxEMPTY	Transmitter Empty
V_{CC}	Supply (+5-V)
GND	Ground (0-V)

Fig. 7.29 : Pin description of 8251A.

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.

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 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, $\overline{TxC}$ 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 the serial data and convert it 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 the 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 $\overline{RxC}$ 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 the 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 times 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 the synchronous character. If the 8251A finds a synchronous character in the incoming string of data bits then it asserts SYNDET signal as **high**.

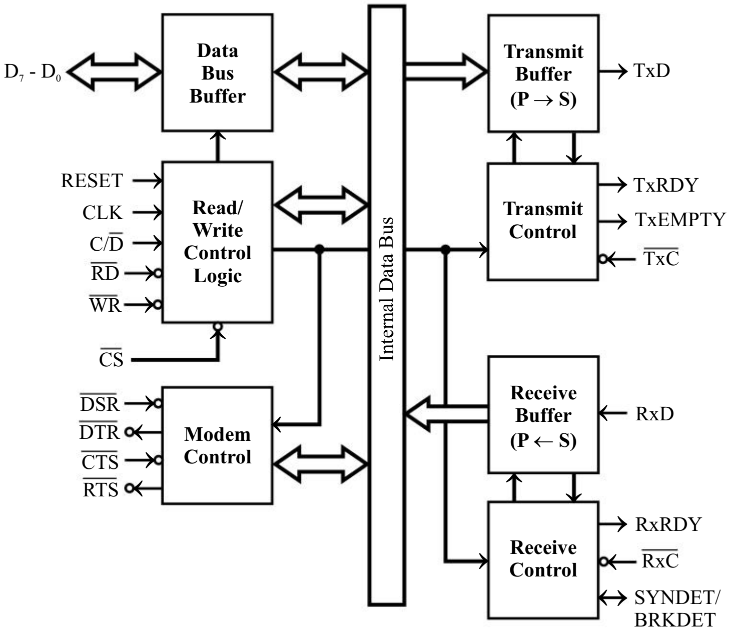


Fig. 7.30 : The functional block diagram of 8251A - USART.

MODEM Control

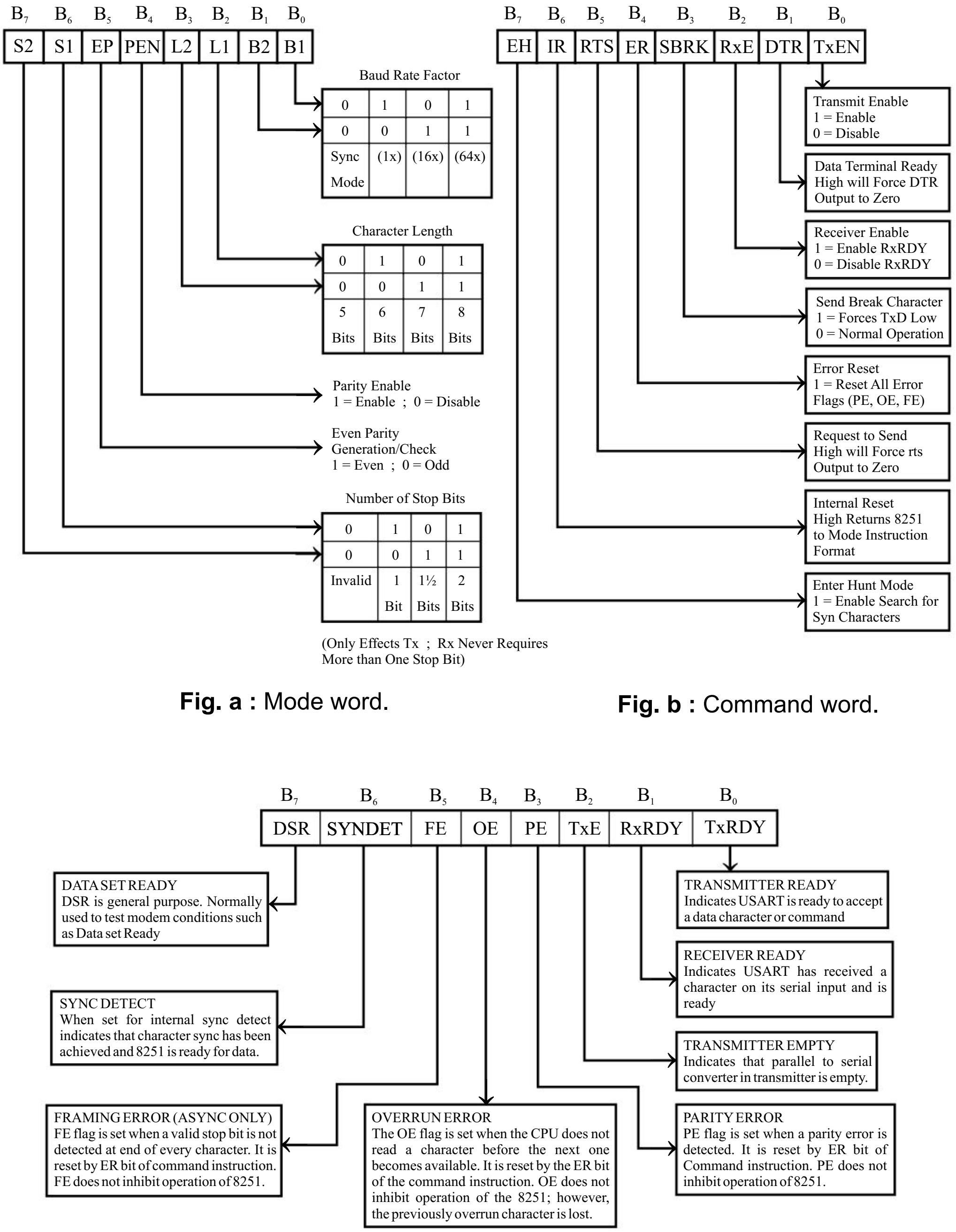
The MODEM control unit allows to interface a MODEM to 8251A and to establish data communication through MODEM over telephone lines. This unit takes care of handshake signals for MODEM interface.

Programming the 8251A

The 8251A is programmed by sending the 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. 7.31.

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 the 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 the baud rate factor is 16x, then the baud rate is clock frequency divided by 16. If the baud rate factor is 1x, then the baud rate is given by clock frequency.



Baud Rate Factor

0	1	0	1
0	0	1	1
Sync Mode	(1x)	(16x)	(64x)

Character Length

0	1	0	1
0	0	1	1
5 Bits	6 Bits	7 Bits	8 Bits

Parity Enable
1 = Enable ; 0 = Disable

Even Parity
Generation/Check
1 = Even ; 0 = Odd

Number of Stop Bits

0	1	0	1
0	0	1	1
Invalid	1 Bit	1½ Bits	2 Bits

(Only Effects Tx ; Rx Never Requires
More than One Stop Bit)

Fig. a : Mode word.

Transmit Enable
1 = Enable
0 = Disable

Data Terminal Ready
High will Force DTR
Output to Zero

Receiver Enable
1 = Enable RxRDY
0 = Disable RxRDY

Send Break Character
1 = Forces TxD Low
0 = Normal Operation

Error Reset
1 = Reset All Error
Flags (PE, OE, FE)

Request to Send
High will Force rts
Output to Zero

Internal Reset
High Returns 8251
to Mode Instruction
Format

Enter Hunt Mode
1 = Enable Search for
Syn Characters

Fig. b : Command word.

DATA SET READY
DSR is general purpose. Normally used to test modem conditions such as Data set Ready

SYNC DETECT
When set for internal sync detect indicates that character sync has been achieved and 8251 is ready for data.

FRAMING ERROR (ASYNC ONLY)
FE flag is set when a valid stop bit is not detected at end of every character. It is reset by ER bit of command instruction. FE does not inhibit operation of 8251.

OVERRUN ERROR
The OE flag is set when the CPU does not read a character before the next one becomes available. It is reset by the ER bit of the command instruction. OE does not inhibit operation of the 8251; however, the previously overrun character is lost.

TRANSMITTER READY
Indicates USART is ready to accept a data character or command

RECEIVER READY
Indicates USART has received a character on its serial input and is ready

TRANSMITTER EMPTY
Indicates that parallel to serial converter in transmitter is empty.

PARITY ERROR
PE flag is set when a parity error is detected. It is reset by ER bit of Command instruction. PE does not inhibit operation of 8251.

Fig. c : Status word.

Fig. 7.31 : Format of 8251A mode, command and status words.

Interfacing 8251A to 8086

A simple schematic for interfacing the 8251A with 8086 processor is shown in Fig. 7.32. The 8251A can be either memory-mapped or IO-mapped in the system. In the schematic shown in Fig. 7.32, the 8251A is IO-mapped in the system, with even addresses. The chip select signals for IO-mapped devices are generated by using a 3-to-8 decoder. The address lines A_5, A_6 and A_7 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_0 and the control signal $M/\overline{IO}$ are used as enable for the decoder.

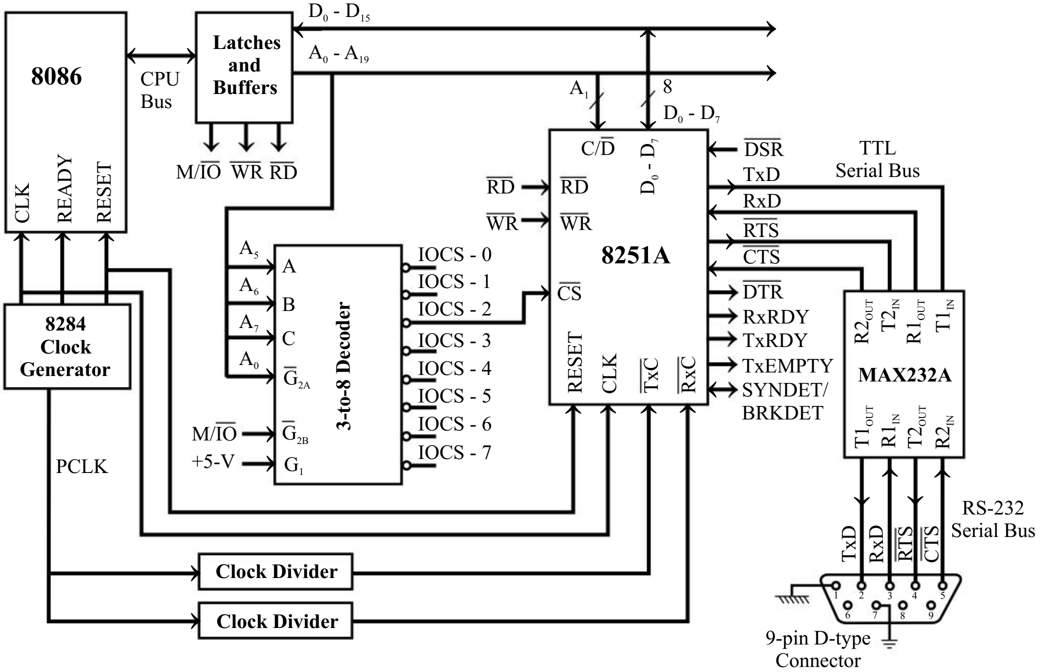


Fig. 7.32 : Interfacing of 8251A to 8086 microprocessor.

TABLE - 7.11 : IO ADDRESSES OF 8251A

Internal device of 8251A	Binary address								Hexa address
	Decoder input			Input to address pin of 8251				Decoder enable	
	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
Data buffer	0	1	0	x	x	x	0	0	40
Control register	0	1	0	x	x	x	1	0	42

Note : The don't care "x" is considered as zero.

The address line A_1 of 8086 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-7.11. 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 8284 clock generator. Here the processor clock is directly connected to 8251A. This clock controls the parallel data transfer between the processor and 8251A.

The **P**eripheral **Cl**oc**k** (PCLK) supplied by 8284, is divided by suitable clock dividers and then used as clock for serial transmission and reception ($\overline{\text{TxC}}$ and $\overline{\text{RxC}}$). In 8251A, the transmission and reception baud rates can be different or same. Usually a programmable timer, 8254 (which is discussed in Section 7.5) is used to divide the PCLK, and supply to $\overline{\text{TxC}}$ and $\overline{\text{RxC}}$ 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 a 9-core cable.

The signals TxEMPTY, TxRDY and RxRDY can be used as interrupt signals to initiate the interrupt driven data transfer scheme between the processor and 8251A.

7.4 KEYBOARD AND DISPLAY INTERFACE

7.4.1 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. 7.33. 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.

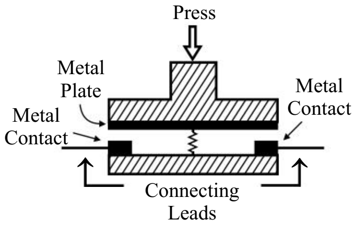


Fig. 7.33 : A representation of keyboard switch.

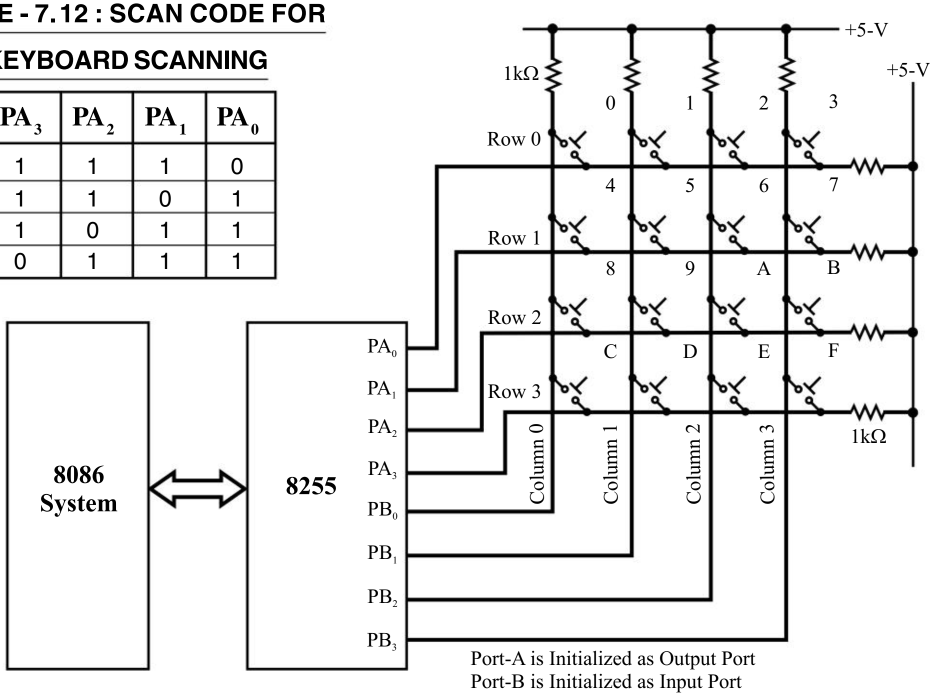
The three major tasks mentioned above can be performed by the software, when a keyboard is connected through ports to 8086 processor. Consider a simple keyboard in which the keys are arranged in rows and columns as shown in Fig. 7.34. The rows are connected to port-A lines of 8255. The columns are connected 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, and 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 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-7.12 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 key. When this bounce occurs, it may appear to the microcomputer that the same key has been actuated several times instead of just once. This problem can be eliminated by scanning the row in which the key press is deducted after 10 to 20 milliseconds 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.

TABLE - 7.12 : SCAN CODE FOR KEYBOARD SCANNING

PA_3	PA_2	PA_1	PA_0
1	1	1	0
1	1	0	1
1	0	1	1
0	1	1	1



Port-A is Initialized as Output Port
Port-B is Initialized as Input Port

Fig. 7.34 : Keyboard interfacing using ports.

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 a hexadecimal or an 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. 7.35.

In keyboard interfacing there are two methods of handling multiple key presses 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 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 generate a 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.

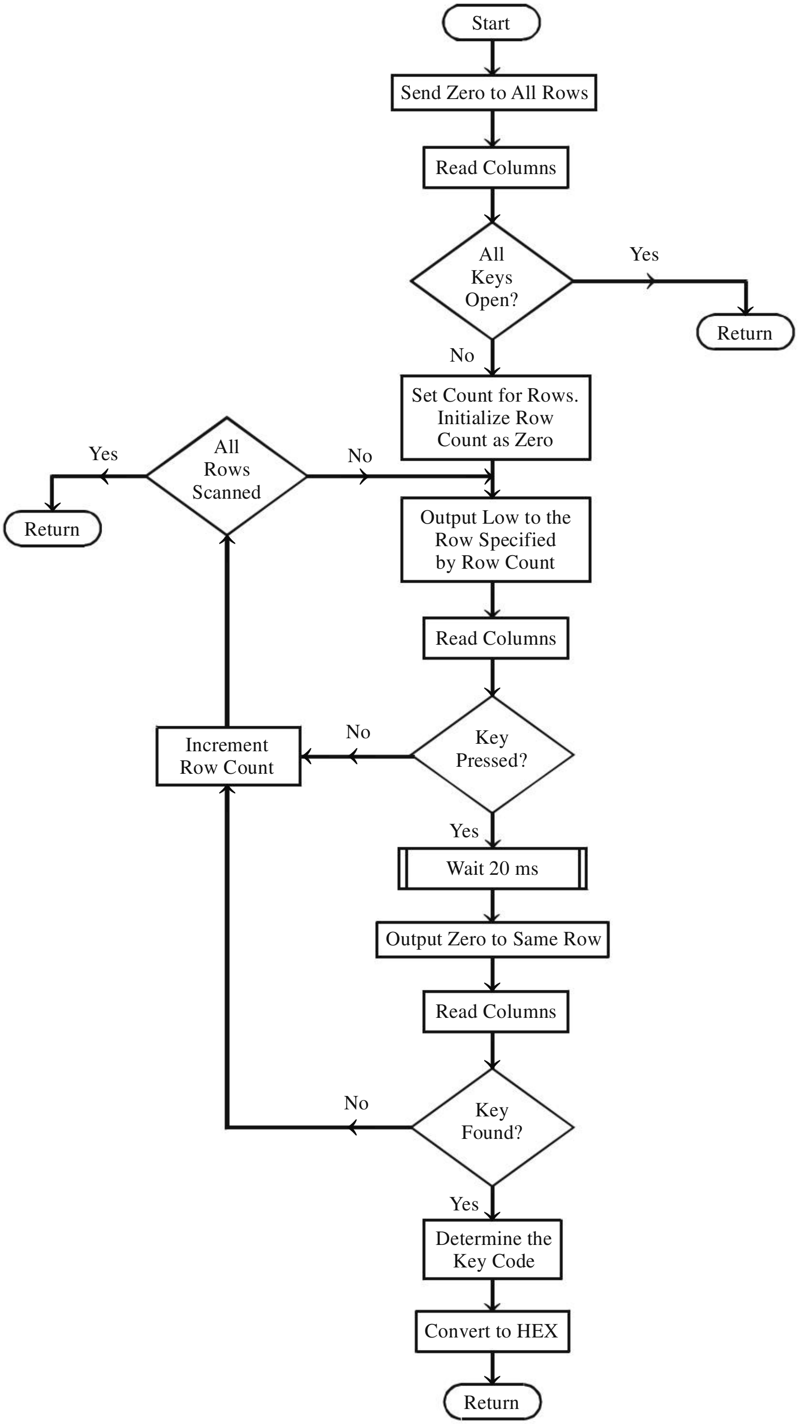


Fig. 7.35 : Flowchart for keyboard scanning subroutine.

7.4.2 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 a common cathode 7-segment LED are shown in Fig. 7.36.

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 the common anode 7-segment LED are shown in Fig. 7.37.

In a 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 the anode and a **low**(logic-0/0-V) to cathode. The alphabetic/numeric characters can be displayed on the 7-segment LED by forward biasing the appropriate segments.

In a common cathode 7-segment LED, the common point is tied to logic-0. To display a character, logic-1 is applied to the 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 the common cathode 7-segment LED are listed in Table-7.13.

In a common anode 7-segment LED, the common point is tied to logic-1. To display a character, logic-0 is applied to the cathode of segments which has to emit light and logic-1 is applied to the cathode of segments which should not emit light. The binary and hex codes for displaying the decimal digits 0 to 9 in the common anode 7-segment LED are listed in Table-7.14.

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. 7.38, to interface a common anode 7-segment LED to 8086 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.

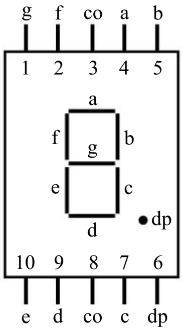


Fig. a : Pin configuration.

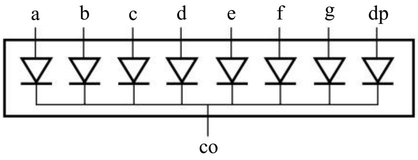


Fig. b : Internal connection.

co - common cathode
dp - anode of dot point
a, b, c, d, e, f, g - anodes of segments

Fig. 7.36 : Common cathode 7-segment LED.

TABLE - 7.13 : 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

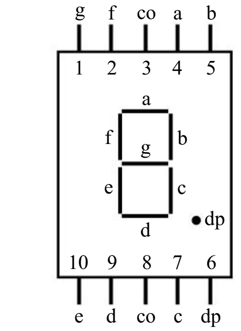


Fig. a : Pin configuration.

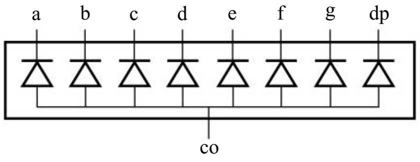


Fig. b : Internal connection.

co - common anode
dp - cathode of dot point
a, b, c, d, e, f, g - cathodes of segments

Fig. 7.37 : Common anode 7-segment LED.

TABLE - 7.14 : 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

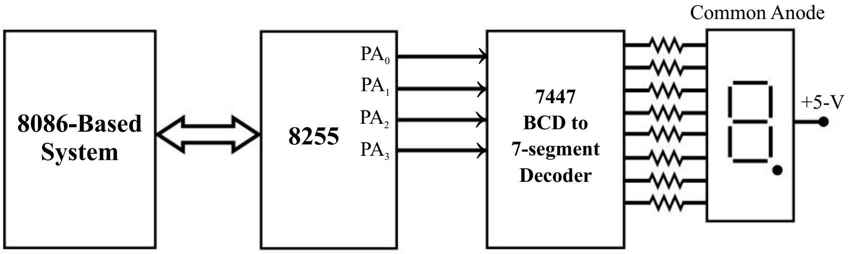


Fig. 7.38 : 7-Segment LED display using port.

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 the 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.

The advantages of a multiplexed display are the following:

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 7.39 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 the first output from port-A to the 7447. The 7447 outputs the corresponding 7-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** turns 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.

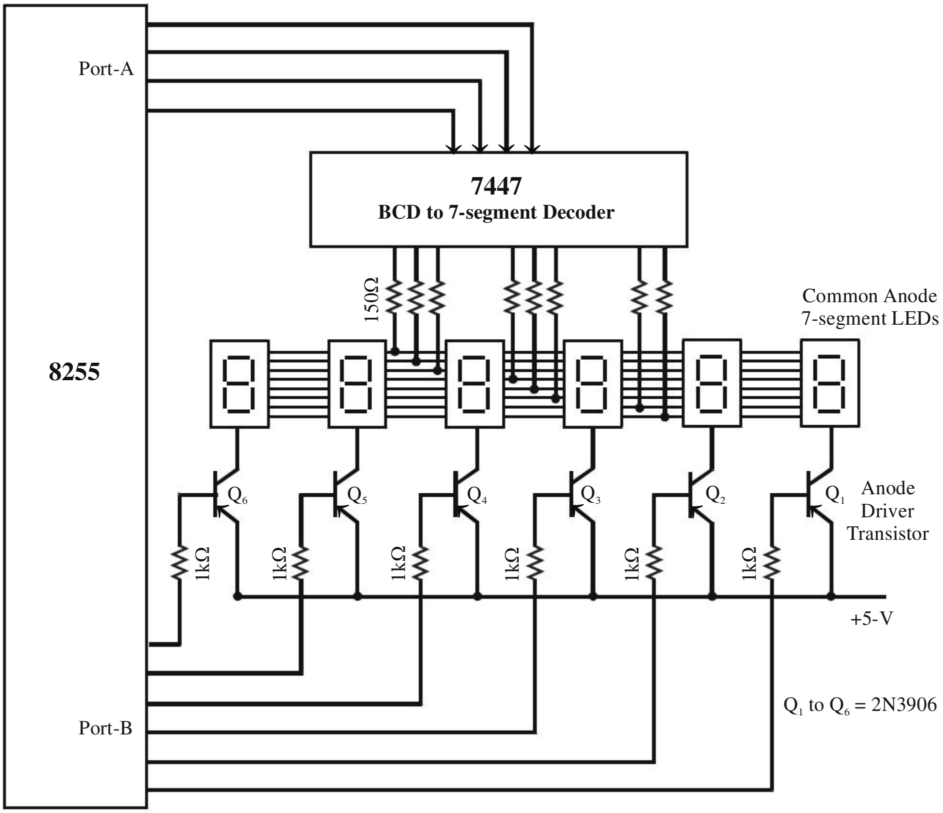


Fig. 7.39 : A schematic diagram of a multiplexed display using ports.

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 millisecond, 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 the 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 a multiplexed display is shown in Fig. 7.40.

The greatest 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 of power and parts.

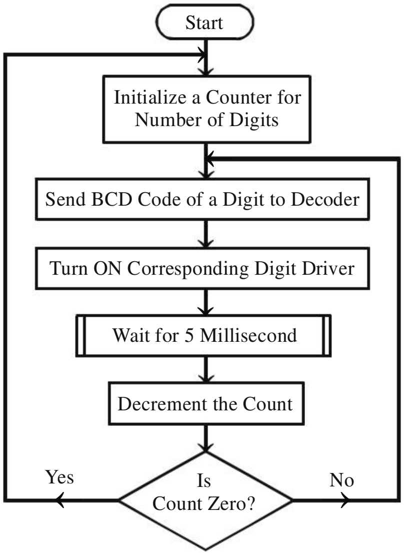

Start
Initialize a Counter for Number of Digits
Send BCD Code of a Digit to Decoder
Turn ON Corresponding Digit Driver
Wait for 5 Millisecond
Decrement the Count
Is Count Zero?
Yes
No

Fig. 7.40 : Flowchart for a multiplexed display.

7.4.3 Keyboard/Display Controller - INTEL 8279

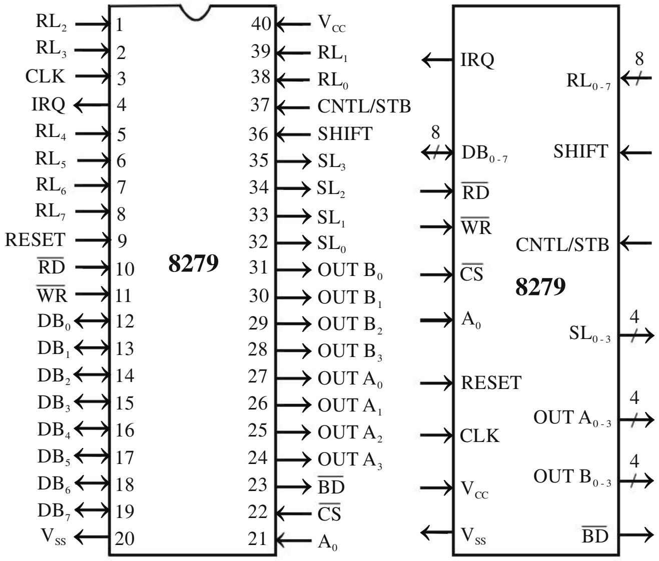
The INTEL 8279 is a dedicated controller specially developed for interfacing keyboard and display devices to 8085/8086/8088 microprocessor-based system. It relieves the processor from the time consuming task like keyboard scanning and display refreshing.

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 × 8 matrix array of key switches). The 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.

The 8279 provides a multiplexed interface for 7-segment LEDs and other popular display devices. It consist of 16 × 8 display RAM which can also be organized into dual 16 × 4 RAM. The CPU have 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.



Pin	Description
DB_{0-7}	Data Bus (Bidirectional)
CLK	Clock Input
RESET	Reset Input
$\overline{CS}$	Chip Select
$\overline{RD}$	Read Control
$\overline{WR}$	Write Control
A_0	Internal Address
IRQ	Interrupt Request Output
SL_{0-3}	Scan Lines
RL_{0-7}	Return Lines
SHIFT	Shift Input
CNTL/STB	Control/Strobe Input
OUT A_{0-3}	Display (A) Output
OUT B_{0-3}	Display (B) Output
$\overline{BD}$	Blank Display Output

Fig. 7.41 : Pin description of 8279.

The 8279 is a 40-pin IC available in DIP (**D**ual **I**n-line **P**ackage). The pin configuration of 8279 is shown in Fig. 7.41. 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 the 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. 7.42. The four major sections of 8279 are keyboard, scan, display and CPU interface.

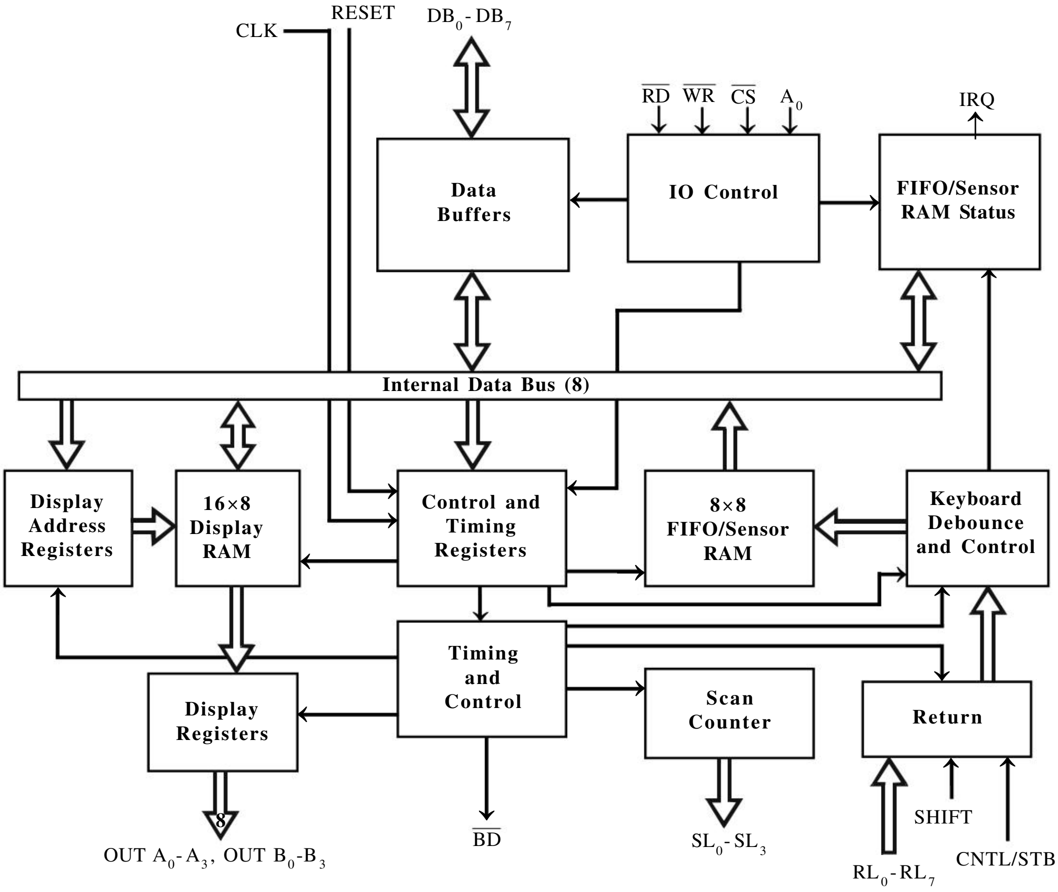


Fig. 7.42 : Block diagram of 8279.

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 inputs : 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 the FIFO.

The keyboard section also has an 8×8 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 the FIFO. The format of the keycode entry in FIFO for the scan keyboard mode is shown in Fig. 7.43.

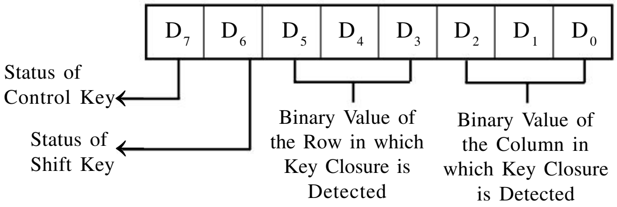

D_7
D_6
D_5
D_4
D_3
D_2
D_1
D_0
Status of Control Key
Status of Shift Key
Binary Value of the Row in which Key Closure is Detected
Binary Value of the Column in which Key Closure is Detected

Fig. 7.43 : Keycode entry in FIFO for scan keyboard mode.

In a 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 a 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 16×8 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 a decoded scan mode, the output of the scan lines will be similar to a 2-to-4 decoder. In encoded scan mode, the output of the scan lines will be binary count, and so an external decoder should be used to convert the binary count to the 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 the digit drivers of a multiplexed display to turn ON/OFF.

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 CD bits are available in this command to clear all rows of the Display RAM to a selectable blanking code as follows.

CD **CD CD**

CD	CD	CD	
	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

D	D		
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

K	K	K	
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 PPPPP 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. 7.44 : 8279 Command word formats.

CPU interface section

The CPU interface section takes care of the 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 the processor.

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. 7.44.

7.4.4 Keyboard and Display Interface using 8279

In a microprocessor-based system, when keyboard and 7-segment LED displays are interfaced using ports or latches then the processor has to carry the following task :

- 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 processors task is to program the 8279 by sending the control words and load the display code in the 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 the interrupt so that the processor can read the keycode from the FIFO of 8279.

A typical hexa keyboard and 7-segment LED display interfacing circuit using 8279 is shown in Fig. 7.45. The circuit can be used in 8086 microprocessor system and consists of 16 numbers of hexa-keys and 8 numbers of 7-segment LEDs. The 7-segment LEDs can be used to display eight digit alphanumeric character.

The 8279 can be either memory-mapped or IO-mapped in the system. In the circuit shown in Fig. 7.45, the 8279 is IO-mapped. The address line A_1 of the system is used as A_0 of 8279. The clock signal for 8279 is obtained by dividing the PCLK (**P**eripheral **cl**oc**k**) of 8284 by a clock divider circuit.

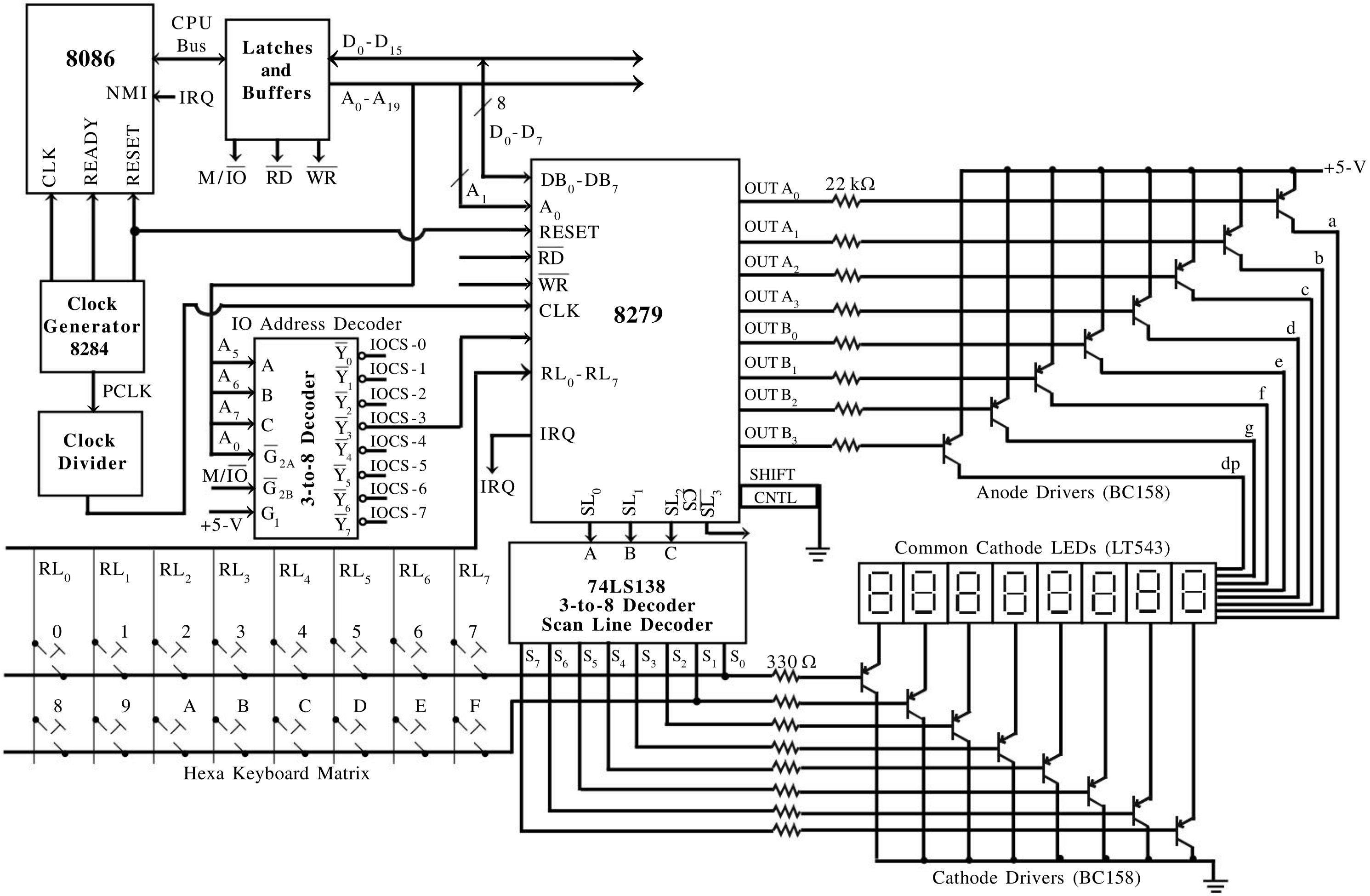


Fig. 7.45 : Keyboard and display interface using 8279.

The chip select signal $\overline{CS}$, is obtained from the IO address decoder of the 8086 system. The chip select signals for IO-mapped devices are generated by using a 3-to-8 decoder. The address lines A_5, A_6 and A_7 are used as input to the decoder. The address line A_0 and the control signal $M/\overline{IO}$ are used as enable for the 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-7.15.

TABLE - 7.15 : IO ADDRESSES OF 8279

Internal device	Binary address								Hexa address
	Decoder input			Input to address pin of 8279				Decoder enable	
	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
Data register	0	1	1	x	x	x	0	0	60
Control register	0	1	1	x	x	x	1	0	62

Note : Don't care "x" is considered as zero.

The circuit has 8 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 the 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_7 are used only for display.

The common cathode LEDs, LT543 are used in the circuit shown in Fig. 7.45. The corresponding segments of the anodes are connected to a common line to form a bus and this bus can be called a segment bus (i.e., segment "a" of all 7-segment LEDs are connected to a common line, similiarly segment "b" and so on).

Anode and cathode drivers are provided to take care of the current requirement of the LEDs. The pnp transistors, BC158, are used as driver transistors. The anode drivers are called segment drivers and the cathode drivers are called digit drivers.

The 8279 outputs the display code for one digit through its output lines (OUT A_0 to OUT A_3 and OUT B_0 to OUT B_3) and sends a scan code through SL_0-SL_3. The display code is inverted by the segment drivers and sent to the 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 milliseconds, typical), the display is turned OFF (i.e., display is blanked) and the above the process is repeated for the next digit. Thus multiplexed display is performed by 8279.

Note : Since the anode drivers inverts the display code, the complement of the data required to turn ON a common cathode LED should be loaded in the display RAM of 8279.

The keyboard matrix is formed using the return lines RL_0 to RL_7 of 8279 as columns and decoded scan lines S_0 and S_1 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 the binary count on SL_0 to SL_3, which is decoded by the 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 the 8279 detects a key press then it waits for debounce time and again read the columns to generate the keycode. In encoded scan keyboard mode, the 8279 stores an 8-bit code for each valid key press. The keycode consists of the binary value of the column and row in which the key is found and the status of the shift and control key. The format of the code entered in FIFO RAM is shown in Fig. 7.45. 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.

7.5 PROGRAMMABLE TIMER - INTEL 8254

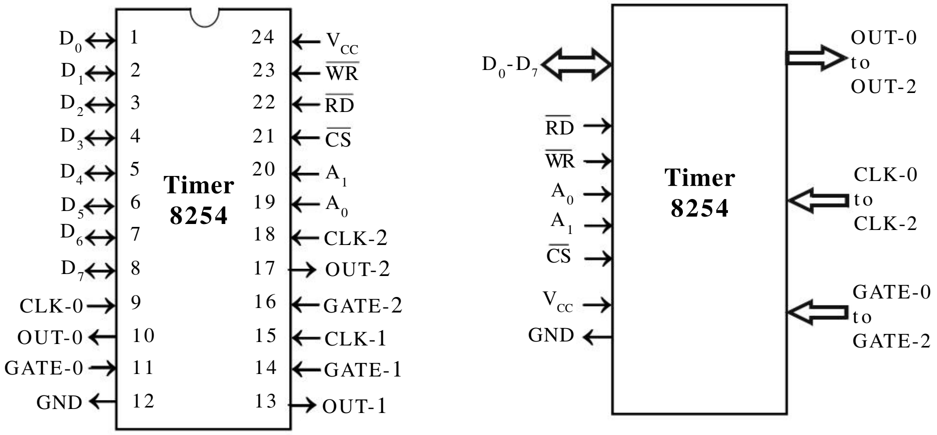
When the processor has to perform time-based activities, there are two methods to maintain the timings of the operations. In one method the processor can execute a delay subroutine. In this method, the delay subroutine will load a count value in one of the registers of the processor and starts decrementing 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 periodic 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 the 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 an interrupt to the processor to initiate the time-based activity. Some of the applications of a 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 e.g., start of conversion signal to ADC.)
3. The timer can be used as a baud rate generator. (For e.g., the timer can be used as a clock divider to divide the processor clock to desired the 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 the 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. 7.46. The functional block diagram of 8254 is shown in Fig. 7.47.

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 the clock input. The counter counts by decrementing the count value by one in each cycle of the clock signal and generates an output depending on the mode of operation. The maximum input clock frequency for 8254 is 10 MHz.



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. 7.46 : Pin configuration of an 8254 timer.

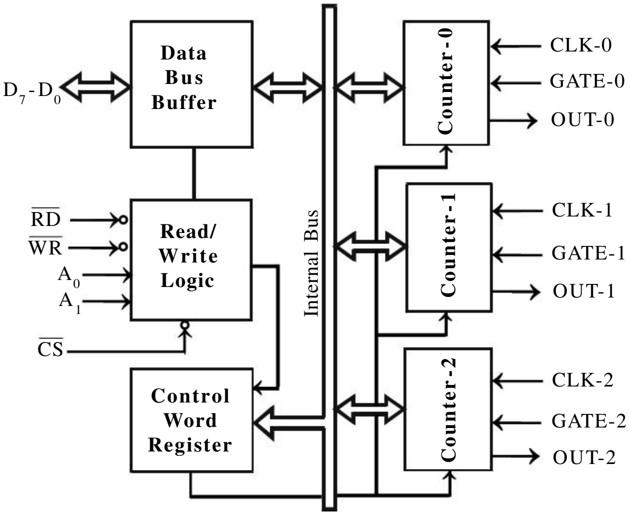


Fig. 7.47 : Functional block diagram of an 8254 timer.

Note : Another timer released by INTEL is 8253 which is a low clock version of the 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.

The 8254 has eight data lines which can be used for communication with the processor. The control words and count values are written into 8254 registers through the 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-7.16. 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 at any time.

TABLE - 7.16 : 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

Interfacing 8254 with 8086 Processor

A simple schematic for interfacing the 8254 with 8086 processor is shown in Fig. 7.48. The 8254 can be either memory-mapped or IO-mapped in the system. In the schematic shown in Fig. 7.48, the 8254 is IO-mapped in the system with even addresses. The chip select signals for IO-mapped devices are generated by using a 3-to-8 decoder. The address lines A_5, A_6 and A_7 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_0 and the control signal $M/\overline{IO}$ are used to enable the decoder.

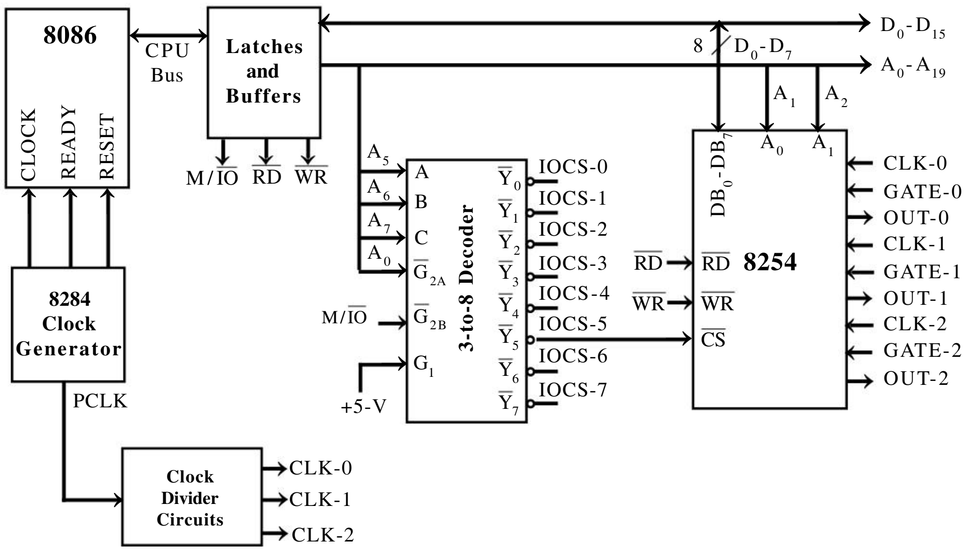


Fig. 7.48 : Interfacing of 8254 with 8086 processor.

The address lines A_1 and A_2 of 8086 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-7.17. The data lines D_0-D_7 are connected to D_0-D_7 of the processor, and $\overline{RD}$ and $\overline{WR}$ signals of 8254 are connected to the $\overline{RD}$ and $\overline{WR}$ of the processor respectively to achieve parallel data transfer.

The clock signals required for the counters can be obtained either from the processor clock or from the **P**eripherals **cl**oc**k** (PCLK) supplied by the clock generator 8284. The clock signals from 8284 can also be divided to lower values by using clock divider circuits and then applied to the clock input of counters.

TABLE- 7.17 : IO ADDRESSES OF 8254

Internal device	Binary address								Hexa address
	Decoder input			Input to address pins of 8254				Decoder enable	
	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
Counter-0	1	0	1	x	x	0	0	0	A0
Counter-1	1	0	1	x	x	0	1	0	A2
Counter-2	1	0	1	x	x	1	0	0	A4
Control Register	1	0	1	x	x	1	1	0	A6

Note : Don't care "x" is considered as zero.

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. 7.49.

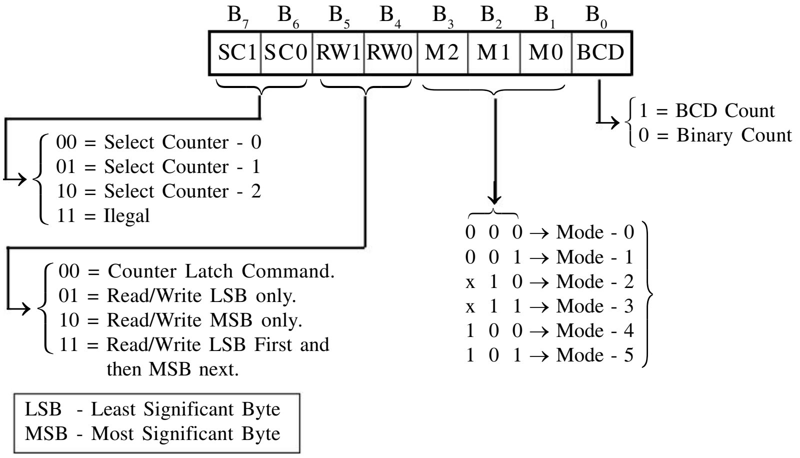


Fig. 7.49 : Format of control word for timer 8254.

The bit B_0 (BCD) of the control word is used to select BCD or binary count and the bits B_1 to B_3 (M0, M1 and M2) 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 separate control word has to be sent to the same control register address. The 8254 identifies the control word for a particular counter from bits B_6 and B_7 of the 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 a 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 the read-back control word of 8254 is shown in Fig. 7.50. This control word has to be sent to the same control register address before the read operation to latch the count value. The control register identifies this control word from the value of bits B_6 and B_7.

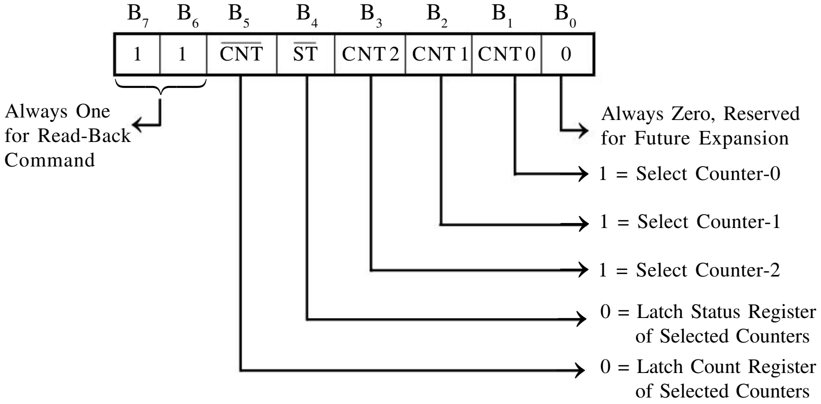

B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0
1 | 1 | $\overline{\text{CNT}}$ | $\overline{\text{ST}}$ | CNT 2 | CNT 1 | CNT 0 | 0
Always One for Read-Back Command
Always Zero, Reserved for Future Expansion
1 = Select Counter-0
1 = Select Counter-1
1 = Select Counter-2
0 = Latch Status Register of Selected Counters
0 = Latch Count Register of Selected Counters

Fig. 7.50 : Format of read-back control word of 8254.

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 the status register of each counter is shown in Fig. 7.51. 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.

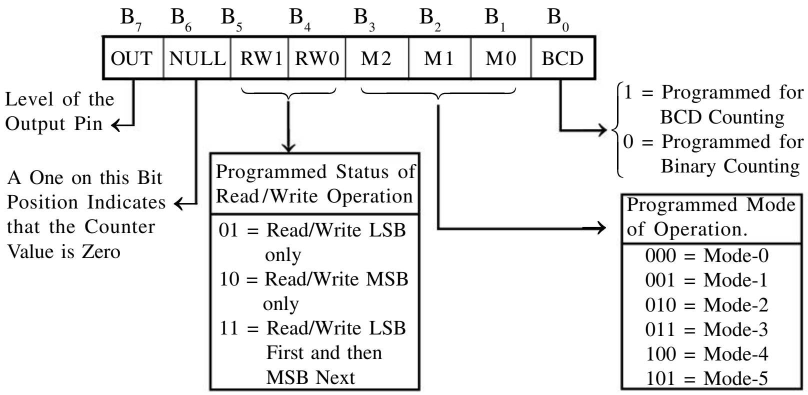

B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0
OUT NULL RW1 RW0 M2 M1 M0 BCD
Level of the Output Pin
A One on this Bit Position Indicates that the Counter Value is Zero
Programmed Status of Read/Write Operation
01 = Read/Write LSB only
10 = Read/Write MSB only
11 = Read/Write LSB First and then MSB Next
1 = Programmed for BCD Counting
0 = Programmed for Binary Counting
Programmed Mode of Operation.
000 = Mode-0
001 = Mode-1
010 = Mode-2
011 = Mode-3
100 = Mode-4
101 = Mode-5

Fig. 7.51 : 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 :

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 the control register.
2. Write a count value in the count register.

The writing of count value depends on the control word. There may be three possible choices:

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 act 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 act 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 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.

The timing diagram for mode-0 operation is shown in Fig. 7.52. In the timing diagram of Fig. 7.52 (a) initially the counter output remains **high**, and it is assumed that the GATE is always **high**. The processor writes the control word and the count value using the write control signal ($\overline{WR}$). Once the control word is written into the 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 the 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**.

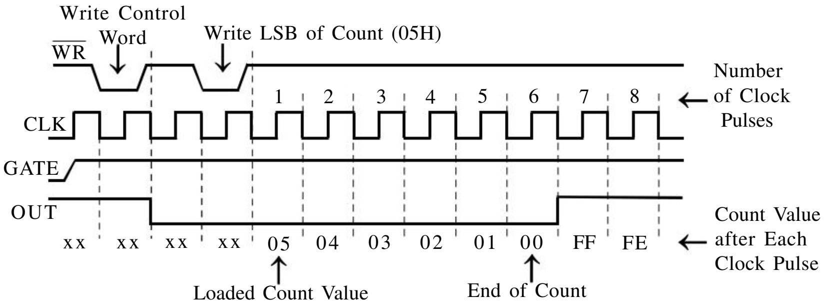


Fig. a : Timing diagram of Mode-0 with GATE always **high**.

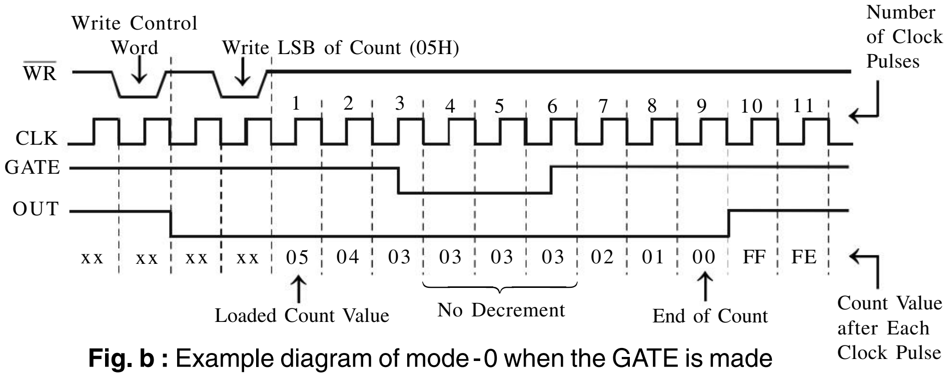


Fig. b : Example diagram of mode-0 when the GATE is made **low** for small duration before the terminal count.

Note : "xx" represents undefined count value.

Fig. 7.52 : Timing diagram of mode-0 of 8254.

Figure 7.52 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 → FF ; FF → FE and so on) as long as GATE is **high** and the clock signal is supplied. The output of the counter remains **high** until a new count or command is sent to the counter.

In the timing diagram shown in Fig. 7.52 (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.

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 the 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, mode-1 produces a logic **low** pulse output whose width is equal to the duration of the count.

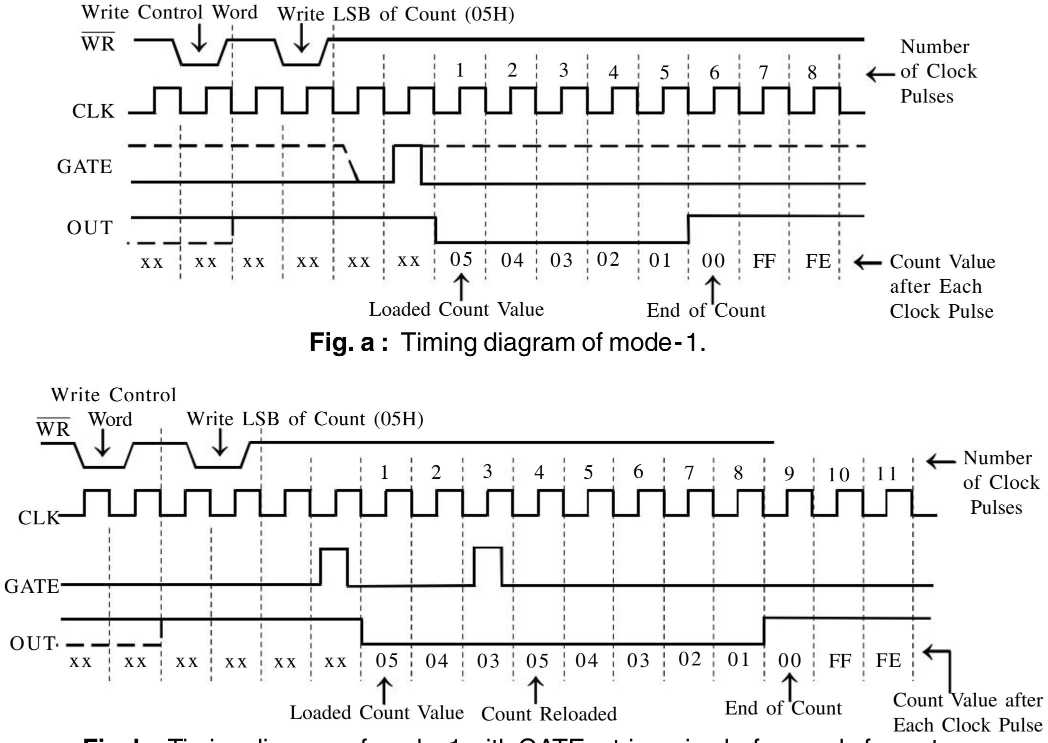


Fig. a : Timing diagram of mode-1.

Fig. b : Timing diagram of mode-1 with GATE retriggering before end of count.

Note : "xx" represents undefined count value.

Fig. 7.53 : Timing diagram of mode-1 of 8254.

The timing diagram of mode-1 operation is shown in Fig. 7.53. The processor writes the control word and count value using the $\overline{\text{WR}}$ 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 the 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 the **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 in the counter the output is asserted **low** and at the end of the count, when the count value is zero, the output is asserted **high**. In the timing diagram shown in Fig. 7.53 (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.

In the timing diagram of Fig. 7.53 (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. 7.54. The processor writes the control word and count value using the $\overline{\text{WR}}$ 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 the control register. The GATE input is permanently tied to logic **high**. In the first clock pulse after the $\overline{\text{WR}}$ signal goes **high**, the count value is loaded in the counter and the count value is decremented by one for each subsequent clock pulse.

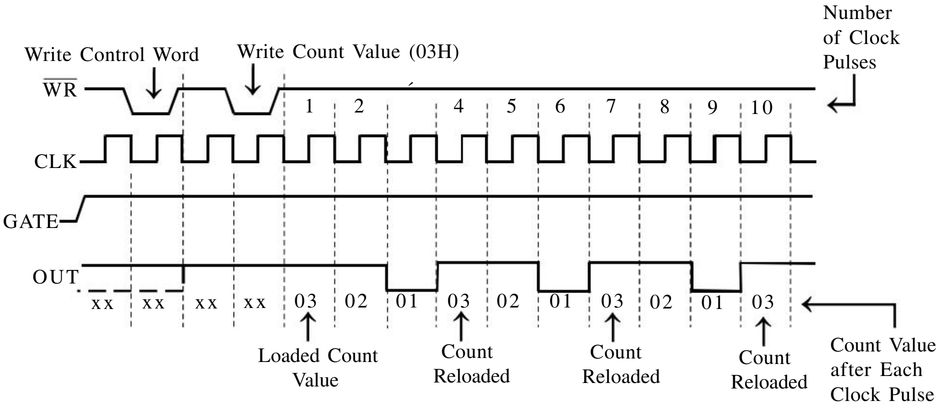


Fig. 7.54 : Timing diagram of mode-2 of 8254.

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. 7.54, 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.

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 an 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. 7.55.

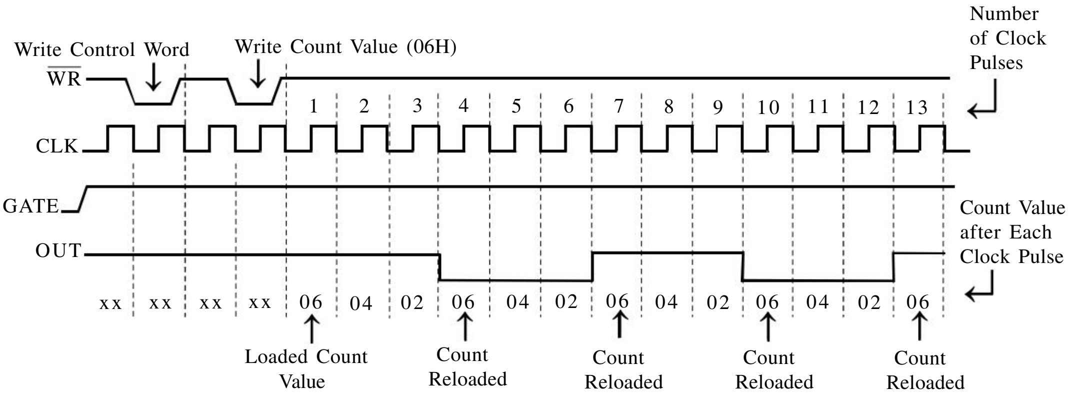


Fig. 7.55 : Timing diagram of mode-3 of 8254.

In the timing diagram shown in Fig. 7.55, 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 the 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 $(N+1)^{th}$ clock pulse. Here the delay time is N clock periods. This signal is often used as strobe signal in parallel data transfer scheme. Mode-4 is called software triggered strobe because the counter starts its operation once the count value is written into the 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. 7.56. 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**.

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**.

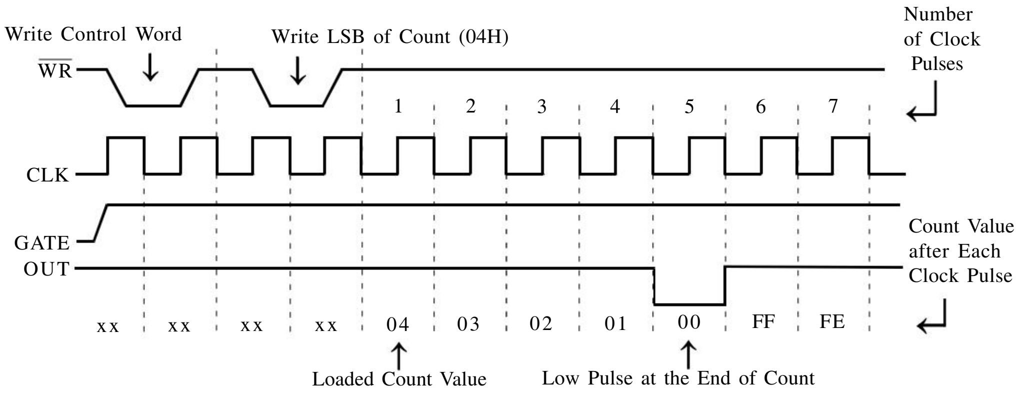


Fig. 7.56 : Timing diagram of mode-4 operation of 8254.

Mode-5 : Hardware Triggered Strobe

Mode-5 is 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.

The timing diagram of mode-5 operation is shown in Fig. 7.57. 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 wait 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., retriggered) before the end of count then the original count value is reloaded in the clock pulse after gate retriggering and count value is decremented by one in each subsequent clock pulse.

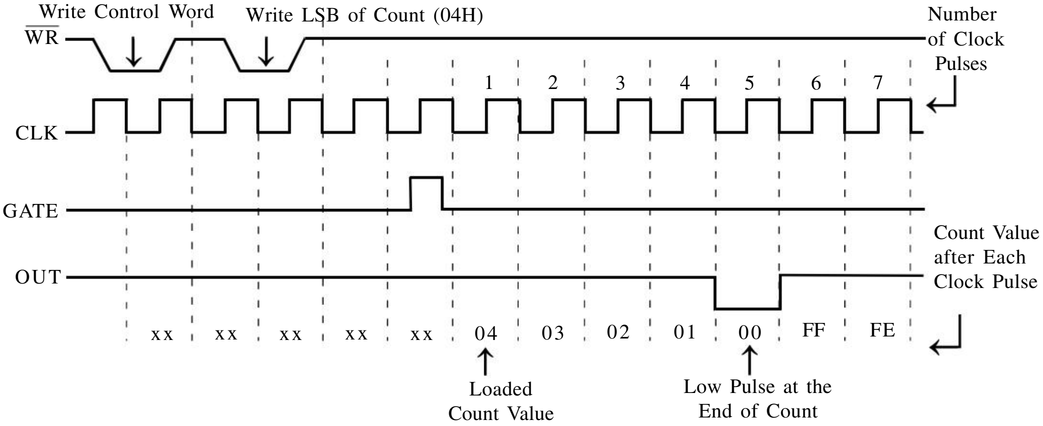


Fig. 7.57 : Timing diagram of mode-5 operation of 8254.

7.6 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 **D**igital-to-**A**nalog **C**onverter (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 the 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 the 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 inputs are as shown in Table-7.18.

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.

TABLE - 7.18

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

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. 7.58

The switches in the circuit of Fig. 7.58 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.

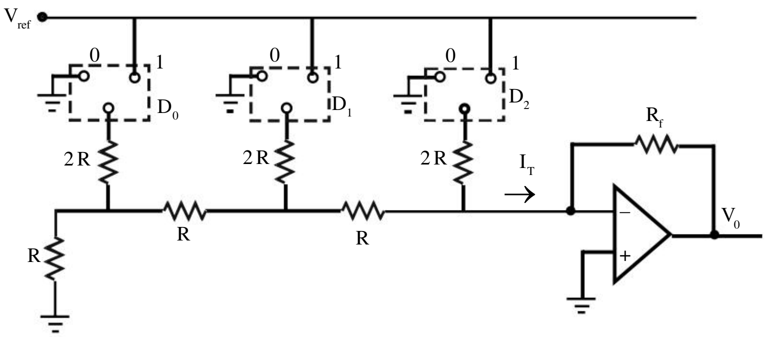


Fig. 7.58 : A typical R/2R ladder resistive network as DAC.

$$V_0 = V_{ref} \frac{R_f}{R}\left(\frac{D_2}{2^1} + \frac{D_1}{2^2} + \frac{D_0}{2^3}\right)$$

The circuit of Fig. 7.58 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 the 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 the microprocessors are available with or without internal latch and I to V converting amplifier. The AD558 of the Analog Device is an example of an 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 8086 microprocessor bus and it requires only two control signals : **C**hip **S**elect ($\overline{CS}$) and **C**hip **E**nable ($\overline{CE}$). [No handshake signals are necessary for interfacing a DAC. The time between loading two digital data to the DAC is controlled by software time delay.]

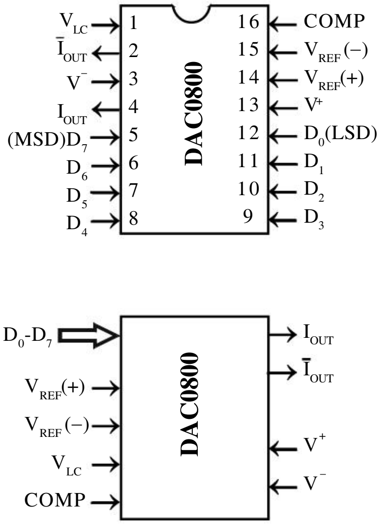
The DAC0800 of the National Semiconductor Corporation is an example of an 8-bit DAC without internal latch and I to V converting amplifier. The DAC0800 can be interfaced to the microprocessor using either a port device or a latch.

7.6.1 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 a simple resistor load.

The DAC0800 is available as a 16-pin IC in DIP. The pin configuration of DAC0800 is shown in Fig. 7.59 and the internal block diagram of a DAC0800 is shown in Fig. 7.60.

The DAC0800 requires a positive and a negative supply voltage in the range of ± 5-V to ±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. 7.61.



Pin	Description
D_0-D_7	Digital input data
I_{OUT}	Current output
$\bar{I}_{OUT}$	Complement of output current
V^-	Negative supply voltage
V^+	Positive supply voltage
COMP	Compensation voltage
V_{LC}	Threshold control
$V_{REF}(+)$	Positive reference voltage
$V_{REF}(-)$	Negative reference voltage

MSD - Most Significant Digit
LSD - Least Significant Digit

Fig. 7.59 : Pin description of DAC0800.

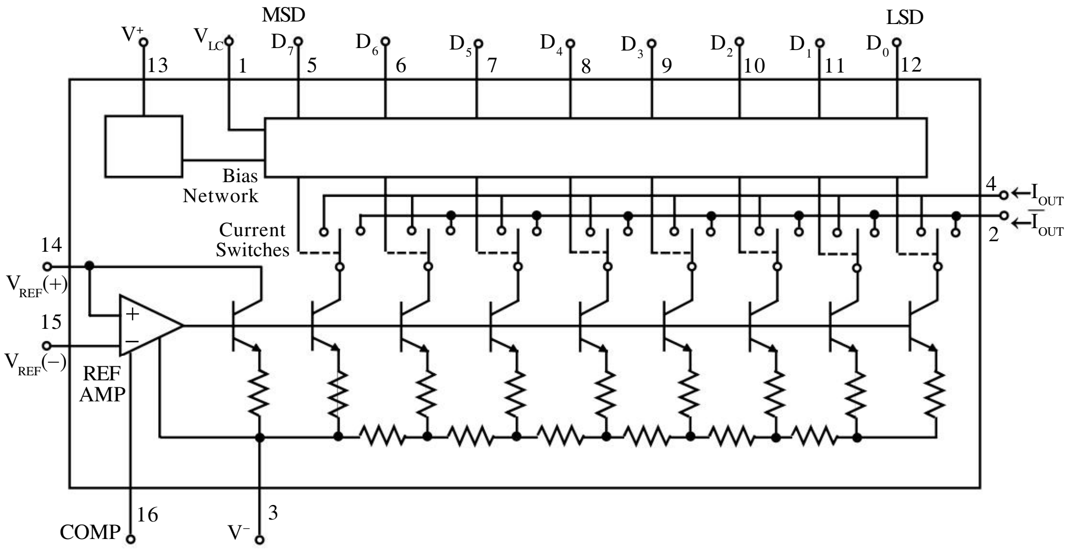
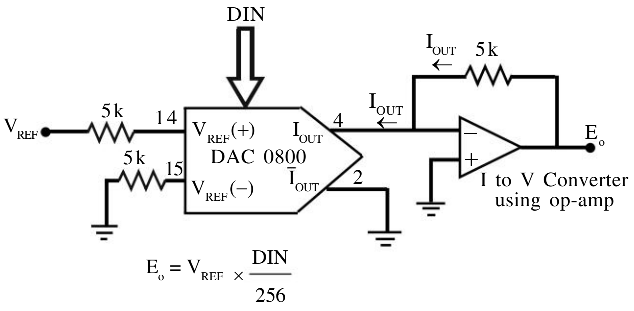


Fig. 7.60 : Block diagram of DAC0800.



$$E_o = V_{REF} \times \frac{DIN}{256}$$

where, DIN = Decimal Equivalent of Binary Input

Fig. 7.61 : DAC 0800 with V to I converter to produce positive output voltage.

Interfacing DAC0800 With 8086

The DAC0800 can be interfaced to 8086 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 the DAC0800 with 8086 is shown in Fig. 7.62. In this schematic the DAC0800 is interfaced using an 8-bit latch 74LS273 to the system bus and the latch is IO-mapped in the system with an even address. The 3-to-8 decoder 74LS138 is used to generate chip select signals for IO devices. The address lines A_5, A_6 and A_7 are used as inputs to the decoder. The address line A_0 and the control signal $M/\overline{IO}$ 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-7.19.

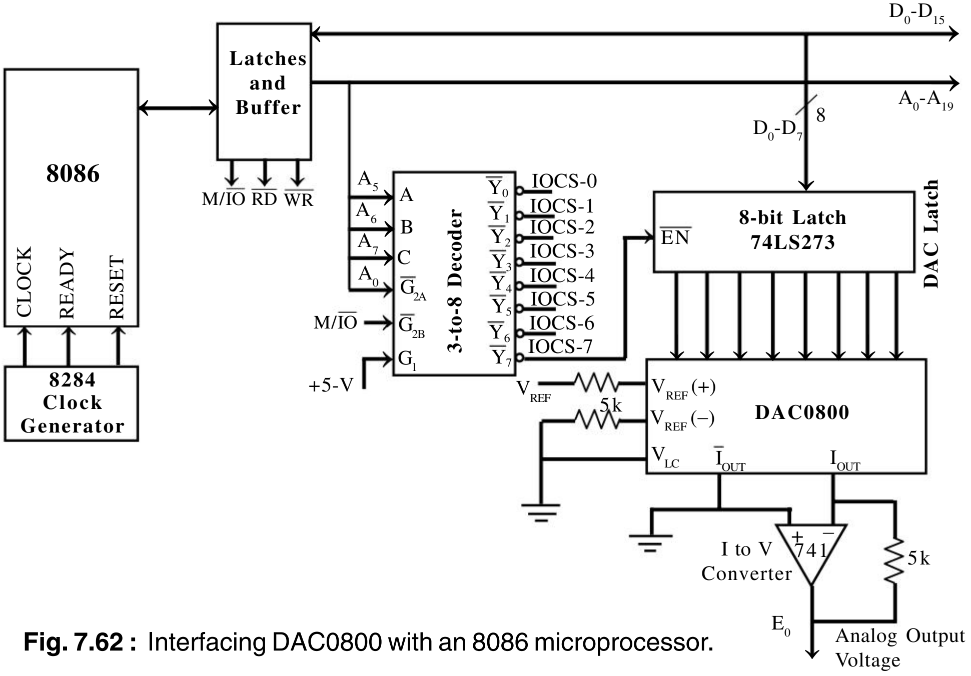

D_0-D_{15}
A_0-A_{19}
Latches and Buffer
8086
CLOCK
READY
RESET
8284 Clock Generator
$M/\overline{IO}$ $\overline{RD}$ $\overline{WR}$
D_0-D_7
8
A_5
A_6
A_7
A_0
A
B
C
$\overline{G}_{2A}$
$\overline{G}_{2B}$
G_1
$M/\overline{IO}$
+5-V
3-to-8 Decoder
$\overline{Y}_0$ $\overline{IOCS\text{-}0}$
$\overline{Y}_1$ $\overline{IOCS\text{-}1}$
$\overline{Y}_2$ $\overline{IOCS\text{-}2}$
$\overline{Y}_3$ $\overline{IOCS\text{-}3}$
$\overline{Y}_4$ $\overline{IOCS\text{-}4}$
$\overline{Y}_5$ $\overline{IOCS\text{-}5}$
$\overline{Y}_6$ $\overline{IOCS\text{-}6}$
$\overline{Y}_7$ $\overline{IOCS\text{-}7}$
$\overline{EN}$
8-bit Latch 74LS273
DAC Latch
V_{REF}
5k
$V_{REF}(+)$
$V_{REF}(-)$
V_{LC}
DAC0800
$\bar{I}_{OUT}$
I_{OUT}
I to V Converter
741
5k
E_0
Analog Output Voltage

Fig. 7.62 : Interfacing DAC0800 with an 8086 microprocessor.

TABLE - 7.19 : IO ADDRESS OF DAC LATCH

Device	Binary address								Hexa address
	Decoder input			Unused address lines				Decoder enable	
	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
DAC Latch 74LS273	0	1	1	x	x	x	x	0	60

7.7 ADC INTERFACE

In many applications, an analog device has to be interfaced to the digital system. But the digital devices cannot accept the analog signals directly and so the analog signals are converted to equivalent digital signals (data) using **A**nalog-to-**D**igital **C**onverter (ADC).

The **A**nalog to **D**igital (A/D) conversion is the reverse process of **D**igital to **A**nalog (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 defined only for discrete instant of time. If the digital data is represented by an 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 the internally generated analog signal. The second group includes integrator converters and voltage to frequency converters. In the devices of the second group, the given analog signal is converted to time or frequency and the new parameters (time or frequency) is compared with the 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 faster 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 7.63 shows the functional blocks of a typical successive approximation A/D converter.

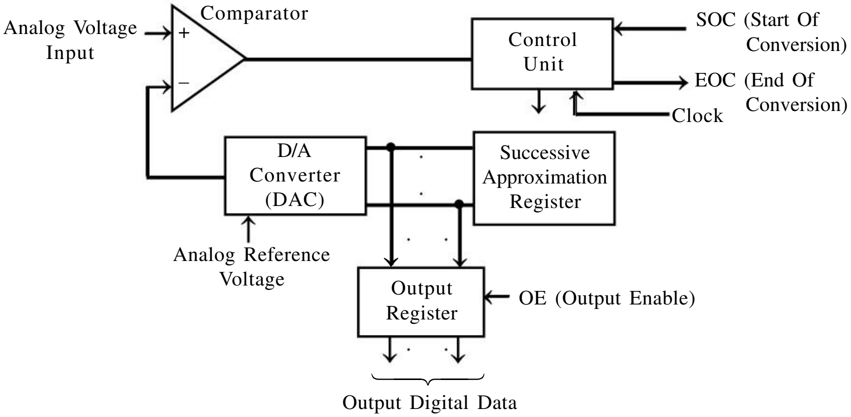


Fig. 7.63 : Successive approximation A/D converter.

The conversion process is initiated by a **S**tart **O**f **C**onversion (SOC) signal from the processor to the ADC. On receiving the SOC, the control unit of the ADC will give a start command to the successive approximation register and it starts generating digital signal by the successive approximation method. The generated digital data is converted to analog signal by the D/A converter and then compared with given the analog signal. When the analog signals are equal the comparator output informs the control unit to stop generation of 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 **E**nd **O**f **C**onversion (EOC) process to the processor.

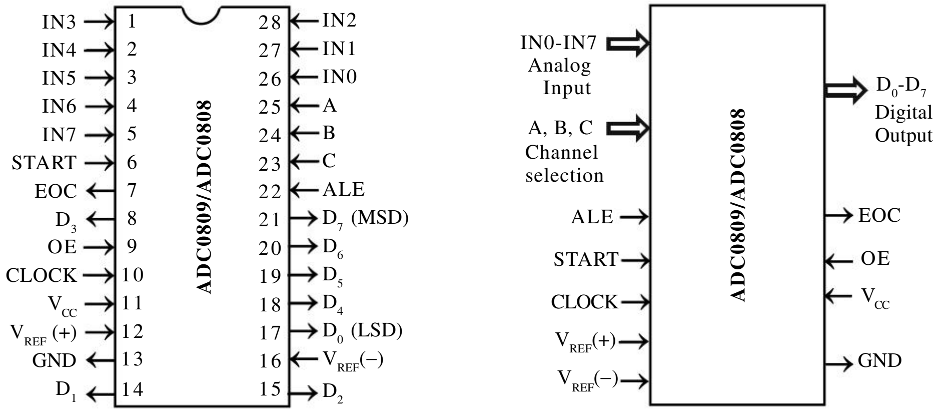
Successive Approximation Method of Conversion

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 the given analog signal. (Initially the comparator output will be **high**. After comparison the output of the 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 the comparator changes from **high** to **low** state.) If the output state of the 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.

7.7.1 ADC0809

The ADC0809 is an 8-bit successive approximation type ADC with inbuilt 8-channel multiplexer. The ADC0809 is suitable for interface with 8086 microprocessor. The ADC0809 is available as a 28-pin IC in DIP (**D**ual **I**n-line **P**ackage). The ADC0809 has a total unadjusted error of ±1 LSD (**L**east **S**ignificant **D**igit). 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. 7.64.

The internal block diagram of ADC0809/ADC0808 is shown in Fig. 7.65. 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.



LSD = Least Significant Digit, MSD = Most Significant Digit

Fig. 7.64 : Pin configuration of ADC0809/ADC0808.

TABLE - 7.20 : 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 **high** and then **low**. This signal should remain **high** 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 - 7.20 : 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 the analog input to the digital data.
EOC	End of conversion. This signal is asserted **high** 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 the output buffer after a valid EOC.
V_{cc}	Power supply, +5-V
GND	Power supply ground, 0-V

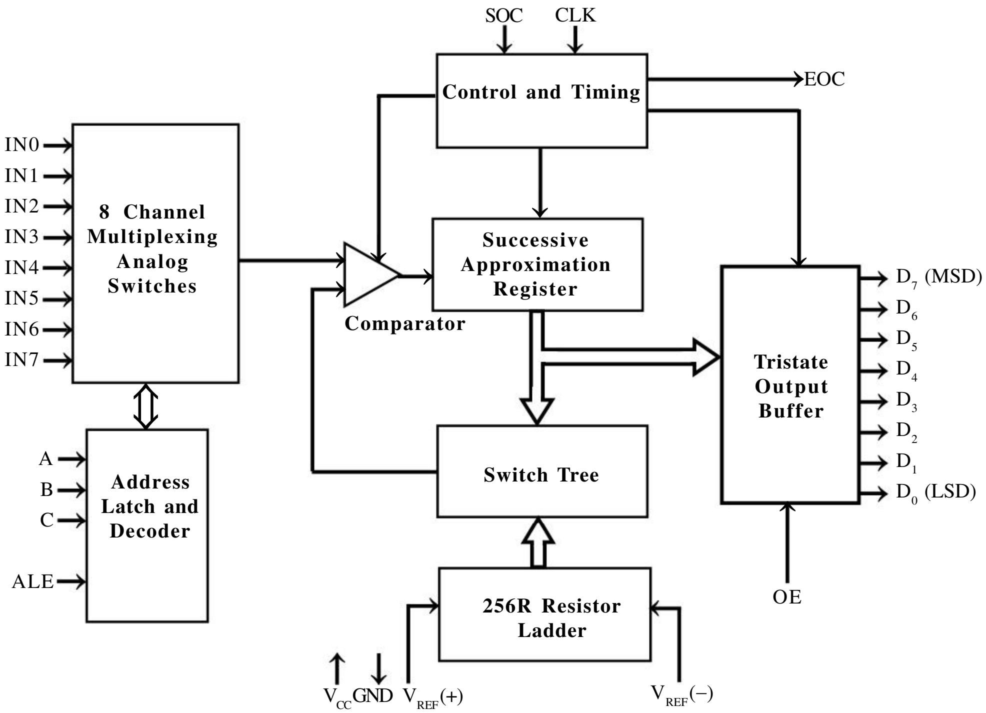


Fig. 7.65 : Functional block diagram of ADC0809/ADC0808.

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-7.21.

The **S**uccessive **A**pproximation **R**egister (SAR) performs eight iterations to determine the digital code for input value. The SAR is reset on the positive edge of the START pulse and start the conversion process on the falling edge of START pulse. A conversion processs will be interrupted on receipt of a new START pulse. The **E**nd **O**f **C**onversion (EOC) will go **low** between 0 and 8 clock pulses after the positive edge of the START pulse. The ADC can be used in continuous conversion mode by tying the EOC output to the 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. 7.66. 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.

TABLE - 7.21

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

The comparator in the ADC0809/ADC0808 is a chopper-stabilized comparator. It converts the DC input signal into an AC signal and amplifies the AC signal using a high gain AC amplifier. Then it converts 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,

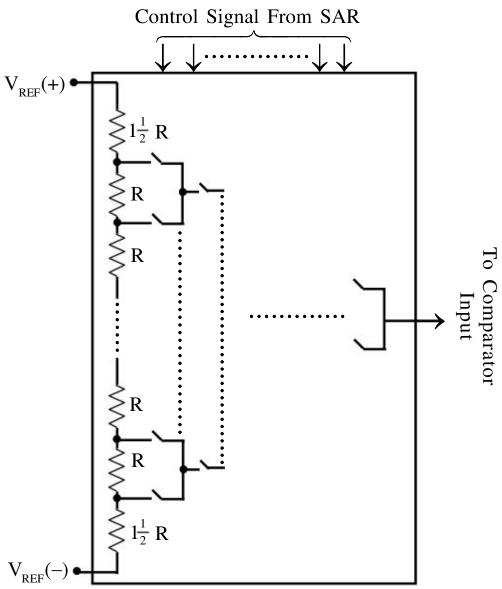


Fig. 7.66: 256R resistor network and switch tree.

$$Q_{step} = \frac{V_{REF}}{2^8} = \frac{V_{REF}(+) - V_{REF}(-)}{256_{10}}$$

The digital data corresponding to an analog input (V_{in}) is given by,

$$\text{Digital data} = \left(\frac{V_{in}}{Q_{step}} - 1\right)_{10}$$

EXAMPLE 1

Let, V_{REF} (+) = 3.84-V, V_{REF} (–) = 0-V

$$\therefore\ Q_{step} = \frac{V_{REF}(+) - V_{REF}(-)}{256_{10}} = \frac{3.84}{256} = 0.015\text{-V} = 15\text{-mV}$$

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

$$\therefore\ Q_{step} = \frac{V_{REF}(+) - V_{REF}(-)}{256_{10}} = \frac{5}{256} = 0.01953125$$

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 8086

A simple schematic for interfacing ADC0809/ADC0808 with 8086 microprocessor is shown in Fig. 7.67. The ADC can be either memory-mapped or IO-mapped in the system. Here the ADC is IO-mapped in the system with even address. The chip select signals for IO-mapped devices are generated by using a 3-to-8 decoder. The address lines A_5, A_6 and A_7 are used as input to decoder. The address line A_0 and the control signal M/$\overline{IO}$ are used as enable for the decoder. The decoder generates eight chip select signals (IOCS-0 to IOCS-7), and these three chip select signals are used for the ADC interface.

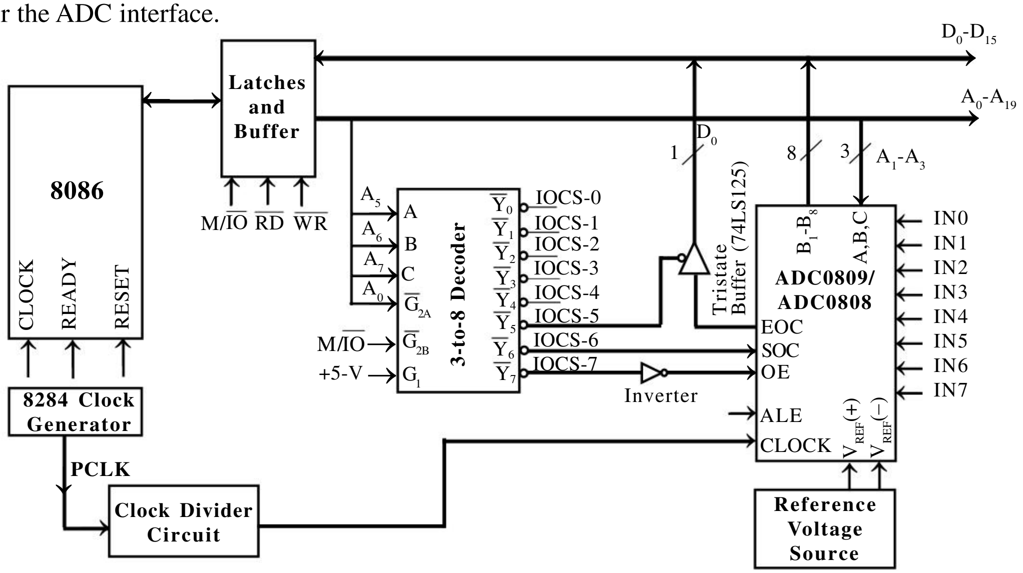


Fig. 7.67 : Interfacing ADC0809/ADC0808 with 8086 microprocessor.

TABLE - 7.22 : IO ADDRESS OF ADC0809/ADC0808

Operation performed	Binary address								Hexa address
	Decoder input			Address input to ADC				Decoder enable	
	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
SOC channel-0	1	1	0	x	0	0	0	0	C0
SOC channel-1	1	1	0	x	0	0	1	0	C2
SOC channel-2	1	1	0	x	0	1	0	0	C4
SOC channel-3	1	1	0	x	0	1	1	0	C6
SOC channel-4	1	1	0	x	1	0	0	0	C8
SOC channel-5	1	1	0	x	1	0	1	0	CA
SOC channel-6	1	1	0	x	1	1	0	0	CC
SOC channel-7	1	1	0	x	1	1	1	0	CE
Read EOC	1	0	1		x	x	x	0	A0
Read ADC Output	1	1	1		x	x	x	0	E0

The chip select signal IOCS-6 is used to give the **S**tart **O**f **C**onversion (SOC) signal to the ADC along with a channel address. The chip select IOCS-5 is used to enable the tristate buffer provided for interfacing EOC with the data bus. The chip select signal IOCS-7 is inverted and used to enable the output buffer of the ADC whenever the digital data has to read from the ADC.

The **P**eripheral **C**lock **S**ignal (PCLK) of the 8284 is divided by a suitable clock divider circuit and used as a clock signal for the ADC. A separate voltage source has to be provided to give an accurate reference voltage levels. The **E**nd **O**f **C**onversion (EOC) signal of the ADC is connected to the bus line D_0 of the system through a tristate buffer, so that the processor can check for a valid EOC before reading the output buffer of ADC.

7.8 SHORT QUESTIONS AND ANSWERS

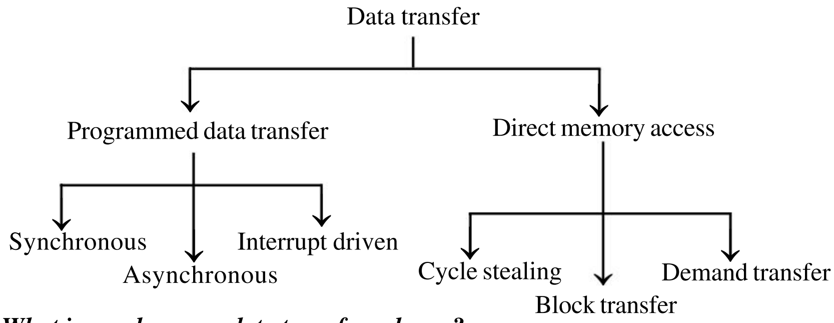
7.1 What is a programmable peripheral device?

If the functions performed by a peripheral device can be altered or changed by a program instruction then the peripheral device is called a programmable device. Usually programmable devices will have control registers. The device can be programmed by sending the control word in the prescribed format to the control register.

7.2 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 the peripheral devices.

The different types of data transfer schemes are shown below :



7.3 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.

7.4 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 execute a data transfer instruction to complete the process.

7.5 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.

7.6 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.

7.7 What are the functions performed by port-C of 8255?

1. The 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 port in mode-0.
4. The individual pins of port-C can be set or reset for various control applications.

7.8 What is a handshake port ?

In a handshake port, signals are exchanged between the IO device and the port or between the port and the processor for checking/informing various condition of the device.

7.9 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 the data to the 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 is repeated.

7.10 Explain the working of a handshake output port.

In a handshake output operation, the processor will load a data to the 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 is repeated.

7.11 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 will send a HOLD request to the processor. When the processor receives a HOLD request, it will drive its tristated pins to **high impedance** state at the end of current instruction execution and send an acknowledge signal to DMA controller. Now the DMA controller will perform DMA transfer.

7.12 What are the different types of DMAs?

The different types of DMA data transfer are cycle stealing (or single transfer) DMA, Block transfer (or Burst mode) DMA and Demand transfer DMA.

7.13 What is cycle stealing DMA?

In cycle stealing DMA (or single transfer mode) the DMA controller will perform one DMA transfer in between instruction cycles (i.e., In this mode the execution of one processor instruction and one DMA data transfer will take place alternatively).

7.14 What is block and demand transfer mode DMA?

In block transfer mode, the DMA controller will transfer a block of data and relieve the bus to 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.

7.15 What are the programmable registers of 8237?

The programmable registers of 8237 are base address registers, base word count registers, command register, request register, mode registers and mask register.

7.16 What is the first-last flip-flop?

The 8237 has an internal flip-flop called first-last flip-flop which takes care of reading/writing 16-bit information through 8-data lines.

The first-last flip-flop selects the low or high byte during read/write operation of the 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.

7.17 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 identifies its end. A typical bit format is shown in Fig. Q7.17.

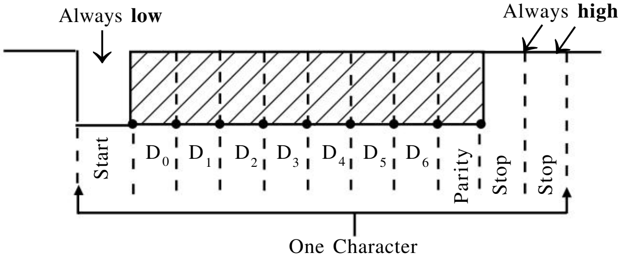


Fig. Q7.17 : Bit format used for sending asynchronous serial data.

7.18 What is baud rate ?

The baud rate is the rate at which the serial data are 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 the same.

7.19 What is RS-232C standard ?

The RS-232C is a serial bus consisting of a maximum of 25 signals. This bus signals are standardized by EIA (**E**lectronics **I**ndustries **A**ssociation), 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.

7.20 What voltage levels are used in RS-232C serial communication standard?

The voltage levels for all RS-232C signals are :

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**).

7.21 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.

7.22 What is USART ?

The device which can be programmed to perform synchronous or asynchronous serial communication is called USART (**U**niversal **S**ynchronous **A**synchronous **R**eceiver **T**ransmitter). The INTEL 8251A is an example of USART.

7.23 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 the MODEM and establish serial communication through the MODEM over telephone lines.

7.24 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.

7.25 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.

7.26 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.

7.27 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 milliseconds.) Hence the processor has to wait for the key bounces to settle down before reading the keycode. This process is called keyboard debouncing.

7.28 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 the first row and coming back to the first row again.

7.29 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.

7.30 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.

7.31 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.

7.32 What is the disadvantage of 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.

7.33 What is the advantage of 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, and so the processor time can be more efficiently used for computing.

7.34 List the functions performed by 8279.

The functions performed by 8279 are,

- Keyboard scanning
- Informing the key entry to CPU
- Display refreshing
- Key debouncing
- Storing display codes
- Keycode generation
- Output display codes to LEDs

7.35 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 the 8279 is 8 x 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.

7.36 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.

7.37 What are the different scan modes of 8279?

The different scan modes of 8279 are decoded scan and encoded scan. In a 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 the decoded output.

7.38 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 a binary count. In encoded mode, an external decoder should be used to decode the scan lines.

7.39 How is a keyboard matrix formed in 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.

7.40 What are the operating modes of the 8254 timer?

The 8254 timer has six operating modes and they are :

1. Mode-0 → Interrupt on terminal count.
2. Mode-1 → Hardware retriggerable one shot.
3. Mode-2 → Rate generator or Timed interrupt generator.
4. Mode-3 → Square wave mode.
5. Mode-4 → Software triggered strobe.
6. Mode-5 → Hardware triggered strobe.

7.41 What is the function of GATE signal in the 8254 timer?

In timer 8254, the GATE signal acts as a control signal to start, stop or maintain the counting process. In modes 0, 2, 3 and 4 the GATE signal should remain **high** to start and maintain the counting process. In modes 1 and 5 the GATE signal has to make a low-to-high transition to start the counting process and need not remain **high** to maintain the counting process.

7.42 What will be the frequency of square wave generated by the 8254 timer in mode-3?

The frequency of the generated square wave is given by the frequency of the input clock signal divided by the count value loaded in the count register. If the count value N is an even number then the square wave will be alternatively **high** and **low** for N/2 clock periods. If the count value N is an odd number then the **high** time of square wave will be $\frac{N+1}{2}$ clock periods and **low** time will be $\frac{N-1}{2}$ clock periods.

7.43 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.

7.44 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.

7.45 What is settling or conversion time in DAC?

The time taken by the DAC to convert a given digital data to a corresponding analog signal is called conversion time.

7.46 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.

7.47 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.

CHAPTER 8

INTEL 80X86 FAMILY OF PROCESSORS

8.1 INTRODUCTION

The microprocessors are used as CPU in personal computers. The IBM Corporation, USA, designed the first **P**ersonal **C**omputer(PC) using INTEL 8088 as CPU. Ever since the introduction of the PC, the applications and usage of personal computers, has increased day-by-day. This leads to great improvement and advancement in PC, which inturn demands improvement in microprocessors.

The INTEL has realized the demand for improvement in microprocessor and so keeps on updating the features of microprocessors year-after-year. Every year INTEL is releasing new processors with improved features. (Please refer to Appendix-I for complete list of processors released by INTEL.) The improvements in microprocessors are related to the following :

- increased word length and memory space
- increased internal performance and clock rating
- increased external communications and error detection
- improved instruction set and support to software
- improved trouble shooting aids/provision

Initially, INTEL had retained the 8086 architecture as base architecture and improved the other features of processor and released 80x86 family of processors. The 80x86 family includes 80186, 80286, 80386 and 80486 processors. Then INTEL switched to superscalar architecture and released Pentium family of processors. The Pentium family of processors includes Pentium, Pentium Pro, Pentium II, Pentium III and Pentium 4 processors. The data and address bus size, addressable memory space and internal clock ratings of 80x86 and Pentium family of processors are listed in Table-8.1. The major new features of 80x86 and Pentium family of processors are listed in Table-8.2.

TABLE - 8.1 : INTEL 80X86 AND PENTIUM FAMILY OF PROCESSORS

INTEL processor	Internal data bus	External data bus	Address bus	Physical memory space	Virtual memory space	Internal clock rating
8086	16-bit	16-bit	20-bit	1 Mb	-	5/8/10 MHz
8088	16-bit	8-bit	20-bit	1 Mb	-	5/8 MHz
80186	16-bit	16-bit	20-bit	1 Mb	-	10/12 MHz
80286	16-bit	16-bit	24-bit	16 Mb	1 Gb	6/10/12 MHz
80386 DX	32-bit	32-bit	32-bit	4 Gb	64 Tb	16/20/25/33 MHz
80486 DX	32-bit	32-bit	32-bit	4 Gb	64 Tb	25/33/50 MHz
Pentium	32-bit	64-bit	32-bit	4 Gb	64 Tb	60 - 200 MHz
Pentium pro	32-bit	64-bit	36-bit	64 Gb	64 Tb	150/166/180/200 MHz
Pentium II	32-bit	64-bit	36-bit	64 Gb	64 Tb	233 - 450 MHz
Pentium III	32-bit	64-bit	36-bit	64 Gb	64 Tb	450 MHz - 1 GHz
Pentium 4	32-bit	64-bit	36-bit	64 Gb	64 Tb	1.4 - 3.3 GHz

TABLE - 8.2 : NEW FEATURES OF 80X86 AND PENTIUM FAMILY OF PROCESSORS

INTEL processor	Major new features
8086/8088	Pipelined architecture, Instruction queue, Coprocessor support, segmented memory.
80286	Instruction pre-decode, Multitasking, Memory protection, Virtual memory, Auto-shutdown.
80386DX	Instruction pipelining, Break-point instruction, Built-in self test, 32-bit internal registers.
80486DX	Cache memory, Internal floating point unit, Restartable instruction, Data bus parity.
Pentium	Superscalar architecture, Dual processor configuration, Internal error detection, Dynamic branch prediction, Performance monitoring, Power management, Functional redundancy check, Machine check, Address bus parity.
Pentium Pro	Out-of-order execution, speculative execution, Register renaming, Secondary cache, ECC (**E**rror **c**hecking and **c**orrecting codes), DIB (**D**ual **I**ndependent **B**us).
Pentium II	MMX (**M**ulti-**M**edia E**x**tension), System management bus, Integrated thermal diode.
Pentium III	Processor serial number, Improved MMX, SIMD (**S**ingle **I**nstruction **M**ultiple **D**ata) extensions.
Pentium 4	Rapid execution engine, Hyperpipelined and **H**yper **T**hreading (HT) technology, Advanced dynamic execution, 400/533/800 MHz system bus.

8.2 INTEL 80186

The 80186 microprocessor was released in 1982. The 80186 is a 16-bit microprocessor and it consists of 15 to 20 of the most common microprocessor-based system components on a single chip. It is actually an integration of BIU (**B**us **I**nterface **U**nit) and EU (**E**xecution **U**nit) of 8086 processor with the following functional units on a single chip.

- Clock generator
- Programmable interrupt controller
- Programmable timers (3 independent 16-bit timers)
- Programmable DMA controller (2 independent DMA channels)
- Programmable memory and peripheral chip selection logic
- Programmable wait state generator
- Local bus controller

The 80186 is object code compatible with 8086 and provides twice the performance of standard 8086. The instruction set of 80186 includes all the instructions of 8086 with additional 10 new instructions.

The 80186 uses a 20-bit address to access memory and hence it can directly address up to one mega-byte (2^{20} = 1 Mega) of memory space. The memory organization in 80186 and the IO addressing are similar to that of 8086.

The 80186 is available with maximum internal clock frequency of 6, 8, 10 and 12 MHz.

8.2.1 Pins and Signals of 80186

The 80186 is a 68-pin IC available in three different packages : PLCC (**P**lastic **L**eaded **C**hip **C**arrier), LCC (Ceramic **L**eadless **C**hip **C**arrier) and PGA (Ceramic **P**in **G**rid **A**rray). The pin configuration of the PLCC package of 80186 is shown in Fig. 8.1. The name of the pins and signals of 80186 processor are listed in Table-8.3.

TABLE - 8.3 : PIN DESCRIPTION OF 80186

Pin name	Description	Type
AD_{15}-AD_0	Multiplexed address/data	Bidirectional, Tristate
A_{19}/S_6-A_{16}/S_3	Multiplexed address/status signals	Output, Tristate
$\overline{BHE}/ S_7$	Bus high enable or status signal S_7	Output, Tristate
ALE/QS_0	Address latch enable or Queue status 0	Output
$\overline{WR}/QS_1$	Write control signal or Queue status 1	Output, Tristate
$\overline{RD}/ \overline{QSMD}$	Read control signal or Queue status mode	Bidirectional
$\overline{S}_0$, $\overline{S}_1$, $\overline{S}_2$	Bus cycle status indicators	Output, Tristate
DT/ $\overline{R}$	Data transmit/receive	Output, Tristate
$\overline{DEN}$	Data enable	Output, Tristate
ARDY	Asynchronous ready signal	Input
SRDY	Synchronous ready signal	Input
$\overline{RES}$	System reset	Input
RESET	Peripheral reset	Output
X_1, X_2	Crystal connection	Input
CLKOUT	Peripheral clock	Output
$\overline{LOCK}$	Bus priority lock control	Output, Tristate
HOLD	Hold request	Input
HLDA	Hold acknowledge	Output
$\overline{TEST}$	Wait on test control	Input
TMR IN 0	Timer 0 input clock or control signal	Input
TMR IN 1	Timer 1 input clock or control signal	Input
TMR OUT 0	Timer 0 output signal	Output
TMR OUT 1	Timer 1 output signal	Output
DRQ 0	Channel 0 DMA request	Input
DRQ 1	Channel 1 DMA request	Input
NMI	Nonmaskable interrupt	Input
INT0, INT1	Maskable interrupt request	Input
INT2, $\overline{INTA0}$	Maskable interrupt request or Acknowledge of INT0	Bidirectional
INT3, $\overline{INTA1}$	Maskable interrupt request or Acknowledge of INT1	Bidirectional
$\overline{UCS}$	Upper memory chip select signal	Output
$\overline{LCS}$	Lower memory chip select signal	Output
$\overline{MCS}_0$ to $\overline{MCS}_3$	Mid-range memory chip select signals	Output

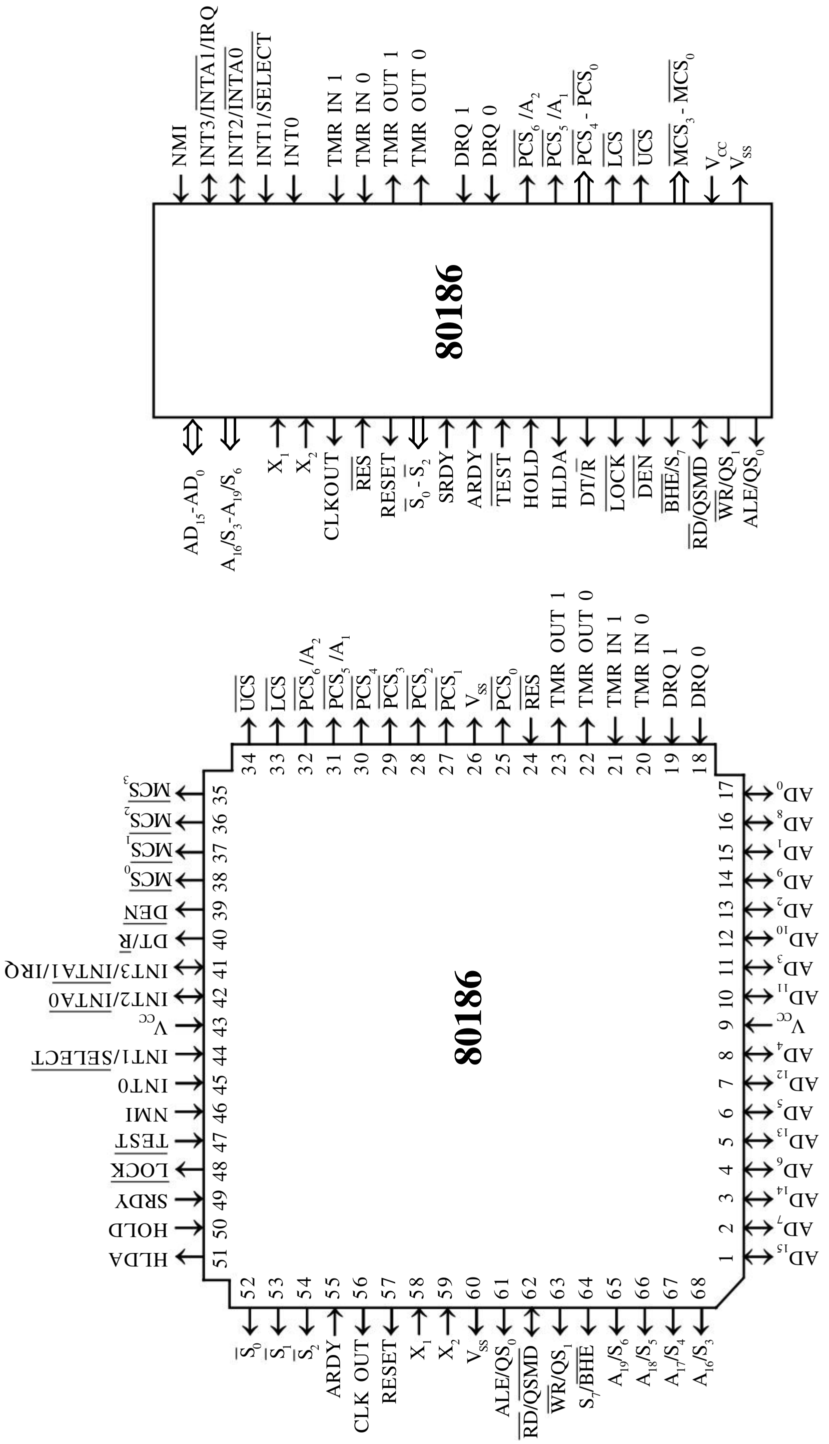


Fig. 8.1 : Pin configuration of 80186.

Table - 8.3 : continued...

Pin name	Description	Type
$\overline{PCS}_0$ to $\overline{PCS}_4$	Peripheral chip select signals	Output
$\overline{PCS}_5/A_1$	Peripheral chip select 5 or Latched address A_1	Output
$\overline{PCS}_6/A_2$	Peripheral chip select 6 or Latched address A_2	Output
V_{CC}	Power supply,+5-V	Input
V_{SS}	Power supply ground, 0-V	Output

In an 80186 processor, the lower sixteen lines of address are multiplexed with the data and the upper four lines of address are multiplexed with the 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 periods of a bus cycle, the data and status signals will be available on these lines. The signal ALE is used to demultiplex address and data/status lines using external latches.

During the processor initiated bus cycle, the status signal S_6 is asserted **low** and during DMA cycle, it is asserted **high**. The status signals S_3, S_4 and S_5 always remain at logic **low**.

The 80186 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 output signal $\overline{BHE}$ on the first T-state of a bus cycle is maintained as status signal S_7 during all other T states of the bus cycle. The $\overline{BHE}$ can be used in conjunction with address bit A_0 (AD_0) to select memory banks. The status of $\overline{BHE}$ and A_0 during word/byte transfer from even/odd memory bank are shown in Table-8.4.

TABLE - 8.4 : STATUS OF $\overline{BHE}$ AND A_0 DURING MEMORY ACCESS

$\overline{BHE}$	A_0	Function
0	0	Word transfer
0	1	Byte transfer on upper half of data bus (D_{15}-D_8)
1	0	Byte transfer on lower half of data bus (D_7-D_0)
1	1	Reserved

The queue status are output through ALE and $\overline{WR}$ pins during the queue status mode. This mode is selected by the coprocessor 80187 by sending a logic **low** signal to $\overline{RD}$ pin. The queue status can be used to track the internal status of the queue. The output on QS_0 and QS_1 can be interpreted as shown in Table-8.5.

TABLE - 8.5 : QUEUE STATUS

QS_1	QS_0	Queue operation
0	0	No queue operation
0	1	First opcode byte fetched from the queue
1	0	Empty the queue
1	1	Subsequent byte fetched from the queue

The status signals $\overline{S}_0$, $\overline{S}_1$ and $\overline{S}_2$ will provide information regarding the nature of the bus cycle performed by the processor. The output on $\overline{S}_0$, $\overline{S}_1$ and $\overline{S}_2$ during various bus cycles are listed in Table-8.6.

TABLE - 8.6 : BUS STATUS SIGNALS

S_2	S_1	S_0	Bus cycle
0	0	0	Input acknowledge
0	0	1	Read IO
0	1	0	Write IO
0	1	1	Halt
1	0	0	Instruction fetch
1	0	1	Read data from memory
1	1	0	Write data to memory
1	1	1	Passive (No bus cycle)

The signal $\overline{\text{DEN}}$ is used to enable the external data bus buffers/transceivers and DT/$\overline{\text{R}}$ is used for direction control of data bus buffers.

The ARDY and SRDY are input signals 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. The ARDY pin accepts a rising edge that is asynchronous to CLKOUT and it is active **high**. The SRDY pin accepts an active **high** input synchronized to CLKOUT. When one of the ready input is used, the other should be tied to logic **low**. When both ready inputs are not used, they should be tied to logic **high**.

The $\overline{\text{RES}}$ is the input reset signal to be applied to bring the processor to a known state. The $\overline{\text{RES}}$ pin should be held **low** for atleast 50 milliseconds after power is switched-ON. Whenever $\overline{\text{RES}}$ goes **low**, the processor generates a logic **high** output RESET signal and this signal can be used to reset the peripheral devices in the system. When the processor is reset, the DS, SS, ES, IP and flag register are cleared, **C**ode **S**egment (CS) register is initialized to FFFF$_H$ and queue is emptied. After reset, the processor will start fetching instructions from the 20-bit physical address FFFF0$_H$.

The pins X_1 and X_2 are provided to connect an external quartz crystal. The frequency of the quartz crystal should be double that of internal clock frequency or processor clock frequency. The processor clock signal is also given out through CLKOUT pin for used by the peripheral devices.

The $\overline{\text{LOCK}}$ is an active **low** output signal activated by the LOCK prefix instruction and remains active until the completion of the instruction prefixed by LOCK, in order to prevent other bus masters from gaining control of the system bus. The $\overline{\text{TEST}}$ input is tested by the WAIT instruction. The processor will enter a wait state after execution of the WAIT instruction, and it will resume execution only when $\overline{\text{TEST}}$ is made **low** by an external hardware.

The $\overline{\text{LCS}}$ is a programmable memory chip select signal to select a memory block of size 1 kb to 256 kb with starting address 00000$_H$. The $\overline{\text{UCS}}$ is a programmable memory chip select signal to select a memory block of size 1kb to 256 kb ending with the address FFFFF$_H$. The $\overline{\text{MCS}}_0$ to $\overline{\text{MCS}}_3$ are programmable to select four middle memory blocks of size 8 kb to 512 kb.

The $\overline{\text{PCS}}_0$ to $\overline{\text{PCS}}_6$ are programmable peripheral chip select signals to select seven IO devices. The $\overline{\text{PCS}}_5$ can also be programmed to provide internally latched address bit A_1 and $\overline{\text{PCS}}_6$ can be programmed to provide internally latched address bit A_2.

8.2.2 Architecture of 80186

The architecture (functional block diagram) of 80186 microprocessor is shown in Fig. 8.2. The various functional blocks of 80186 are **B**us **I**nterface **U**nit (BIU), **E**xecution **U**nit (EU), Clock generator, Programmable interrupt controller, Programmable timers, DMA controller and Chip select unit.

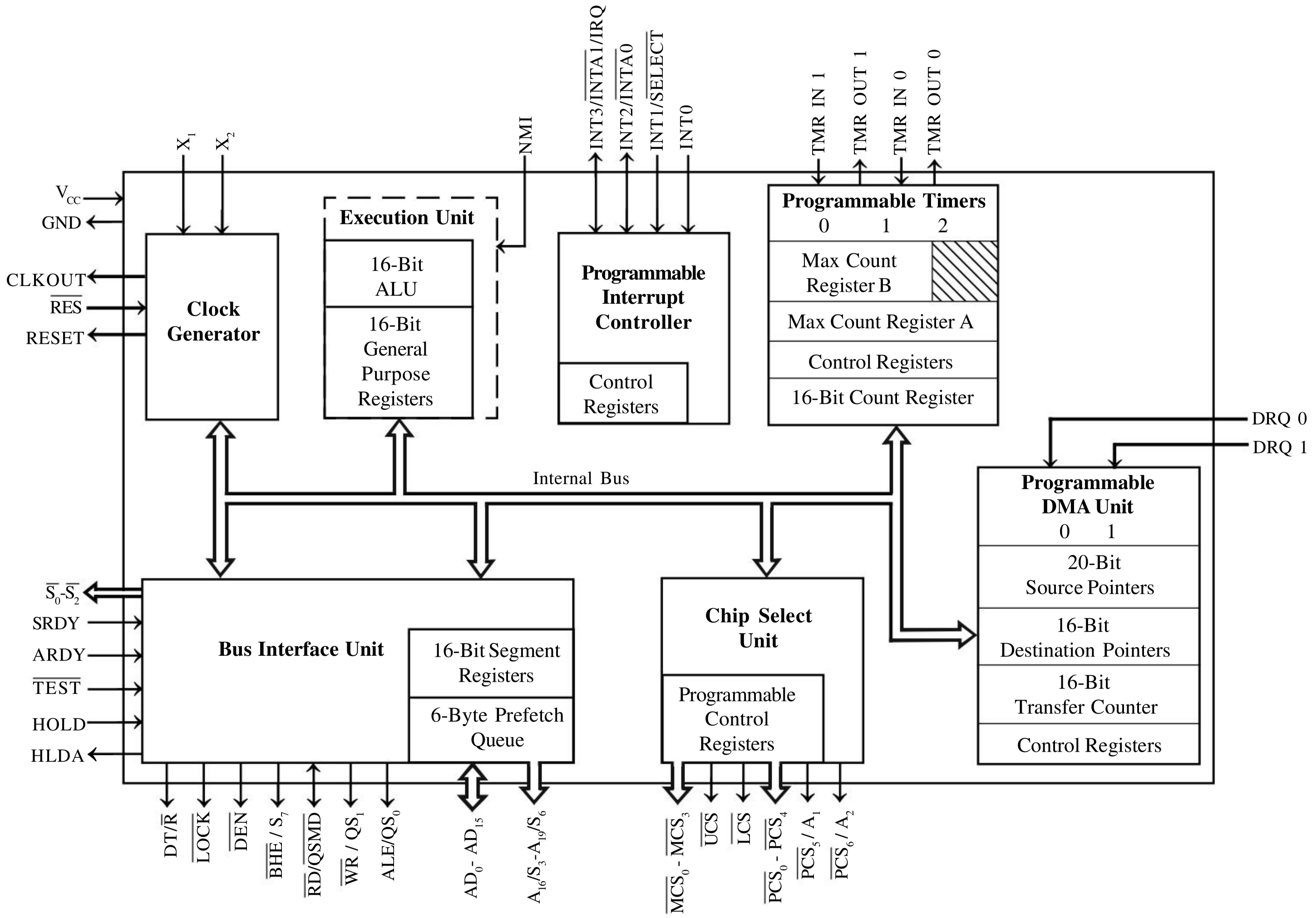


Fig. 8.2 : Functional block diagram of INTEL 80186 microprocessor.

The BIU of 80186 is identical to the BIU of 8086 processor. The BIU consists of a dedicated adder to generate a 20-bit address, 16-bit segment registers CS, DS, SS and ES, logic circuit to produce bus control signals and 6-byte instruction queue. The 20-bit physical address is generated by multiplying the content of one of the segment register by 16_{10} and then adding it to a 16-bit offset. The EU of 80186 is identical to the EU of 8086 processor. The EU consists of 16-bit ALU, Flag register, and general purpose registers AX, BX, CX, DX, SP, BP, SI and DI. The functions of the general purpose registers and the flags of 80186 processor are same as that of 8086 processor.

The on-chip (internal) clock generator consists of a crystal oscillator, a divide-by-two counter, synchronous and asynchronous ready inputs and reset circuitry. An external quartz crystal of frequency double that of the processor clock should be connected through X_1 and X_2 pins to the oscillator of the clock generator. The clock generated by the oscillator is divided by two and used as a processor clock. The clock signal is also given out through the CLKOUT pin for use by peripheral devices. The clock generator also provides the internal timing for synchronizing the ready input signals.

The timer unit consists of three programmable 16-bit timers. The timer-0 and timer-1 can be used to count external events, provide timings for external events, generate waveforms, etc., depending on the mode of operation. The timer-0 and timer-1 can be driven either by a processor clock or by an external clock. The timer-2 can be used only for internal timing operations. It can be used as a clock source for other timers, as a watchdog timer or as a DMA request source. [A watchdog timer is a timer which can internally interrupt the processor after a programmed time interval.]

The DMA controller unit consists of two independent DMA channels. DMA data transfer can be performed between memory spaces or between IO spaces or between memory and IO space. The data can be transferred either in bytes or in words to/from even/odd addresses. Each data transfer consumes two bus cycles, one to read data and the other to write. Each DMA channel has a 20-bit source pointer, a 20-bit destination pointer, a 16-bit count register and a 16-bit control register. The pointers are used to hold the source and destination addresses. The count register is used to program the number of bytes/words to be transferred by DMA. The control register is programmed for the type of DMA and various other functions. The DMA channels can be programmed such that one channel can have higher priority over the other.

The interrupt controller unit arbitrates all internal and external interrupts. The 80186 has five external (hardware) interrupt inputs and they are INT0, INT1, INT2, INT3 and NMI. The external hardware interrupts can be expanded by connecting INTEL 8259 (external Programmable Interrupt Controller) to INT0 and INT1 inputs.

The internal interrupts of 80186 includes INTEL predefined interrupts, software interrupts and interrupts from internal timers and DMA channels. The INTEL predefined interrupts of 80186 includes the five INTEL predefined interrupts of 8086 (Divide error, Single step, NMI, Break point and Interrupt on overflow) and in addition has three predefined interrupts : Array BOUNDS Interrupt, Unused opcode interrupt and ESC opcode interrupt.

The array BOUNDS interrupt occurs if the boundary of an index register is outside the values set up in the memory. The unused opcode interrupt occurs whenever the processor executes any undefined opcode. The ESC opcode interrupt occurs if ESC opcodes are executed.

Each interrupt of 80186 has been allotted a type number and a vector address like that of 8086. The interrupt vector table of 80186 occupies the first 1kb memory space like that of 8086. The type number, vector address and priority of the internal and external interrupts of 80186 are listed in Table-8.7.

TABLE - 8.7 : TYPE NUMBER AND PRIORITIES OF INTERRUPTS OF 80186

Interrupt	Type number	Vector address	Priority level
Divide error	0	00000 - 00003	1
Single step	1	00004 - 00007	1A
NMI	2	00008 - 0000B	1
Breakpoint	3	0000C - 0000F	1
Interrupt on overflow	4	00010 - 00013	1
Array BOUNDS	5	00014 - 00017	1
Unused opcode	6	00018 - 0001B	1
ESC opcode	7	0001C - 0001F	1
Timer-0	8	00020 - 00023	2A
Timer-1	18	00048 - 0004B	2B
Timer-2	19	0004C - 0004F	2C
Reserved	9	00024 - 00027	3
DMA 0	10	00028 - 0002B	4
DMA 1	11	0002C - 0002F	5
INT0	12	00030 - 00033	6
INT1	13	00034 - 00037	7
INT2	14	00038 - 0003B	8
INT3	15	0003C - 0003F	9
80187	16	00040 - 00043	1

Note : *Interrupt priority level 1 is the highest and 9 is the lowest. Some interrupts have the same priority.*

The chip select unit generates the chip select signals for memories and peripherals. This unit provides 6 memory chip select outputs, namely $\overline{\text{UCS}}$, $\overline{\text{LCS}}$, $\overline{\text{MCS}}_0$, $\overline{\text{MCS}}_1$, $\overline{\text{MCS}}_2$ and $\overline{\text{MCS}}_3$. The $\overline{\text{UCS}}$ is used to select the upper/top memory space of size 1kb to 256 kb ending with address FFFFF_H. The $\overline{\text{LCS}}$ is used to select the lower/bottom memory space of size 1kb to 256 kb starting with address 00000_H. The $\overline{\text{MCS}}_0$ to $\overline{\text{MCS}}_3$ can be used to select four address spaces of size 8 kb to 512 kb within 1Mb address space, excluding the address space defined by $\overline{\text{UCS}}$ and $\overline{\text{LCS}}$.

The chip select unit provides seven peripheral chip select signals. Each peripheral chip select signal addresses a 128 byte block of IO address space. The programmable IO address space starts at a base IO (or memory) address programmed by the user. The seven consecutive blocks of 128 bytes starting from this base address will be the IO address space addressed by the seven peripheral chip select signals respectively.

The chip select signals are active for all memory and IO cycles in their programmed areas, whether they are generated by the BIU or the DMA unit.

The 80186 is completely object code compatible with 8086. The instruction set of 80186 consists of the instructions of 8086 and of 10 new instructions, which are listed below:

1.	ENTER	-	Enter a procedure
2.	LEAVE	-	Leave a procedure
3.	BOUND	-	Check if an array index in a register is in range of array
4.	INS	-	Input string byte or string word
5.	OUTS	-	Output string byte or string word
6.	PUSHA	-	Push all registers to stack
7.	POPA	-	Pop all registers from stack
8.	PUSH imm	-	Push immediate(imm) data to stack
9.	IMUL reg,sou,imm	-	Multiply the immediate(imm) data and source(sou) data, and store the result in register(reg)
8.	SHIFT des,imm	-	Shift the destination(des) register /memory contents specified immediate(imm) number of times.

8.3 INTEL 80286

The INTEL 80286 is a 16-bit microprocessor with on-chip memory protection capabilities. The 80286 is an integration of 8086 and memory management unit on a single chip. It is primarily designed for multiuser/multitasking systems. The 80826 is used as a CPU in IBM's personal computers PC/AT and its clones.

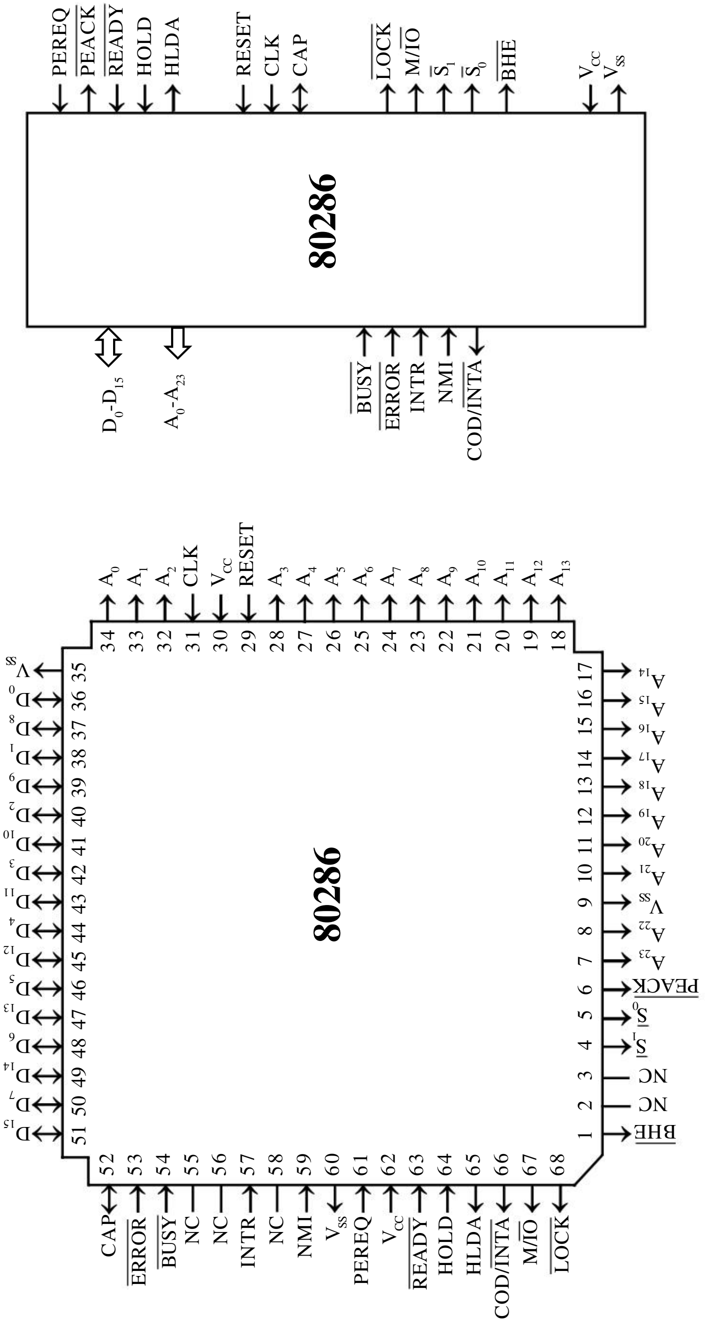
The 80826 has two operating modes : real address mode and protected virtual address mode. In real address mode, the 80286 can address upto 1Mb (Mega-byte) of physical memory address space like 8086. In protected virtual address mode, it can address up to 16 Mb of physical memory address space and 1Gb (Giga-byte) of virtual memory address space.

The instruction set of 80286 includes the instructions of 8086 and 80186, and has some extra instructions to support the operating system and memory management. In real address mode, the 80286 is object code compatible with 8086. In protected virtual address mode, it is source code compatible with 8086 and the software may require some modifications to incorporate the virtual address features. The performance of 80286 is five times faster than that of a standard 8086.

The 80286 is available with maximum internal clock frequency ratings of 4, 6 and 8 MHz.

8.3.1 Pins and Signals of 80286

The 80286 is a 68-pin IC available in ceramic leadless flat package. The pin configuration of 80286 is shown in Fig. 8.3. The pins and signals of 80286 are listed in Table-8.8. The 80286 has nonmultiplexed address and data bus. It has 16 pins (D_0-D_{15}) for data and 24 pins (A_0-A_{23}) for address.



Note : NC - No connection.

Fig. 8.3 : Pin configuration of 80286.

TABLE - 8.8 : PINS AND SIGNALS OF 80286

Pin	Description	Type
D_{15}-D_0	Data	Bidirectional
A_{23}-A_0	Address	Output
$\overline{BHE}$	Bus high enable	Output
$\overline{S}_0$, $\overline{S}_1$	Status signals	Output
M/IO	Memory or IO indicator	Output
$\overline{LOCK}$	Bus priority lock control	Output
RESET	Processor reset input	Input
CLK	Clock input	Input
CAP	Capacitor connection	Bidirectional
$\overline{READY}$	Wait state control	Input
HOLD	Hold request	Input
HLDA	Hold acknowledge	Output
PEREQ	Processor extention request	Input
$\overline{PEACK}$	Processor extention acknowledge	Output
$\overline{BUSY}$	Wait request input from coprocessor	Input
$\overline{ERROR}$	Coprocessor interrupt on error	Input
INTR	Interrupt request	Input
NMI	Non-maskable interrupt	Input
COD/$\overline{INTA}$	Code access/Interrupt acknowledge	Output
V_{cc}	+5-V, Power supply	Input
V_{ss}	0-V, Ground	Output

In 80286-based system, the memory is organized as odd bank and even bank. The odd bank is enabled by the signal $\overline{BHE}$ and the even bank is enabled by the address bit A_0 when it is **low**.

From the control point of view, the 80286 works as 8086 in maximum mode. The 80286 requires an external bus controller 82288 to generate the bus control signals. The status signals $\overline{S}_0$, $\overline{S}_1$ and M/$\overline{IO}$ are decoded by the bus controller to generate memory read, memory write, IO read, IO write, interrupt acknowledge and other bus control signals.

The COD/$\overline{INTA}$ output and M/$\overline{IO}$ can be used to produce early control bus signals. The COD/$\overline{INTA}$ is asserted **low** for interrupt acknowledge and data memory read/write bus cycles. It is asserted **high** for IO read/write and instruction/code read bus cycles.

The 80286 does not have an internal clock generation circuit. Hence an external clock generator 8284 should be employed in 80286-based system to generate the required clock for 80286 processor. The clock generator also supplies the RESET and $\overline{READY}$ inputs to 80286. The function of READY and LOCK of 80286 are similar to READY and LOCK of 8086.

The 80286 has a logic **high** reset to bring the processor to a known state. After a reset, the 80286 processor will work in real address mode and start executing the program stored at $FFFFF0_H$. The reset will also initialize the internal registers as follows:

Flag register : 0002_H
Machine Status Word : $FFF0_H$
Instruction Pointer : $FFF0_H$

CS-register : $F000_H$
DS-register : 0000_H
SS-register : 0000_H
ES-register : 0000_H

The CAP pin has been provided to connect an external filter capacitor for the negative bias voltage generator located internally. The negative bias voltage is required for the substrate of MOS devices in 80286 in order to work at maximum speed. An electrolyte capacitor of rating 0.047 µF, ±20%, 25-V should be connected to a CAP pin. The positive end of capacitor is grounded and the negative end is connected to the CAP pin.

The HOLD input is used by the DMA controller to request the bus to perform DMA transfer. The HLDA signal is the acknowledge signal sent by 80286 to the DMA controller to inform that the bus has been released for performing DMA.

The pins PEREQ, $\overline{\text{PEACK}}$, $\overline{\text{BUSY}}$ and $\overline{\text{ERROR}}$ are used for interfacing coprocessor 80287 with 80286. The coprocessor (80287) will assert PEREQ as **high** whenever it requires a memory data transfer. When the processor (80286) starts performing the data transfer it will send an acknowledge to the coprocessor (80287) by asserting $\overline{\text{PEACK}}$ as **low**.

When the coprocessor executes an instruction, the 80286 will wait (by executing wait instruction) until the $\overline{\text{BUSY}}$ signal is asserted **low** by the coprocessor. Whenever the coprocessor finds some error during processing it will interrupt the processor by asserting $\overline{\text{ERROR}}$ as **low**. In response the processor will perform a type 16_H interrrupt call, which will automatically execute a procedure written for the desired response to error condition.

8.3.2 Architecture of 80286

The architecture (functional block diagram) of 80286 is shown in Fig. 8.4. The 80286 has four separate processing units: **B**us **U**nit (BU), **I**nstruction **U**nit (IU), **E**xecution **U**nit (EU) and **A**ddress **U**nit (AU).

The **B**us **U**nit (BU) performs all the memory and IO read/write operations. Whenever the bus is free, the BU will fetch instruction bytes and put them in 6-byte prefetch queue. When a jump/call instruction is encountered, the BU will dump/clear the queue and starts filling it from the jump/call address. The BU also controls transfer of data to and from the coprocessor 80287.

The **I**nstruction **U**nit (IU) will read the prefetched instructions from the BU and decode them, and then put them in the instruction queue. The instruction queue can accommodate up to three decoded instructions. This additional pipelining in 80286 increases the speed of processing of 80286, when compared to an 8086.

The **E**xecution **U**nit (EU) will read the decoded instructions from the IU and execute them sequentially. It directs the BU to fetch memory or IO operands whenever needed. The EU has a 16-bit ALU, a set of registers identical to that of 8086 and in addition it has a 16-bit **M**achine **S**tatus **W**ord (MSW). The MSW is used to switch from real address mode to protected virtual address mode after a reset.

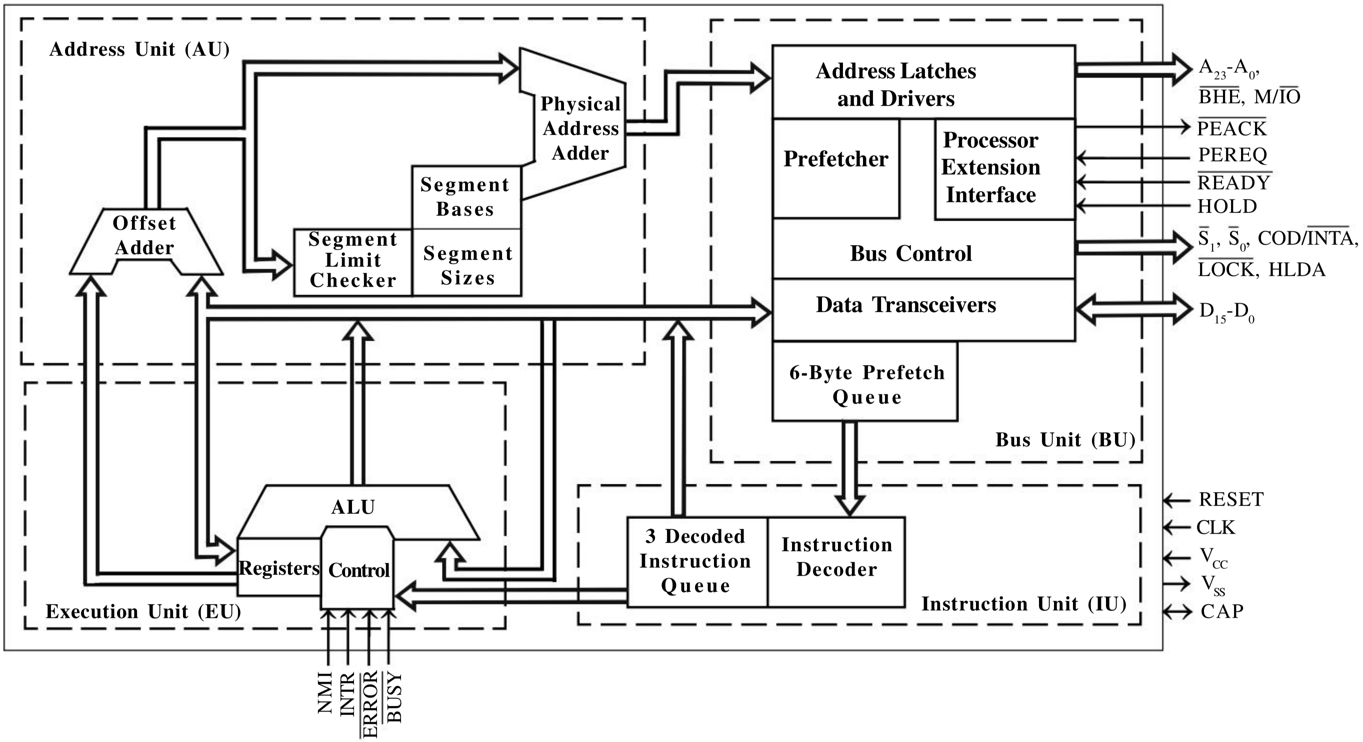


Fig. 8.4 : Functional block diagram of an INTEL 80286 microprocessor.

The general purpose registers available in the EU of 80286 are AX, BX, CX, DX, SP, BP, SI and DI. The function of these registers are same as that of 8086. Also, the EU has a 16-bit **I**nstruction **P**ointer (IP) and a 16-bit flag register. The format of the flag register of 80286 is shown in Fig. 8.5.

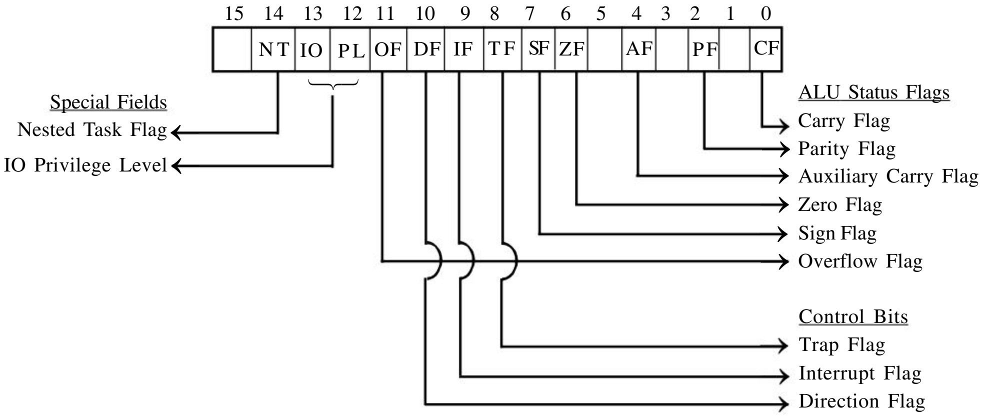


Fig. 8.5 : Flag register of 80286.

The **A**ddress **U**nit (AU) takes care of computing physical address of memory or IO and sends the address to the BU. The 80286 can operate either in real address mode or protected virtual address mode. After a reset, the 80286 processor will work in real address mode. In real address mode, the address computation is similar to that of 8086 and the processor uses only lower 20 bits of the address. Therefore, the maximum physical address space in this mode is only 1Mb. The AU has four numbers of 16-bit segment registers like 8086 and they are **C**ode **S**egment (CS), **D**ata **S**egment (DS), **S**tack **S**egment (SS) and **E**xtra **S**egment (ES) registers. The program/ code address is computed by multiplying the content of the CS by 16_{10} and then adding it to 16-bit offset available in IP. The data address is computed by multiplying the content of the DS/ES by 16_{10} and then adding it to a 16-bit offset specified by the instruction. The stack address is computed by multiplying the content of SS by 16_{10} and then adding to a 16-bit offset specified by the instruction.

The processor can be switched from real address mode to **P**rotected **V**irtual **A**ddress **M**ode (PVAM) by loading an appropriate word in MSW register. In PVAM, the address unit functions as a complete **M**emory **M**anagement **U**nit (MMU). In PVAM, the 80286 uses all the 24 address lines and can access up to 16 Mb (2^{24} = 16 Mega) of physical memory address space and 1Gb of virtual address space. In PVAM mode, each memory location is represented by a 32-bit virtual address (or logical address). The MMU translates the 32-bit virtual address to a 24-bit physical address. The 32-bit virtual address has a 16-bit selector and 16-bit offset. The 16-bit selector is used to fetch a descriptor from a descriptor table. The descriptor contains a 24-bit segment base address which is added to the 16-bit offset to get a 24-bit physical address.

> *Note : The 80286 has a 13-bit index for descriptor and allows two descriptor tables (Global Descriptor Table [GDT] and Local Descriptor Table [LDT]). Therefore, the processor allows $2^{13} \times 2 = 2^{14}$ = 16 k descriptors. Each descriptor can define a segment of size 1kb to 64 kb. Hence total virtual space is 16 k × 64 kb = $2^{14} \times 2^{16}$ bytes = $2^{10} \times 2^{20}$ bytes = 1024 × 1Mb = 1Gb.*

8.3.3 Real Address Mode of 80286

After a reset, the 80286 processor will work in real address mode. The working of 80286 in real address mode is similar to that of 8086 in maximum mode. In real address mode, the 80286 will directly execute the 8086 machine code programs with only minor modifications. In 80286, the execution will be faster due to extensive pipelining and other hardware improvements.

In real address mode, the 80286 computes the memory address similar to that of 8086. It uses 20-bit address to access memory. The 20-bit physical address is computed by multiplying a 16-bit segment base address by 16_{10} and adding it to a 16-bit offset. Hence, 80286 can address up to 1Mb (2^{20} = 1 Mega) of physical memory space. In this mode, the upper 4 bits of memory address bus are ignored.

The 80286 has 256 interrupts like 8086 and each interrupt has a type number in the range 0 to 255. In real address mode, the execution of interrupt is similar to that of 8086 processor. The interrupt vector table is located in the first 1kb of memory space. When an interrupt occurs, the processor multiplies the interrupt type number by four to get the address of the vector table. From this address in the vector table, the segment base address and offset address of the interrupt service procedure/subroutine are read and loaded in CS and IP respectively. The INTEL predefined interrupts of 80286 includes the predefined interrupts of 8086 and has additional few predefined interrupts. The predefined interrupts of the 80286 are listed in Table-8.9

The real address mode is mainly used to initialize peripheral devices, load the main part of the operating system from disk into memory, enable interrupts and enter the protected virtual address mode.

TABLE - 8.9 : PREDEFINED INTERRUPTS OF 80286

Interrupt	Type number
Divide error	0
Single step	1
NMI	2
Breakpoint	3
Interrupt on overflow	4
Array bounds	5
Invalid opcode	6
Processor extension not available	7
Interrupt table limit too small	8
Processor extension segment overrun	9
Invalid task state segment	10
Segment not present	11
Stack segment overrun or not present	12
Segment overrun	13
Processor extension error	16

8.3.4 Protected Virtual Address Mode of 80286

After a reset, the 80286 processor will work in real address mode and it can be made to work in **P**rotected **V**irtual **A**ddress **M**ode (PVAM) by loading an appropriate word in the **M**achine **S**tatus **W**ord (MSW) register. The format of the MSW is shown in Fig. 8.6. In order to enter PVAM, frame a word such that the most significant bit (PE) is one (i.e., to enter PVAM, PE bit is set to one) and load this word in a register/memory, and then execute the instruction LMSW (**L**oad **MSW**). Once the processor enters PVAM, the only way to get back to the real address mode is by resetting the system. In PVAM, the **A**ddress **U**nit (AU) functions as a **M**emory **M**anagement **U**nit (MMU). In PVAM, 24-bit address is used to access memory and this address is computed by adding a 24-bit segment base address and a 16-bit offset. The segment base address is obtained from a descriptor, which is stored in the descriptor table.

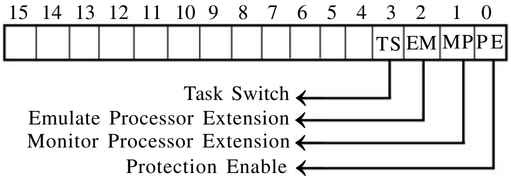


Fig. 8.6 : Format of Machine Status Word (MSW).

Memory Management in PVAM

In PVAM, the 80286 allows the user to create memory segments of length/size 1kb to 64 kb for each task/program. A size or limit is given to each segment when it is created. The 80286 allows 64 k memory segments and all these segments are not available in the physical memory space at the same time. Hence, these segments are called logical/virtual segments. The segments currently being used by a task/program are kept in physical memory. The segments which are not currently used will be in secondary memory like hard disk and they are brought to physical memory whenever needed.

In PVAM, the user has to provide a 32-bit virtual address for memory. The lower word (lower 16-bit) of virtual address is offset address and the upper word (upper 16-bit) is called selector, which is loaded in segment register. The 16-bit selector is used to fetch an 8-byte descriptor from the descriptor table. The descriptor contains the 24-bit segment base address which is added to the 16-bit offset to get a 24-bit physical address. Thus, address calculation in PVAM involves selector, descriptor and descriptor table.

Selector

When a program is assembled for execution on an 80286 processor in PVAM, each segment is assigned a unique 16-bit selector. The format of the selector is shown in Fig. 8.7.

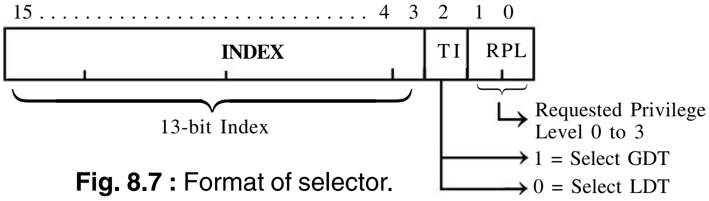
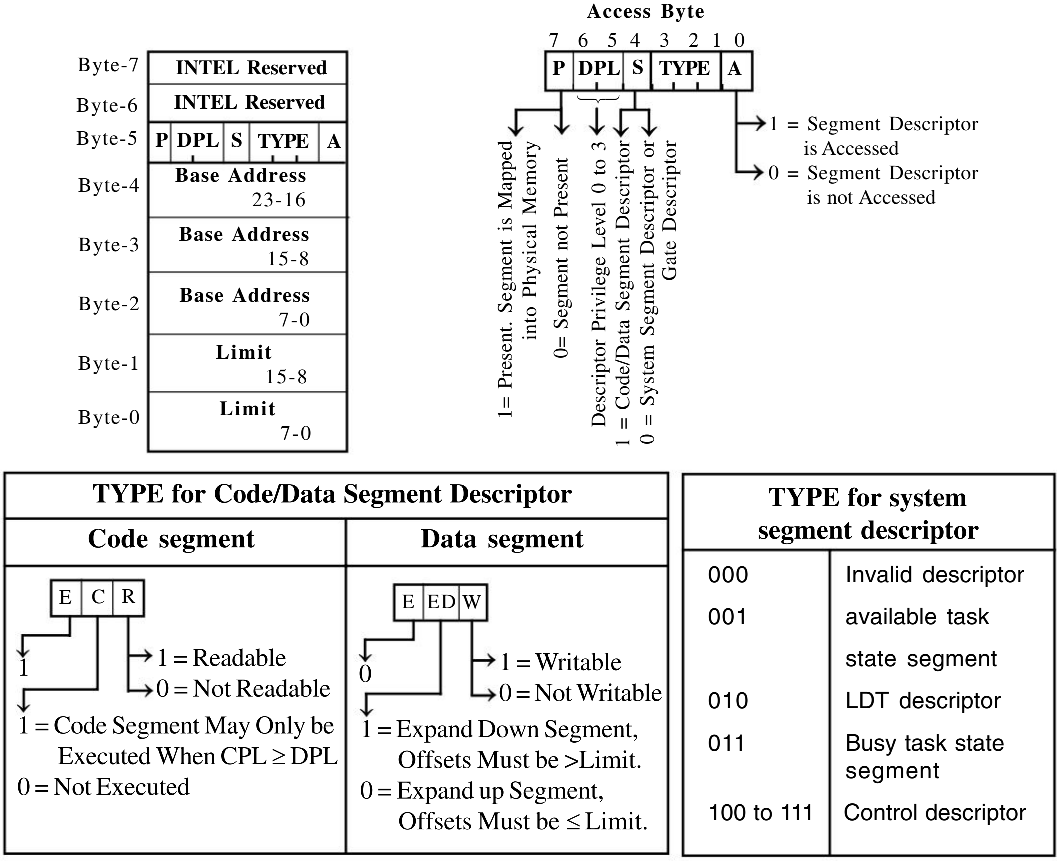


Fig. 8.7 : Format of selector.

The 2-bit RPL (**R**equested **P**rivilege **L**evel) field is used by operating system for implementing the 80286's protection features and is not used in address calculation. The 1-bit TI field is used to select one of the two descriptor tables. When TI field is one, **G**lobal **D**escriptor **T**able (GDT) is selected and when TI field is zero, **L**ocal **D**escriptor **T**able (LDT) is selected. The 13-bit index is used to create $2^{13} = 8192_{10}$ = 8 k descriptors in each table. The 13-bit index is multiplied by 8 to get the address of the descriptor in a descriptor table.

Descriptor

When a program is assembled for execution on an 80286 processor in PVAM, a unique descriptor is produced for each segment. The length of the descriptor is 8-byte. The format of descriptor is shown in Fig. 8.8.


Byte-7
INTEL Reserved
Byte-6
INTEL Reserved
Byte-5
P DPL S TYPE A
Byte-4
Base Address 23-16
Byte-3
Base Address 15-8
Byte-2
Base Address 7-0
Byte-1
Limit 15-8
Byte-0
Limit 7-0
Access Byte
7 6 5 4 3 2 1 0
P DPL S TYPE A
1= Present. Segment is Mapped into Physical Memory
0= Segment not Present
Descriptor Privilege Level 0 to 3
1 = Code/Data Segment Descriptor
0 = System Segment Descriptor or Gate Descriptor
1 = Segment Descriptor is Accessed
0 = Segment Descriptor is not Accessed
TYPE for Code/Data Segment Descriptor
Code segment
E C R
1
1 = Readable
0 = Not Readable
1 = Code Segment May Only be Executed When CPL ≥ DPL
0 = Not Executed
Data segment
E ED W
0
1 = Writable
0 = Not Writable
1 = Expand Down Segment, Offsets Must be >Limit.
0 = Expand up Segment, Offsets Must be ≤ Limit.

TYPE for system segment descriptor	
000	Invalid descriptor
001	available task state segment
010	LDT descriptor
011	Busy task state segment
100 to 111	Control descriptor

Fig. 8.8 : Format of a descriptor.

The byte-0 and byte-1 of the descriptor contains the length/limit of the segment in bytes. When an attempt is made to access a location beyond the limit, the MMU will generate an interrupt. The byte-2, byte-3 and byte-4 of the descriptor contain the 24-bit segment base address. Byte-5 of the descriptor is called the access byte and it contains information regarding privilege level, access right and the type of segment. Byte-6 and byte-7 are reserved for future expansion and should be filled with zero for compatibility with a higher version of 80x86 family.

Descriptor table

In 80286-based system, the descriptors are stored in descriptor tables. There are two types of descriptor tables: **G**lobal **D**escriptor **T**able (GDT) and **L**ocal **D**escriptor **T**able (LDT). The descriptor tables are created by system software and stored in memory. The processor can read the descriptors from the tables in memory, whenever needed.

The GDT contains the descriptors for the operating system segments and descriptors for segments which need to be accessed by all user tasks. The LDT contains descriptors for each task. A four level protection scheme can be used to protect the operating system descriptors in the GDT from unauthorized access by user tasks.

Address Translation in 80286

The 80286 has four numbers of segment descriptor cache registers which are invisible to the programmer. The size of these register is 48 bits (so that it can hold the 6-byte descriptor).

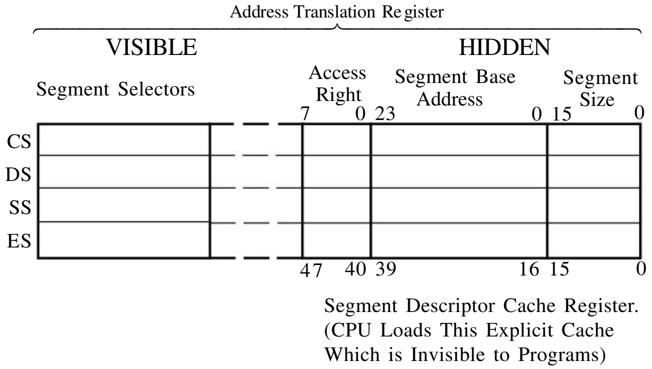

Address Translation Re gister
VISIBLE
HIDDEN
Segment Selectors
Access Right
Segment Base Address
Segment Size
7
0
23
0
15
0
CS
DS
SS
ES
47
40
39
16
15
0
Segment Descriptor Cache Register. (CPU Loads This Explicit Cache Which is Invisible to Programs)

Fig. 8.9 : Address translation register.

When the 80286 is operating in PVAM, descriptors must be copied to the processor from tables in memory and loaded in the invisible segment descriptor cache registers. The descriptors are then used for producing and checking physical addresses.

The 80286 also has internal registers for storing base addresses of descriptor tables and they are GDTR and LDTR. The 80286 keeps the base addresses and limits for the descriptor tables currently in use in these internal registers.

The **G**lobal **D**escriptor **T**able **R**egister (GDTR) contains the 24-bit base address and limit for the table containing the global address space description. This register is initialized with LGDT (**L**oad **GDT**) instruction when the system is booted.

The **L**ocal **D**escriptor **T**able **R**egister (LDTR) in the 80286 contains the base address and limit of the local descriptor table for the task currently being executed. The LLDT(**L**oad **LDT**) instruction is used to load this register when the system is booted.

Each address in the PVAM is represented by a 32-bit virtual address. The virtual address consists of a 16-bit selector and 16-bit offset. To access a segment, the selector for that segment is loaded into the visible part of the appropriate segment register in the 80286. The 80286 then automatically multiplies the index value of the selector by 8 and adds the result to a descriptor table base address in its GDTR or LDTR. The 80286 then fetches the segment descriptor from the resultant address in a descriptor table. Byte-0 to byte-5 of the descriptor is loaded into an invisible segment descriptor cache register. The descriptor has a 24-bit base address. This base address is added directly to the offset part of the virtual address to produce the physical address of the desired byte or word. The conversion of virtual address to physical address is diagrammatically shown in Fig. 8.10.

In a system which always runs the same program, the physical base addresses in descriptors are fixed by program development tools when the program is built. In a general purpose system which runs the same program, the physical base addresses in the descriptors may be changed by the operating system when a program is loaded into the memory to be run. This is done so that the program can be loaded into available memory without disturbing the programs or tasks already present.

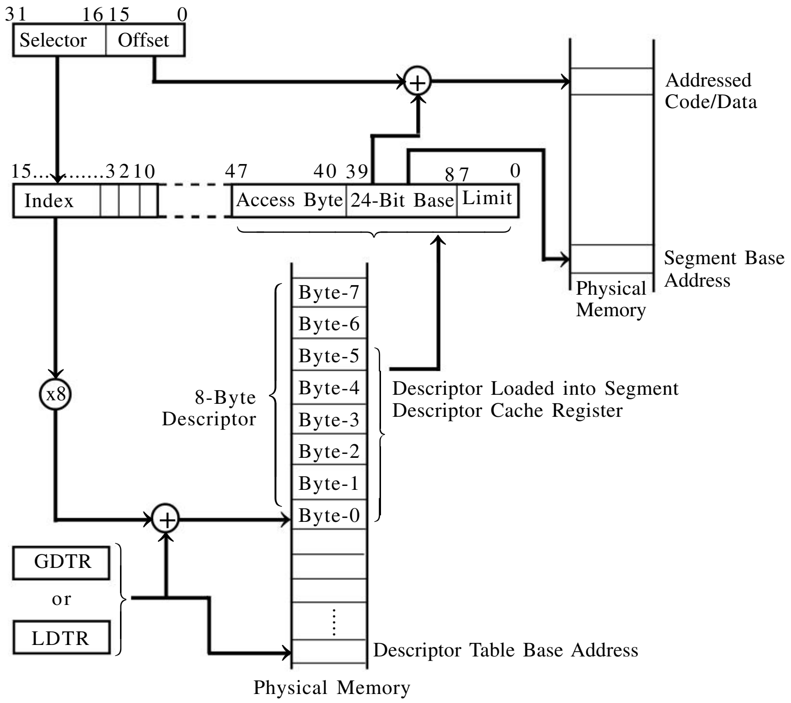


Fig. 8.10 : Translation of 32-bit virtual address to 24-bit physical address in 80286.

8.4 INTEL 80386 MICROPROCESSOR

The 80386 is a 32-bit microprocessor and it is an improved version of 80286 with software compatibility with 8086, 80186 and 80286. The major improvements in 80386 over 80286 are the following:

- The processor registers and ALU are 32 -bit wide and the instruction set is extended to support 32-bit addresses and data.
- The main memory and the data path to memory can be 32- bit wide, so instructions and data read/write operations will be two times faster.
- The maximum size of physical memory is extended from 16 Mb (2^{24} bytes) to 4 Gb (2^{32} bytes).
- Since 80386 runs at higher clock frequency, faster execution speed is obtained and most instructions take fewer clock cycles to execute.
- The on-chip memory management supports paging.

The 80386 is available in two versions: 80386DX and 80386SX. The internal architecture of both the versions of 80386 are same, but they differ only in external address and data bus. The 80386DX has separate external 32-bit data bus and address bus. The 80386SX has external 16-bit data bus and 24-bit address bus. The 80386DX is called the full version of 80386 and 80386SX is called the reduced bus version of 80386. The 80386SX was developed after the 80386DX for applications that did not require an external 32-bit bus and at the same time had the advantage of internal 32-bit computation.

The 80386SX can address up to 16 Mb (2^{24} = 16 M) of physical memory space and memory is organized as two banks of 8 Mb. The 80386DX can address up to 4 Gb (2^{32} = 4 G) of physical memory space and memory is organized as four banks of 1 Gb. Both 80386SX and 80386DX have virtual address space of 64 Tera bytes (2^{46} = 64 T).

The 80386 can work in three modes : real address mode, protected virtual address mode and virtual 8086 mode. The 80386 processor will enter real address mode after a hardware reset and in this mode it works as fast as an 8086 processor with a few additional new instructions. The real address mode is mainly used for initialization and enters into the **P**rotected **V**irtual **A**ddress **M**ode (PVAM).

In PVAM, the 80386 works as a 32-bit processor and all instructions and features of 80386 are available in this mode. While working in PVAM, the processor can switch to **V**irtual 80**86** (V86) mode to run 8086 applications and then return to PVAM. In V86 mode, the processor can run 8086 applications with protection features of 80386.

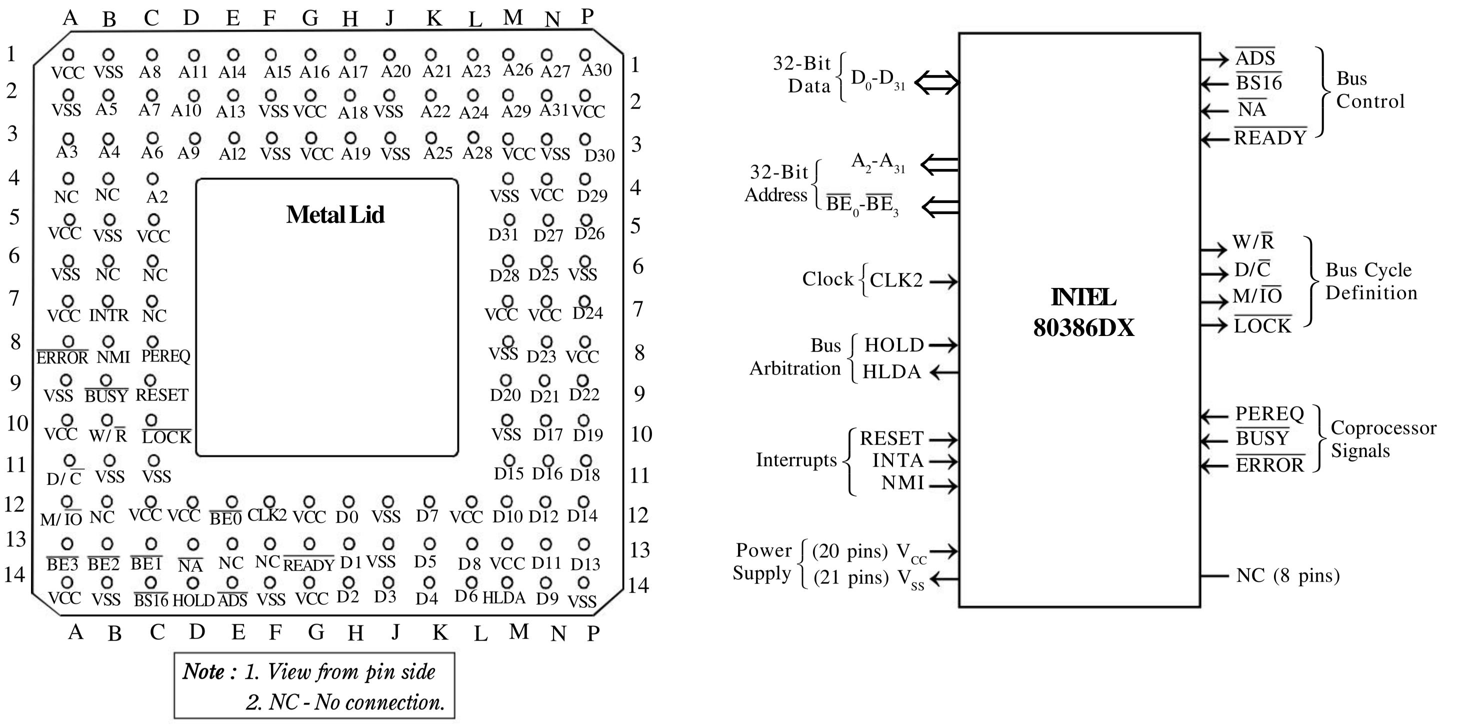
The 80386 is available with maximum clock speed rating of 12.5, 16, 20, 25 or 33 MHz.

8.4.1 Pins and Signals of 80386

The 80386DX is a 132-pin IC available in **P**in **G**rid **A**rray (PGA) package. The pin configuration of an 80386DX is shown in Fig. 8.11. The 80386SX is a 100-pin IC available in plastic quad flatpack package. The pin configuration of 80386SX is shown in Fig. 8.12. The signals of 80386DX are listed in Table-8.8.

The 80386DX has 32 pins for data transfer from/to memory or IO. It can access byte/word/double word from memory or IO in one bus cycle.

The 80386DX has 30 pins for addressing 1 Gb (2^{30} = 1G) of physical address space and four byte/bank enable signals $\overline{BE}_0$ to $\overline{BE}_3$ to enable four memory banks, each of size 1 Gb. The signals $\overline{BE}_0$ to $\overline{BE}_3$ are generated internally by decoding the address bits A_0 and A_1.



	A	B	C	D	E	F	G	H	J	K	L	M	N	P	
1	VCC	VSS	A8	A11	A14	A15	A16	A17	A20	A21	A23	A26	A27	A30	1
2	VSS	A5	A7	A10	A13	VSS	VCC	A18	VSS	A22	A24	A29	A31	VCC	2
3	A3	A4	A6	A9	A12	VSS	VCC	A19	VSS	A25	A28	VCC	VSS	D30	3
4	NC	NC	A2									VSS	VCC	D29	4
5	VCC	VSS	VCC									D31	D27	D26	5
6	VSS	NC	NC									D28	D25	VSS	6
7	VCC	INTR	NC									VCC	VCC	D24	7
8	$\overline{ERROR}$	NMI	PEREQ									VSS	D23	VCC	8
9	VSS	$\overline{BUSY}$	RESET									D20	D21	D22	9
10	VCC	W/$\overline{R}$	$\overline{LOCK}$									VSS	D17	D19	10
11	D/$\overline{C}$	VSS	VSS									D15	D16	D18	11
12	M/$\overline{IO}$	NC	VCC	VCC	$\overline{BE0}$	CLK2	VCC	D0	VSS	D7	VCC	D10	D12	D14	12
13	$\overline{BE3}$	$\overline{BE2}$	$\overline{BE1}$	$\overline{NA}$	NC	NC	$\overline{READY}$	D1	VSS	D5	D8	VCC	D11	D13	13
14	VCC	VSS	$\overline{BS16}$	HOLD	$\overline{ADS}$	VSS	VCC	D2	D3	D4	D6	HLDA	D9	VSS	14
	A	B	C	D	E	F	G	H	J	K	L	M	N	P	

Note : 1. View from pin side
2. NC - No connection.

Fig. 8.11 : Pin description of INTEL 80386DX microprocessor.

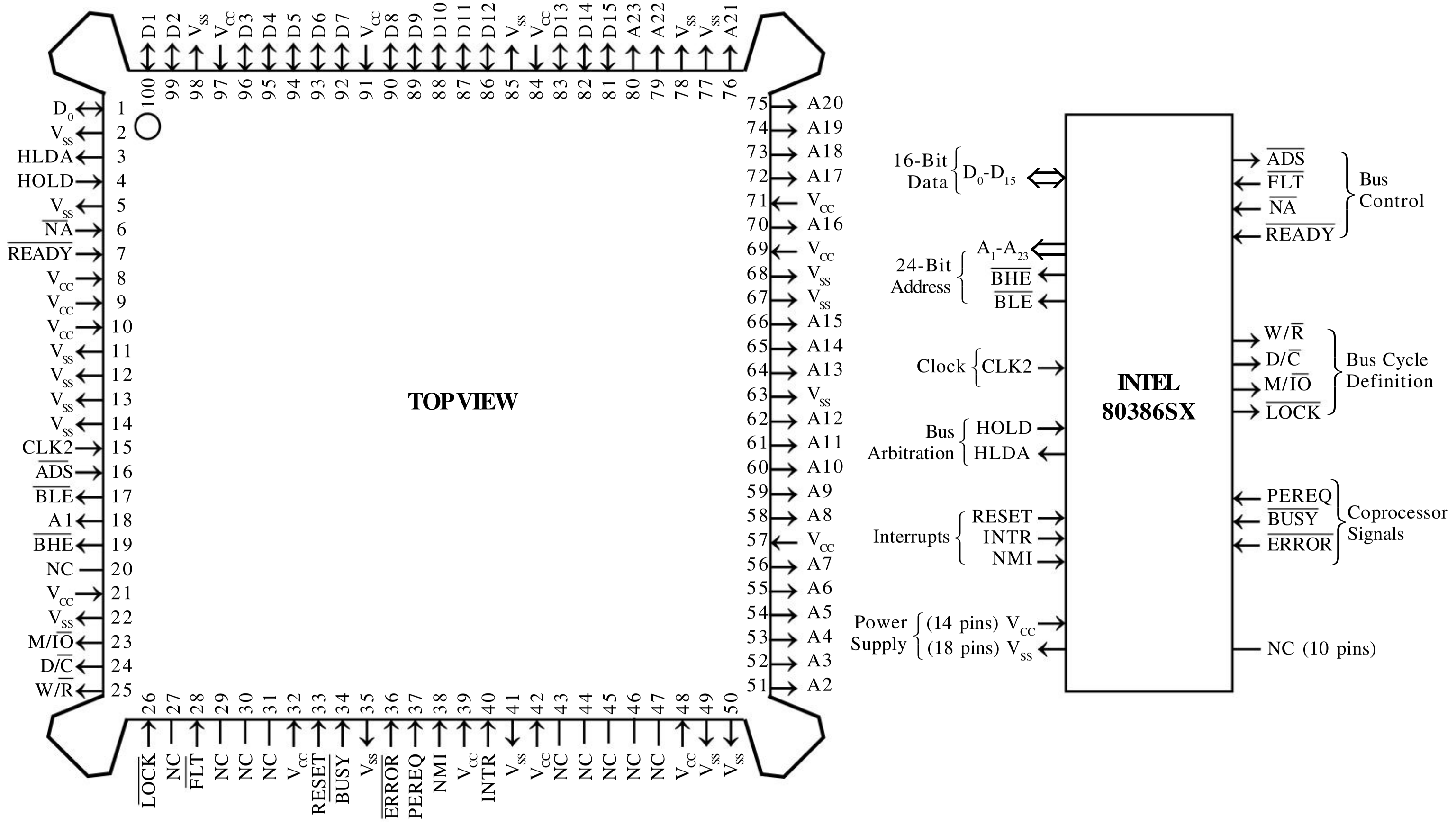


Fig. 8.12 : Pin description of INTEL 80386SX microprocessor.

TABLE - 8.10 : SIGNALS OF 80386DX MICROPROCESSOR

Signal	Function	Type
D_{31}-D_0	32-bit data bus	Bidirectional
A_{31}-A_2	30-bit address bus. Used to address 1 Gb memory space	Output
BE_3-BE_0	Byte/Bank enables. Used to enable four memory banks each of size 1 Gb	Output
CLK2	Clock input	Input
W/$\overline{R}$	Write/Read control	Output
D/$\overline{C}$	Data/Control bus cycle indicator	Output
M/$\overline{IO}$	Memory or IO bus cycle indicator	Output
$\overline{LOCK}$	Bus priority lock control	Output
$\overline{ADS}$	Address status	Output
$\overline{NA}$	Next address request input	Input
$\overline{READY}$	Wait state request	Input
$\overline{BS16}$	Bus size 16. Used to select 32/16-bit data bus	Input
HOLD	Bus request input	Input
HLDA	Bus hold acknowledge	Output
$\overline{BUSY}$	Busy signal from coprocessor	Input
$\overline{ERROR}$	Error signal/Interrupt from coprocessor	Input
PEREQ	Processor extension (coprocessor) data transfer request	Input
RESET	Processor reset	Input
INTR	Maskable interrupt request	Input
NMI	Nonmaskable interrupt request	Input
V_{CC}	+5-V, Power supply input	Input
V_{SS}	0-V, Power supply ground	Output
NC	No connection	-

The 80386 does not have an internal clock generator. The clock signal for 80386 is generated by using an external clock generator 82384 and supplied through the CLK2 pin. The clock input to the CLK2 pin should have a frequency double that of internal clock frequency. The processor divides the clock signal by two and uses them for internal operations.

The W/$\overline{R}$ is used to indicate write/read operation. During write operation, it is asserted **high** and during read operation, it is asserted **low**. The D/$\overline{C}$ signal is asserted **high** during data read/write bus cycles and it is asserted **low** during instruction fetch, interrupt acknowledge and halt bus cycles. The M/$\overline{IO}$ is used to indicate memory or IO access. During memory read/write operation, it is asserted **high** and during IO read/write operation, it is asserted **low**.

The $\overline{LOCK}$ signal is asserted **low** by the processor during execution of the instruction prefixed by LOCK. This prevents other bus masters from taking control of the bus during execution of important instructions.

The $\overline{ADS}$ is asserted **low** whenever an address is output by the processor. The signal $\overline{NA}$ is used for address pipelining and when it is asserted **low**, the 80386 outputs the address of the next instruction/data in the current bus cycle. The $\overline{\text{READY}}$ signal is used to introduce wait states in bus cycles. The $\overline{\text{BS16}}$ is used to interface a 16-bit bus to a 32-bit bus. It is tied to logic **low** for 16-bit external data bus and tied to logic **high** for 32-bit external data bus.

The HOLD and HLDA signals are used for DMA data transfer. The signals $\overline{\text{BUSY}}$, $\overline{\text{ERROR}}$ and PEREQ are used for 80387 coprocessor interface. The functions of these pins are similar to that of 80286.

The 80386 has logic **high** RESET input. Whenever the processor is reset, it is initialized to execute instructions from the memory location FFFF FFF0_H in the real mode. The 80386 has one maskable interrupt request input INTR, which can be expanded using interrupt controller 8259 and one nonmaskable interrupt request input, NMI.

8.4.2 Architecture of 80386 Microprocessor

The architecture of 80386 is shown in Fig. 8.13. It has a highly pipelined architecture with six functional units operating in parallel. In 80386 processor fetching, decoding, execution, memory management (address computation) and bus access for several instructions are performed simultaneously. The six functional units of 80386 are:

- Bus interface unit
- Instruction prefetch unit
- Instruction decode unit
- Execution unit
- Segmentation unit
- Paging unit

The bus interface unit generates signals for memory and IO interface. This unit generates the control signals for the current bus cycle based on internal requests for fetching instructions from the instruction prefetch unit and transferring data from the execution unit. The physical address of memory and IO are output through the bus interface unit and also the data transfer to the external world takes place through this unit.

Whenever the bus is free, the instructions are fetched and stored in the 16-byte code queue in the instruction prefetch unit. The instruction decode unit reads the instructions from the prefetch unit and decodes them. The decoded instructions are then stored in the queue in decode unit. The queue in the decode unit can accommodate three decoded instructions. The queues in the instruction prefetch and decode units are FIFO queues.

The execution unit reads the decoded instructions from the decode unit and processes them. The execution unit consists of a control unit, data unit and a protection test unit. The control unit contains the microcode stored in the ROM for the instructions, flag register and generates internal control signals. The data unit includes an ALU, a 64-bit barrel shifter and eight general purpose registers. The data unit performs the operations requested by the control unit. The barrel shifter is used for fast shift, rotate, multiply and divide operations. The protection test unit checks for segmentation violations under the control of the microcode.

The segmentation and paging units can be considered as MMU (**M**emory **M**anagement **U**nit) of 80386. In real address mode, the paging is disabled and the segmentation unit computes the address similar to that of 8086 or 80286 in real mode. In **P**rotected **V**irtual **A**ddress **M**ode (PVAM) the segmentation unit computes a 32-bit linear address using an offset, selector and descriptor (similar to that of 80286). In PVAM mode, each memory location is represented by a

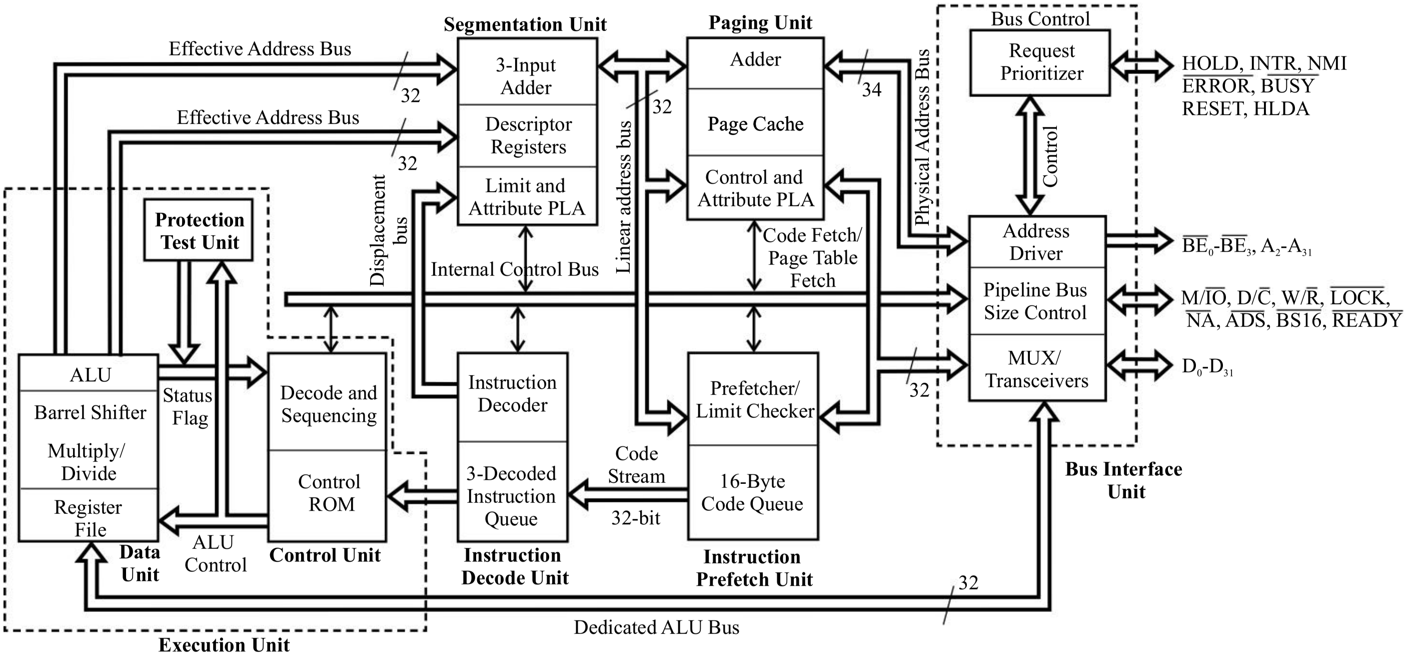


Fig. 8.13 : Architecture of an 80386 microprocessor.

48-bit virtual address. The MMU translates the 48-bit virtual address to the 32-bit physical address. The 48-bit virtual address has a 16-bit selector and 32-bit offset. The 16-bit selector is used to fetch a descriptor from a descriptor table. The descriptor contains a 32-bit segment base address which is added to a 32-bit offset to get a 32-bit linear address. If the paging unit is not enabled, then this linear address is used as physical address. If paging is enabled, then the paging unit will translate the 32-bit linear address to a 32-bit physical address. The paging provides an additional memory management mechanism to handle very large segments.

> *Note : The 80386 has 13-bit index for descriptor and allows two descriptor table (Global Descriptor Table [GDT] and Local Descriptor Table [LDT]). Therefore, the processor allows $2^{13} \times 2 = 2^{14} = 16$ k descriptors. Each descriptor can define a segment of maximum size 4 Gb. Hence, total virtual address space is 16 k × 4 Gb = 64 Tb (terabyte).*

8.4.3 Registers of 80386 Microprocessor

The 80386 processor has 32 internal registers and they can be classified into following seven categories:

- General purpose registers
- Segment registers
- Instruction pointer and flag register
- Control registers
- System address and segment registers
- Debug registers
- Test registers

Most of the 80386 registers are 32-bit registers. The registers of 80386 are a superset of 8086, 80186 and 80286 registers and so all the registers of 8086, 80186 and 80286 are contained within the 32-bit registers of 80386. The general purpose registers, segment registers, instruction pointer and flag register are called base architecture registers. Figure 8.14 shows the base architecture registers of 80386.

General Purpose Registers

The 80386 has eight numbers of 32-bit general purpose registers and they are EAX, EBX, ECX, EDX, ESI, EDI, EBP and ESP. The least significant 16 bits of these registers can be accessed separately with their 16-bit names AX, BX, CX, DX, SI, DI, BP and SP.

Each of the 16-bit registers AX, BX, CX and DX can be accessed as two numbers of 8-bit registers. The lower 8 bits of these registers can be accessed with the name AL, BL, CL and DL. The higher 8 bits of these registers can be accessed with the name AH, BH, CH and DH.

Segment Register

The 80386 has six segment registers to address six segments of memory at any given time. The segment registers of 80386 includes the four segment registers (CS, SS, DS and ES) of 80286 and has two additional data segment registers FS and GS registers. The four data segment registers DS, ES, FS and GS registers can be used to access four separate data areas and allow programs to access different types of data structures. In real address mode, the segment registers will hold the segment base address like that of 8086. In PVAM, the segment registers will hold the selectors like that of 80286. The selector in CS and SS registers indicates the current code and stack segment respectively. The selectors in DS, ES, FS and GS registers indicate the current data segments.

Each segment register has a program invisible 64-bit register called segment descriptor register. These registers are used to hold the 8-byte descriptor of the current memory segment in PVAM.

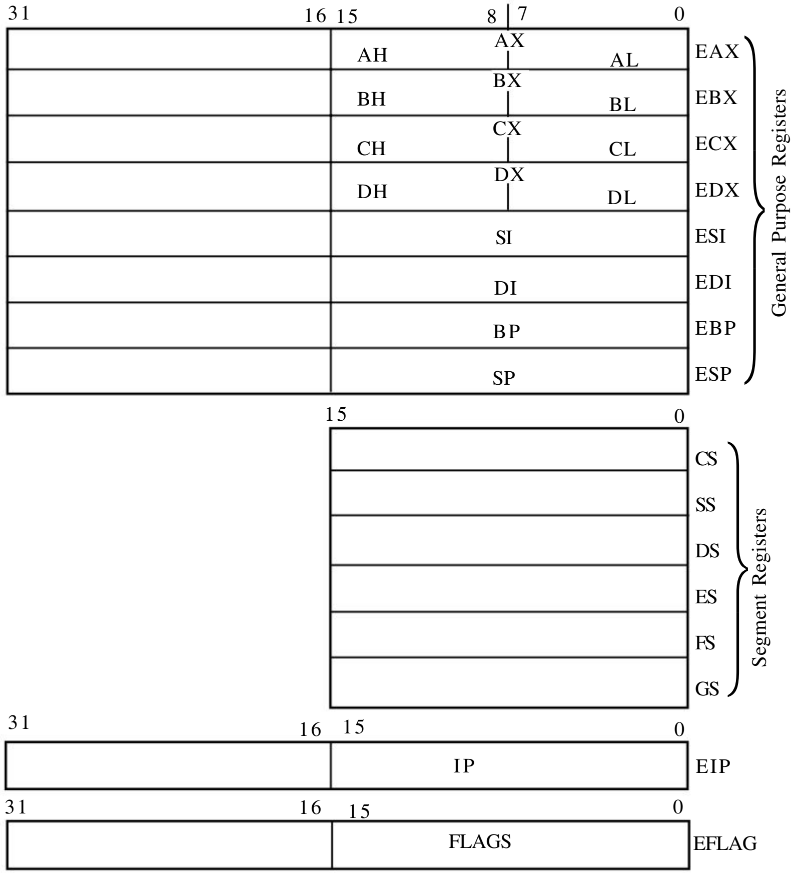


Fig. 8.14 : Base architecture registers of an 80386.

Instruction Pointer and Flag Register

The 80386 has a 32-bit instruction pointer and it is named as EIP. It is used to hold a 32-bit offset. The lower 16 bits of EIP is called 16-bit instruction pointer, IP, which is used in real address mode.

The 80386 has a 32-bit flag register and it is called EFLAG. The format of flag register of 80386 is shown in Fig. 8.15. The flags of 80386 can be classified into three groups: status flags, control flags and system flags. The status flags are CF, AF, PF, ZF, SF and OF. The control flags are TF, DF and IF. (The status and control flags are same as that of 8086.) The system flags are ID, VIP, VIF, AC, VM, RF, NT and IOPL. The system flags control IO access, maskable interrupt, debugging, task switching and operating mode switching.

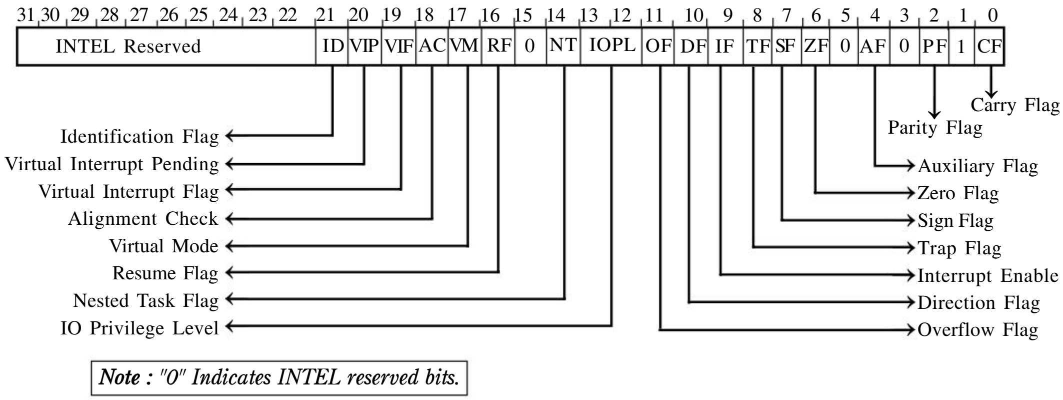


Fig. 8.15 : Flag register of 80386 microprocessor.

Control Registers

The 80386 has four numbers of 32-bit control registers CR0, CR1, CR2 and CR3. The control registers of 80386 are shown in Fig. 8.16. The lower 16-bit of CR0 is the **M**achine **S**tatus **W**ord (MSW) similar to MSW of 80286. The most significant bit of CR0 is used to enable/disable paging. The CR1 is reserved for future expansion. The CR2 holds the 32-bit linear address of the last page accessed before a page fault interrupt. The CR3 holds the 32-bit base address of the page directory. Since the page directory table is always page aligned (4 kb - aligned), the lower 12 bits of CR3 are undefined.

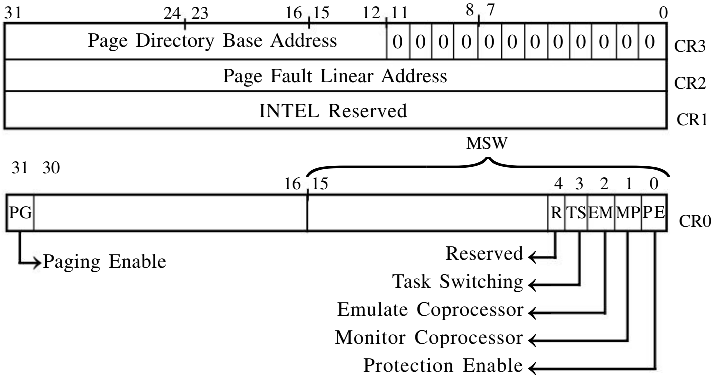


Fig. 8.16 : Control registers of 80386 microprocessor.

System Address and Segment Registers

The 80386 has two numbers of 32-bit system address registers and two numbers of 16-bit system segment registers. The system address registers are **G**lobal **D**escriptor **T**able **R**egister (GDTR) and **I**nterrupt **D**escriptor **T**able **R**egister (IDTR). The GDTR holds the 32-bit base address and 16-bit limit of GDT (**G**lobal **D**escriptor **T**able). The IDTR holds the 32-bit base address and 16-bit limit of IDT (**I**nterrupt **D**escriptor **T**able).

The system segment registers are **L**ocal **D**escriptor **T**able **R**egister (LDTR) and **T**ask **R**egister (TR). The LDTR holds the 16-bit selector for LDT (**L**ocal **D**escriptor **T**able) descriptor. The TR holds the 16-bit selector for TSS (**T**ask **S**tate **S**egment) descriptor. Each system segment register has a program invisible 64-bit descriptor register and they are used to hold an 8-byte descriptor for LDT and TSS.

Debug and Test Register

The 80386 has eight numbers of 32-bit debug registers in which six are accessible by programmer and two are reserved by INTEL. The debug registers are denoted as DR_0 - DR_7 as shown in Fig. 8.17. The DR_0 - DR_3 are used to specify four linear breakpoint address. The DR_4 and DR_5 are reserved by INTEL. The DR_7 is used to set the breakpoints and DR_6 displays the current status of the breakpoints.

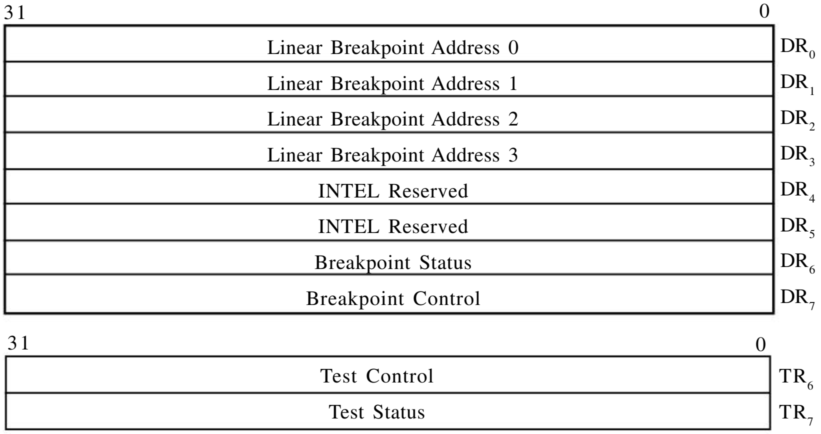


Fig. 8.17 : Debug and test registers of 80386 microprocessor.

The 80386 has two numbers of 32-bit test registers denoted as TR_6 and TR_7. The test registers will test the **T**ranslation **L**ookaside **B**uffer (TLB) in the paging unit. The TLB holds the most commonly used page table address translations, which are tested by test registers. The TR_6 holds the tag field (linear address) of the TLB and TR_7 holds the physical address of the TLB.

8.4.4 Operating Modes of 80386 Microprocessor

The 80386 has three operating modes and they are Real address mode, **P**rotected **V**irtual **A**ddress **M**ode (PVAM) and **V**irtual 80**86** (V86) mode. After a hardware reset, the processor will start working in the real address mode and this mode appears to programmers as a fast 8086 processor with a few additional new instructions. The main purpose of real address mode is to initialize the system for protected virtual address mode of operation. In PVAM, the 80386 works as a normal 32-bit processor, and all the instructions and features are available in this mode. The V86 mode allows the programmer to execute 8086 applications without disturbing the protection mechanism of PVAM. The 80386 processor can switch from PVAM to V86 mode to execute 8086 applications and then can return to PVAM.

Real Address Mode

The 80386 processor will enter the real address mode when it is reset. This mode is similar to 8086 but allows access to the 32-bit registers. In real address mode, the default operand size is 16 bits and so override prefixes should be employed to use 32-bit registers and addressing modes.

In real address mode, paging is disabled and the processor can access 1 Mb of physical memory space like 8086 using a 20-bit address. Therefore, the address lines A_2-A_{19} alone are active. The 20-bit physical address is computed by multiplying the segment register by 16_{10} and adding to an offset.

> ***Note*** *: Multiplying by $16_{10}$is equivalent to shifting four times left.*

In real address mode, the size of memory segments are 64 kb and the segments can be overlapped like 8086. Since segment size is 64 kb, when 32-bit effective address or offset is employed it should be less than 0000 $FFFF_H$. In real address mode, two memory areas are reserved. One for system initialization and the other for interrupt pointer table. The memory address 00000_H to $003FF_H$ are reserved for interrupt pointer table and the memory address $FFFF0_H$ to $FFFFF_H$ are reserved for system initialization.

All the instructions of 80386, except a few can be executed in real address mode. The primary purpose of real address mode is to set up the 80386 processor for protected virtual address mode of operation.

Protected Virtual Address Mode

The processor can switch from the real address mode to PVAM by setting the PE bit (**P**rotection **E**nable **bit**) in the control register CR_0. In PVAM, the processor can run all the 8086 and 80286 programs with sophisticated memory management and hardware assisted protection mechanism. In PVAM, the processor has a very large address space. It has 4 Gb of physical memory address space and 64 Tb of virtual memory address space. The address lines A_2 - A_{31} along with four bank select signals BE_0 - BE_3 are used to address 4 Gb of physical memory.

In PVAM, each physical address is represented by a 48-bit virtual address. The 80386 supports two method of converting a virtual address to physical address. In one method, the paging is disabled and the physical address is computed using the selector, descriptor and offset like 80286. In another method, the paging is enabled and a linear address is computed using the selector, descriptor and offset like 80286, and then the linear address is converted to physical address by the paging unit.

The physical address computation in 80386, when paging unit is disabled is shown in Fig. 8.18. The virtual address (pointer) can be 48-bit or 32-bit. The 48-bit pointer has a 16-bit selector and 32-bit offset. The 32-bit pointer has 16-bit selector and a 16-bit offset. The selector is used to fetch an 8-byte segment descriptor from the descriptor table. The descriptor has a 32-bit segment base address, segment length /limit, protection level, privilege level, the default operand size and segment type. The sum of segment base address and offset gives the 32-bit physical address. The 80386 allows a maximum size of 4 Gb to each segment.

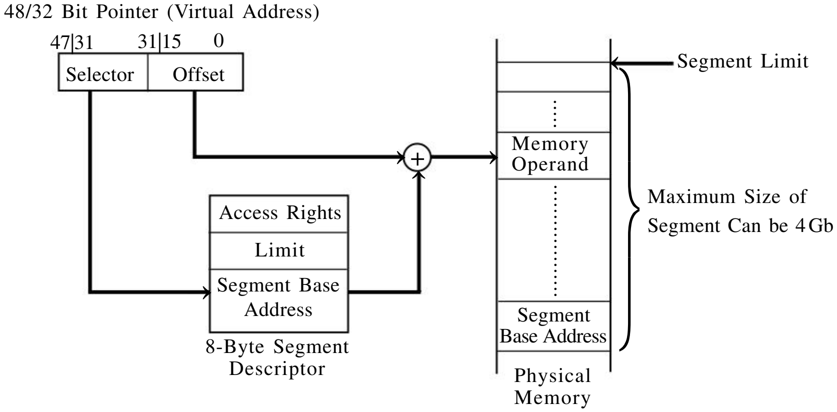


Fig. 8.18 : Address computation in PVAM of 80386 when paging is disabled.

The physical address computation in 80386 when paging is enabled is shown in Fig. 8.19. When paging is enabled the 80386 processor provides additional memory management mechanism which is available only in PVAM. The paging provides a method for managing very large memory segments defined in PVAM. The paging divides each segment into equal sized pages. When paging is enabled, the segmentation unit will generate a 32-bit linear address and supply it to the paging unit, which translates this linear address to a 32-bit physical address.

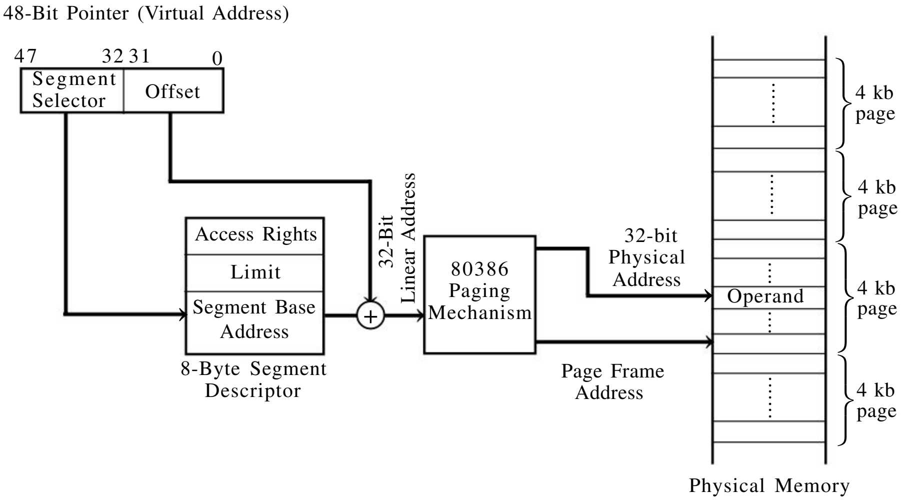


Fig. 8.19 : Address computation in PVAM of 80386 when paging is enabled.

When paging in enabled the segmentation unit computes a 32-bit linear address using a selector, descriptor and offset. (This computation is similar to the computation of physical address when paging is disabled.) In fact, the output of segmentation is used as a physical address when paging is disabled and when paging is enabled the output of segmentation unit is fed as input to the paging unit.

In paging mechanism, the memory segments can be organized as pages of size 4 kb. The 80386 paging mechanism allows 2^{20} pages ($1024 \times 1024 = 2^{10} \times 2^{10}$) of size 4 kb. The paging mechanism involves three elements and they are page directory, page table and page. The paging mechanism allows one page directory of size 4 kb and the page directory can define 1024 (1k) page tables. Four bytes (32 bits) of the page directory are used to store information about a page table. Each page table can define 1024 pages of size 4 kb. The size of each page table is also 4 kb and four bytes (32 bits) of page table are used to store information about a page.

The paging mechanism of 80386 is shown in Fig. 8.20. The segmentation unit will supply a 32-bit linear address to paging unit. The upper 10 bits (A_{22}-A_{31}) of linear address is the index for page directory, the middle 10 bits (A_{12}-A_{21}) of linear address is the index for the page table and the lower 12 bits (A_0-A_{11}) of linear address is the lower 12 bits of physical address.

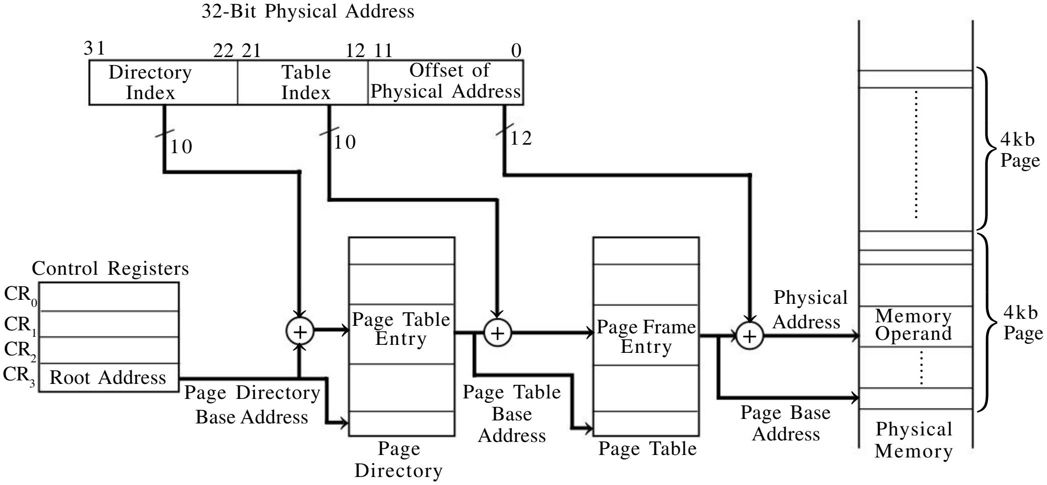


Fig. 8.20 : Paging mechanism of 80386 microprocessor.

The control register CR_3 holds the base/root address of the page directory. The root address is added to the 10-bit page directory index (given by the linear address) to get the address page directory entry of the page table. The page directory entry has the base address of the page table, which is added to the 10-bit page table index (given by the linear address) to get the address of the page table entry of the page. The page table entry has the upper 20 bits of the page frame address, which is concatenated with lower 12 bits of the linear address to form the physical address.

Virtual 8086 Mode

The processor can switch from PVAM to virtual 8086 mode by setting the VM bit in the EFLAG register to logic 1. While working in privilege level 0, the processor can also enter the virtual 8086 mode by executing the IRET instruction. The processor can return to PVAM from the virtual 8086 mode only on receipt of an interrupt or exception.

The virtual 8086 mode permits the execution of 8086 applications with all protection features of 80386. In the virtual 8086 mode, the segment registers are used similar to that of the real mode. In this mode, the processor computes 20-bit address by shifting the segment register left by four times and adding to the offset. The 20-bit address can be used to access 1 Mb of physical memory space.

In virtual 8086 mode, paging can be enabled to run multiple virtual mode tasks, to provide protection and operating system isolation. When paging is enabled, the 20-bit linear address can be divided into 256 pages and each page can be located anywhere in the 4 kb physical address space of 80386.

8.5 INTEL 80486 MICROPROCESSOR

The INTEL 80486 is a 32-bit processor with higher performance than 80386. It is integration of the improved 80386 processor, 80387 coprocessor and 8 kb RAM memory (called cache memory) on a single chip. The INTEL 80486 family of processors includes 80486SX, 80486DX, SX2, DX2, Write-back enhanced DX2, DX4 and Write-back enhanced DX4 processors. The base architecture for the entire family of 80486 processor is same except minor differences. The 80486SX and SX2 processors does not have an internal coprocessor unit. The DX2 and DX4 are double clock version of 80486. Also the DX4 processor has 16 kb internal cache memory. The concepts discussed in this section refer to 80486DX processor.

The 80486 has 1.2 million transistors and works three times faster than the combined operation of 80386 and 80387. The 80486 has five stages of instruction pipeline execution and allows simultaneous execution of two consecutive instructions if resources used by one instruction are not used by the other instruction. Due to extensive pipelining the execution time of most of the instruction is one clock cycle and average execution time of an instruction is 1.6 clock cycle.

The base architecture, memory address capability, memory management unit and operating modes are identical to that of 80386. The 80486 processor is software compatible with 80386. The instruction set of 80486 includes the instructions of 80386 and a few new instructions to support the new applications and increase performance.

The 80486DX is available with maximum clock speed ratings of 33, 66 and 100 MHz.

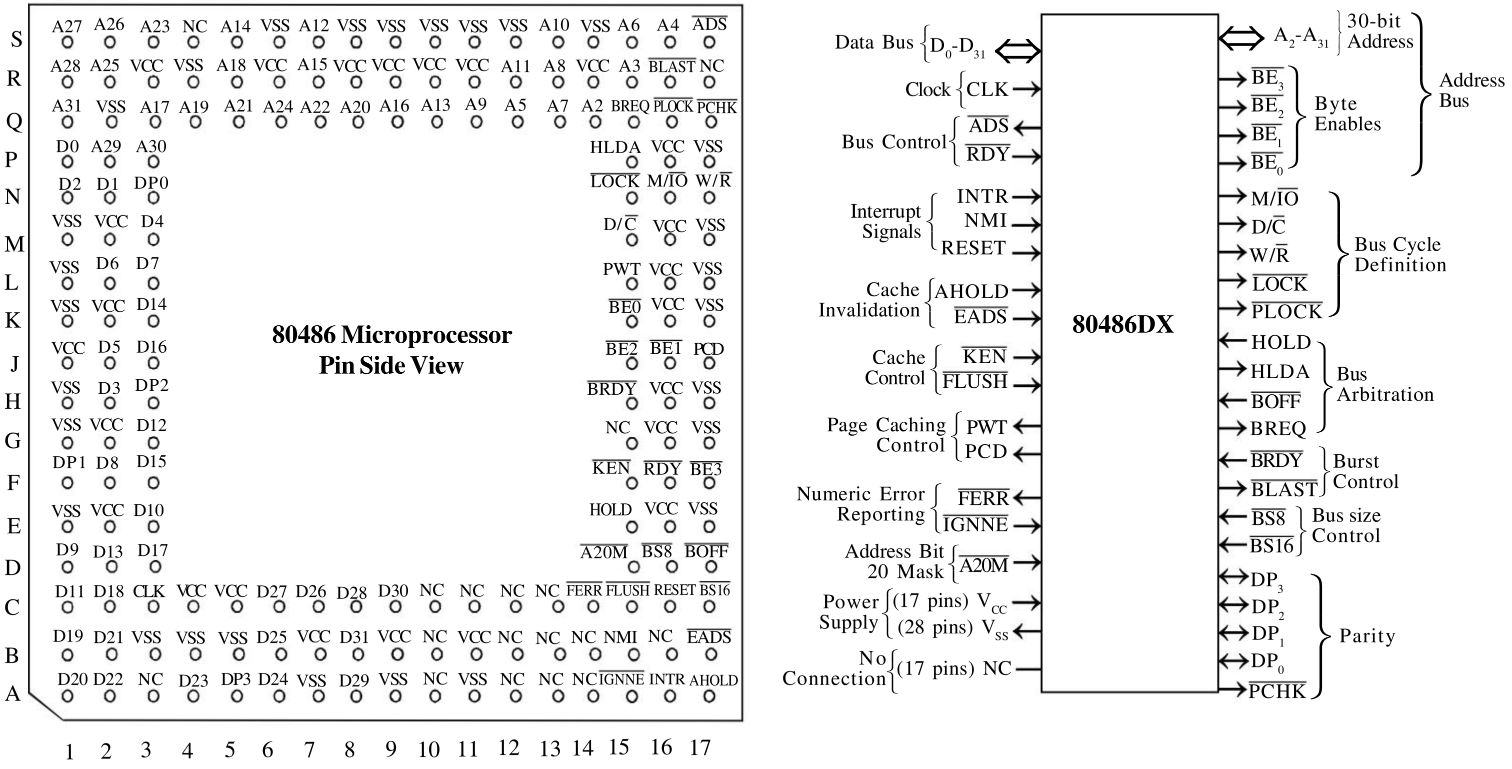
8.5.1 Pins and Signals of 80486

The 80486DX is a 168-pin IC and available in PGA package. The pin configuration of 80486DX is shown in Fig. 8.21. The functions of pins of 80486 are listed in Table-8.11.

The 80486 has 32 data pins and so we can form a 32-bit data bus. The 80486 has dynamic bus size feature, which allows 8-bit and 16-bit devices to be interfaced with the processor through a 32-bit data bus. When the signal $\overline{\text{BS8}}$ is asserted **low** the processor selects an 8-bit data bus and when $\overline{\text{BS16}}$ is asserted **low** the processor selects a 16-bit data bus.

The pins DP_0 to DP_3 are used for parity bits of the data bytes of the data bus. One pin is used for each byte of data bus. During write operation, the processor generates even parity bits and output on these lines. During read operation, the external device has to supply even parity bits through these lines. When parity is not employed these pins should be tied to V_{CC} through a pull up resistor. During read operation, whenever the processor detects a parity error, it asserts $\overline{\text{PCHK}}$ signal as **low**.

In 80486, the memory is organized as 4 banks, each of size 1 Gb. The address lines A_2-A_{31} are used to select the memory locations and the bank/byte enable signals $\overline{\text{BE}}_0$ - $\overline{\text{BE}}_3$ are used to select memory banks. The bank select signals are generated internally by decoding the address lines A_0 and A_1.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
S	A27	A26	A23	NC	A14	VSS	A12	VSS	VSS	VSS	VSS	VSS	A10	VSS	A6	A4	$\overline{ADS}$
R	A28	A25	VCC	VSS	A18	VCC	A15	VCC	VCC	VCC	VCC	A11	A8	VCC	A3	$\overline{BLAST}$	NC
Q	A31	VSS	A17	A19	A21	A24	A22	A20	A16	A13	A9	A5	A7	A2	BREQ	$\overline{PLOCK}$	$\overline{PCHK}$
P	D0	A29	A30												HLDA	VCC	VSS
N	D2	D1	DP0												$\overline{LOCK}$	M/$\overline{IO}$	W/$\overline{R}$
M	VSS	VCC	D4												D/$\overline{C}$	VCC	VSS
L	VSS	D6	D7												PWT	VCC	VSS
K	VSS	VCC	D14												$\overline{BE0}$	VCC	VSS
J	VCC	D5	D16												$\overline{BE2}$	$\overline{BE1}$	PCD
H	VSS	D3	DP2												$\overline{BRDY}$	VCC	VSS
G	VSS	VCC	D12												NC	VCC	VSS
F	DP1	D8	D15												$\overline{KEN}$	$\overline{RDY}$	$\overline{BE3}$
E	VSS	VCC	D10												HOLD	VCC	VSS
D	D9	D13	D17												$\overline{A20M}$	$\overline{BS8}$	$\overline{BOFF}$
C	D11	D18	CLK	VCC	VCC	D27	D26	D28	D30	NC	NC	NC	NC	$\overline{FERR}$	$\overline{FLUSH}$	RESET	$\overline{BS16}$
B	D19	D21	VSS	VSS	VSS	D25	VCC	D31	VCC	NC	VCC	NC	NC	NC	NMI	NC	$\overline{EADS}$
A	D20	D22	NC	D23	DP3	D24	VSS	D29	VSS	NC	VSS	NC	NC	NC	$\overline{IGNNE}$	INTR	AHOLD

Fig. 8.21 : Pin configuration of an 80486DX.

TABLE - 8.11 : SIGNALS OF 80486DX MICROPROCESSOR

Signal	Function	Type
D_{31}-D_0	32-bit data bus	Bidirectional
A_{31}-A_2	30-bit address bus. Used to address 1Gb memory space	Bidirectional
BE_3-BE_0	Byte/Bank Enables. Used to enable four memory banks each of size 1Gb	Output
CLK	Clock input	Input
$W/\overline{R}$	Write/Read control	Output
$D/\overline{C}$	Data/Control bus cycle indicator	Output
$M/\overline{IO}$	Memory or IO bus cycle indicator	Output
$\overline{LOCK}$	Bus priority lock control	Output
$\overline{PLOCK}$	Pseudo-lock output	Output
$\overline{ADS}$	Address Status	Output
$\overline{RDY}$	Nonburst Ready Input - (wait state request)	Input
$\overline{BRDY}$	Burst Ready Input	Input
$\overline{BLAST}$	Burst Last Output	Output
$\overline{BS8}$	Bus size 8. Used to select 8/32 bit data bus	Input
$\overline{BS16}$	Bus size 16. Used to select 16/32 bit data bus	Input
$\overline{A20M}$	Address bit 20 mask	Input
INTR	Maskable interrupt request	Input
NMI	Nonmaskable interrupt request	Input
RESET	Processor reset input	Input
HOLD	DMA hold request	Input
HLDA	Hold Acknowledge	Output
$\overline{BOFF}$	Backoff Input	Input
BREQ	Bus Request	Output
DP_3-DP_0	Data Parity IO	Bidirectional
$\overline{PCHK}$	Parity error status (Parity Check)	Output
AHOLD	Address hold	Input
$\overline{EADS}$	External address strobe	Input
$\overline{KEN}$	Cache memory enable	Input
$\overline{FLUSH}$	Cache memory flush (clear) control	Input
PWT	Page write through status	Output
PCD	Page cache disable status	Output
$\overline{FERR}$	Floating point error status	Output
$\overline{IGNNE}$	Ignore numeric error control	Input
V_{CC}	Power supply (+5-V)	Input
V_{SS}	Power supply ground (0-V)	Output
NC	No connection	–

The signal $\overline{\text{A20M}}$ input can be used to mask the physical address bit 20 while working in the real address mode. When $\overline{\text{A20M}}$ is asserted **low,** the processor address space is wraparound into 1 Mb memory space (from FFFFF_H to 00000_H) as that of 8086 processor.

The processor asserts $\overline{\text{ADS}}$ as **low** in the first clock of a bus cycle to indicate the start of a bus cycle and it is inactive in the subsequent clock of a bus cycle. The signal $\overline{\text{RDY}}$ is used to introduce wait states in nonburst bus cycles and the signal $\overline{\text{BRDY}}$ is used to introduce wait states in burst bus cycles. For normal bus timings, these signals should be tied **low**. The $\overline{\text{BLAST}}$ signal is asserted **low** by the processor to indicate the completion of a burst bus cycle.

The 80486 processor does not have an internal clock generator. The clock required for 80486 should be generated externally and supplied through CLK pin. The 80486 processor has active **high** reset. The RESET input is used to bring the processor to a known state. The reset will clear all segment registers except the CS-register and after a reset all general purpose registers except EDX will be in an undefined state. The reset will load F000_H in CS-register, 0FFF0_H in EIP and a component identifier is loaded in DX-register.

The 80486 has two hardware interrupt pins INTR and NMI. The INTR is a maskable interrupt request. The INTR is active **high** and it is not provided with an internal pull-down resistor. The NMI is nonmaskable interrupt request and when asserted, it executes a type-2 interrupt. The NMI is rising edge sensitive and it is not provided with an internal pull-down resistor.

The signals $\text{M}/\overline{\text{IO}}$, $\text{D}/\overline{\text{C}}$ and $\text{W}/\overline{\text{R}}$ are primary bus cycle definition signals and these signals are asserted after a valid $\overline{\text{ADS}}$. For memory read/write cycle $\text{M}/\overline{\text{IO}}$ is asserted **high.** For IO read/write cycle, $\text{M}/\overline{\text{IO}}$ is asserted **low**. For memory or IO data bus cycles (read/write), the $\text{D}/\overline{\text{C}}$ is asserted **high**. For interrupt acknowledge, code access and halt bus cycles the $\text{D}/\overline{\text{C}}$ is asserted **low**. The $\text{W}/\overline{\text{R}}$ differentiates between read and write bus cycles.

The $\overline{\text{LOCK}}$ signal is activated during execution of instruction prefixed by lock prefix . The $\overline{\text{PLOCK}}$ is asserted **low** during memory read/write operations that involves operands greater than 32 bits. The $\overline{\text{PLOCK}}$ is asserted **low** during floating point unit read/write (which involve 64 bits operand), segment table descriptor read (which involve 64 bits operand) and cache line fills (which involve 128 bits operand). When $\overline{\text{LOCK}}$ or $\overline{\text{PLOCK}}$ is asserted **low**, the other bus masters cannot take control of the system bus.

The HOLD input forces the processor to **high impedance** state after execution of current bus cycle and asserts HLDA to acknowledge the device which requested the HOLD. The HOLD and HLDA are mainly used for DMA data transfer. The $\overline{\text{BOFF}}$ input will force the processor to **high impedance** state immediately in the next clock. The $\overline{\text{BOFF}}$ is similar to HOLD but differs in two ways : First, the processor will not complete the current bus cycle before releasing the bus. Second, the processor does not assert HLDA.

The processor asserts BREQ whenever a bus cycle is pending internally. External logic can use BREQ signal to arbitrate among multiple processors. The AHOLD and $\overline{\text{EADS}}$ inputs are used during cache invalidation cycles. When AHOLD is asserted, the processor will stop driving its address bus in the next clock and so an external device can supply an address to the processor through the address bus. The $\overline{\text{EADS}}$ input is asserted **low** to indicate that a valid external address has been driven into the address bus.

The $\overline{\text{KEN}}$ is a cache enable pin and is used to determine whether the data being returned by the current cycle is cacheable. The $\overline{\text{FLUSH}}$ input forces the processor to flush its entire internal cache and this signal needs to be asserted **low** for one clock period to clear the cache memory.

The PWT and PCD output signals correspond to two user attribute bits in the page table entry. The PCD output reflects the state of the PCD attribute bit in the page table entry or the page directory entry. The PWT output reflects the state of the PWT attribute bit in the page table entry or page directory entry.

The processor asserts $\overline{\text{FERR}}$ pin as **low** whenever an unmasked floating point error is encountered. The assertion of $\overline{\text{IGNNE}}$ input signal informs the processor to ignore the floating point errors and continue executing noncontrol floating-point instructions.

8.5.2 Architecture of 80486

The INTEL 80486 is a 32-bit processor with on-chip memory management, floating point and cache memory units. The architecture (or functional block diagram) of 80486 processor is shown in Fig. 8.22. The various functional units of 80486 processor are:

- Data processing unit consisting of ALU, barrel shifter and an array of registers.
- Bus interface unit consisting of drivers and various control logic unit.
- 32-byte instruction prefetch queue.
- Instruction decode unit.
- Floating point unit.
- Memory management unit consisting of segmentation and paging units.
- 8 kb cache memory unit.

The data processing unit consists of a 32-bit ALU, a 64-bit barrel shifter and eight 32-bit general purpose registers. The functions of ALU and barrel shifter are same as that of 80386 processor. The general purpose registers of 80486 are also same as that of 80386 processor. The general purpose registers are EAX, EBX, ECX, EDX, EBP, EDI, ESI and ESP. Part of these 32-bit registers can be accessed as 16 or 8-bit registers like that of 80386. The Instruction Pointer and Flag register of 80486 are identical to that of 80386 processor.

The bus interface unit consists of drivers for address and data bus and various control logic units, which includes bus cycle control, burst bus cycle control, bus size control and cache control logic units. The control signals necessary for memory, IO and interrupt bus cycles are generated by this unit. It also takes care of managing the control signals for cache memory control.

The bus interface unit also has a parity generation and control unit. The parity unit generates a parity bit (for even parity) for each byte of data during write operation, and output on DP_0 - DP_3 lines. These parity bits can be stored in the memory along with data when parity is employed in the system. During read operation the parity unit checks for even parity and if it finds an error then it generates a parity check error signal.

The 80486 processor has five stage instruction pipeline execution, which includes prefetch, first decode, second decode, execute and write back. Due to five stage pipeline, several instructions will be in the pipeline at a time. Hence, the 80486 processor can execute two instructions

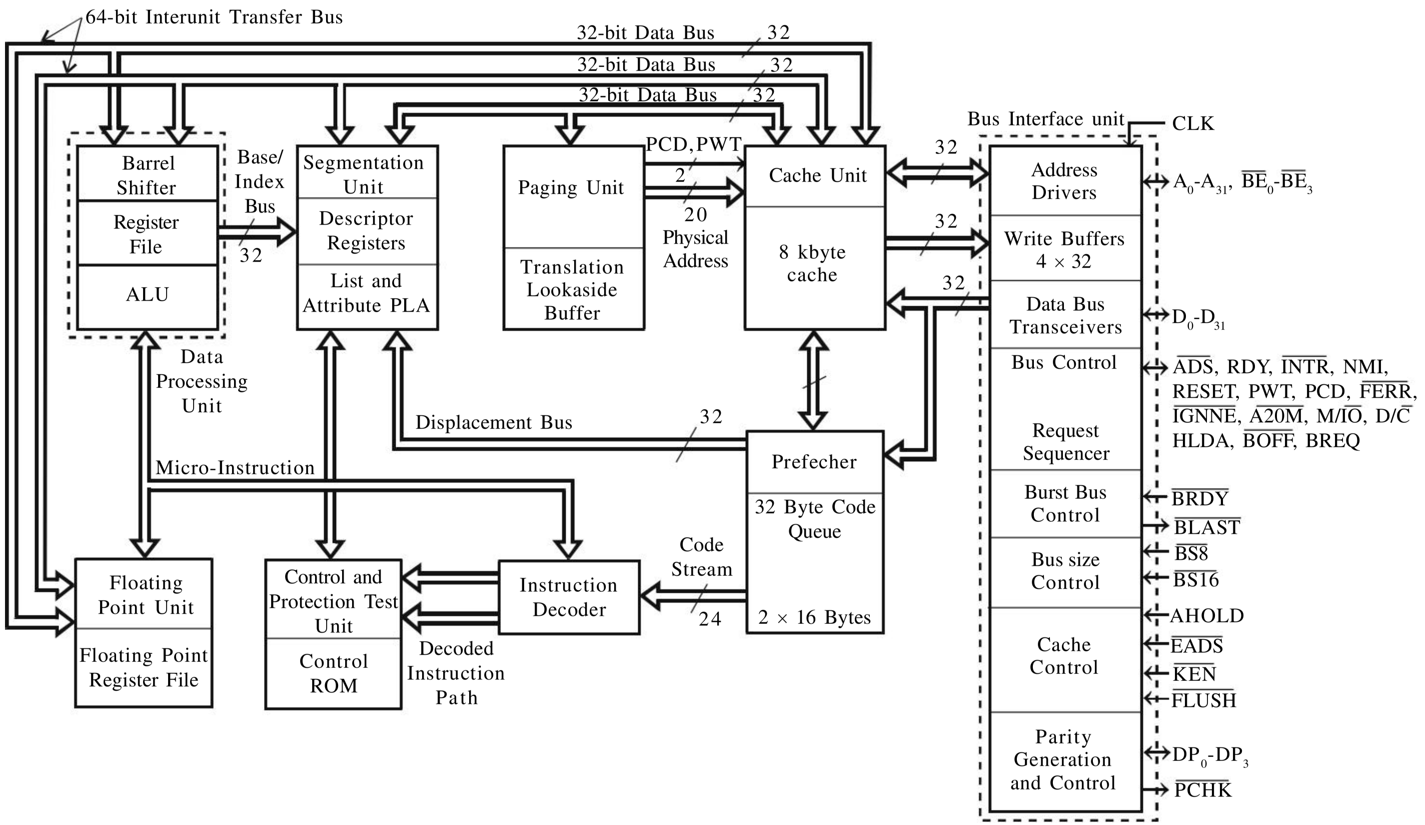


Fig. 8.22 : Architecture of 80486 microprocessor.

simultaneously if execution of one instruction does not depend on the other instruction. While decoding jump instructions, the processor automatically prefetch the instructions from the jump destination, which improves the processor performance greatly.

The floating point unit consists of eight numbers of 80-bit data registers, three numbers of 16-bit registers called status register, control register and tag word register and two numbers of 48-bit pointer registers called instruction pointer and data pointer. The floating point unit supports 32/64/80 bits floating point data types, 16/32/64 bits (signed) integer data types and 80-bit packed BCD data types.

The **M**emory **M**anagement **U**nit (MMU) consists of a segmentation unit and paging unit. The MMU of 80486 is almost identical to that of 80386. Segmentation allows management of the logical address space. The paging mechanism operates beneath segmentation. The paging is optional and can be disabled by system software. Memory is organized into one or more variable length segments, each up to 4 Gb in size. Each segment can be divided into one or more 4 kb pages. The segment registers and descriptors of 80486 are identical to that of 80386.

The cache memory contains static RAMs which are very fast as compared to the dynamic RAMs. The cache memory address is a small part of total memory space which the processor can address. The cache memory can be used to store both code and data. The cache memory is organised as a four-way set associative cache with LRU (**L**east **R**ecently **U**sed) replacement technique. The 8 kb cache is divided into 128 sets. Each set has 64 bytes (8 kb = 2^{13} bytes = $2^7 \times 2^6$ bytes = 128×64 bytes) and organized as 4 lines with 16 bytes per line (4×16 bytes = 64 bytes). Thus, cache organization is a 4-way set-associative cache.

8.6 PENTIUM MICROPROCESSOR

The Pentium processor is an advanced 32-bit superscalar processor with 64-bit data bus and 32-bit address bus to address up to 4 Gb of physical memory space. It was released in the year **1993** and consists of **3.1** million transistors. The Pentium employs two general purpose integer pipelines, branch prediction, highly pipelined floating point unit and separate code and data caches to achieve the highest performance level while preserving the binary code compatibility with 80x86 processors. The Pentium processor can execute two integer instructions simultaneously. The Pentium is available with maximum clock speed ratings of 60 to 233 MHz.

The features of Pentium processor are:

- Superscalar architecture
- Pipelined Floating-Point Unit
- Separate code and data caches
- Bus cycle pipelining
- Internal parity checking
- Execution tracing
- IEEE 1149.1 boundary scan
- Virtual mode extensions
- Advanced power management feature
- On-chip local APIC (**A**dvanced **P**rogrammable **I**nterrupt **C**ontroller) device.
- Dynamic branch prediction
- Improved instruction execution time
- 64-bit data bus
- Address parity
- Functional redundancy checking and lock-step operation
- Performance monitoring
- System Management Mode
- Dual processing support
- Fractional bus operation

8.6.1 Pins and Signals of Pentium Microprocessor

The Pentium processor is a 296-pin IC available in SPGA package. The pin configuration of Pentium processor is shown in Fig. 8.23 and functional grouping of pins of Pentium processor is shown in Fig. 8.24. The functions of pins of the Pentium processor are listed in Table-8.12.

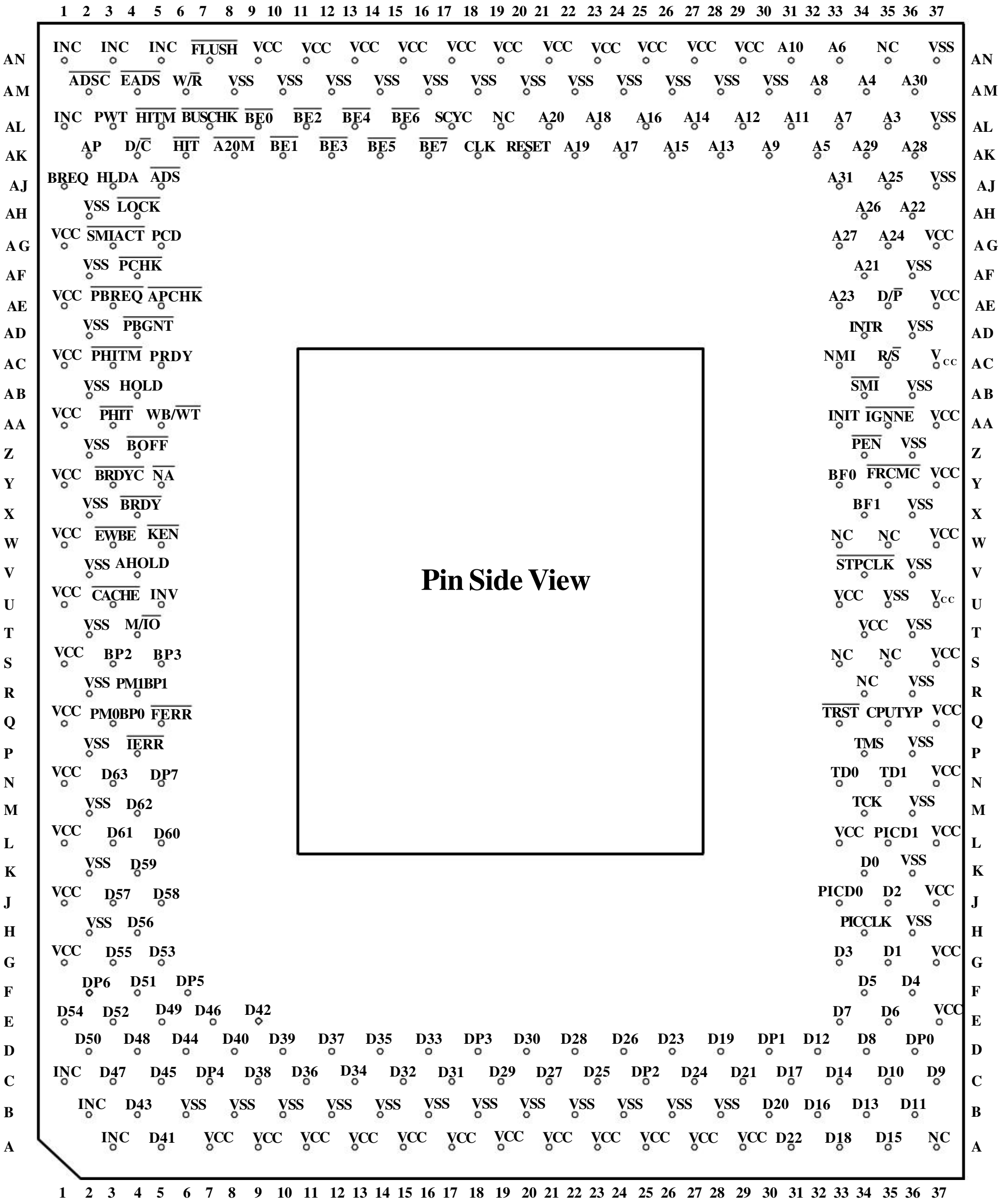

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37
AN INC INC INC FLUSH VCC VCC VCC VCC VCC VCC VCC VCC VCC VCC VCC A10 A6 NC VSS AN
AM ADSC EADS W/R VSS VSS VSS VSS VSS VSS VSS VSS VSS VSS VSS VSS A8 A4 A30 AM
AL INC PWT HITM BUSCHK BE0 BE2 BE4 BE6 SCYC NC A20 A18 A16 A14 A12 A11 A7 A3 VSS AL
AK AP D/C HIT A20M BE1 BE3 BE5 BE7 CLK RESET A19 A17 A15 A13 A9 A5 A29 A28 AK
AJ BREQ HLDA ADS A31 A25 VSS AJ
AH VSS LOCK A26 A22 AH
AG VCC SMIACT PCD A27 A24 VCC AG
AF VSS PCHK A21 VSS AF
AE VCC PBREQ APCHK A23 D/P VCC AE
AD VSS PBGNT INTR VSS AD
AC VCC PHITM PRDY NMI R/S Vcc AC
AB VSS HOLD SMI VSS AB
AA VCC PHIT WB/WT INIT IGNNE VCC AA
Z VSS BOFF PEN VSS Z
Y VCC BRDYC NA BF0 FRCMC VCC Y
X VSS BRDY BF1 VSS X
W VCC EWBE KEN NC NC VCC W
V VSS AHOLD STPCLK VSS V
U VCC CACHE INV VCC VSS Vcc U
T VSS M/IO VCC VSS T
S VCC BP2 BP3 NC NC VCC S
R VSS PM1BP1 NC VSS R
Q VCC PM0BP0 FERR TRST CPUTYP VCC Q
P VSS IERR TMS VSS P
N VCC D63 DP7 TD0 TD1 VCC N
M VSS D62 TCK VSS M
L VCC D61 D60 VCC PICD1 VCC L
K VSS D59 D0 VSS K
J VCC D57 D58 PICD0 D2 VCC J
H VSS D56 PICCLK VSS H
G VCC D55 D53 D3 D1 VCC G
F DP6 D51 DP5 D5 D4 F
E D54 D52 D49 D46 D42 D7 D6 VCC E
D D50 D48 D44 D40 D39 D37 D35 D33 DP3 D30 D28 D26 D23 D19 DP1 D12 D8 DP0 D
C INC D47 D45 DP4 D38 D36 D34 D32 D31 D29 D27 D25 DP2 D24 D21 D17 D14 D10 D9 C
B INC D43 VSS VSS VSS VSS VSS VSS VSS VSS VSS VSS VSS VSS D20 D16 D13 D11 B
A INC D41 VCC VCC VCC VCC VCC VCC VCC VCC VCC VCC VCC VCC D22 D18 D15 NC A
Pin Side View
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37

Fig. 8.23 : Pin configuration of Pentium processor.

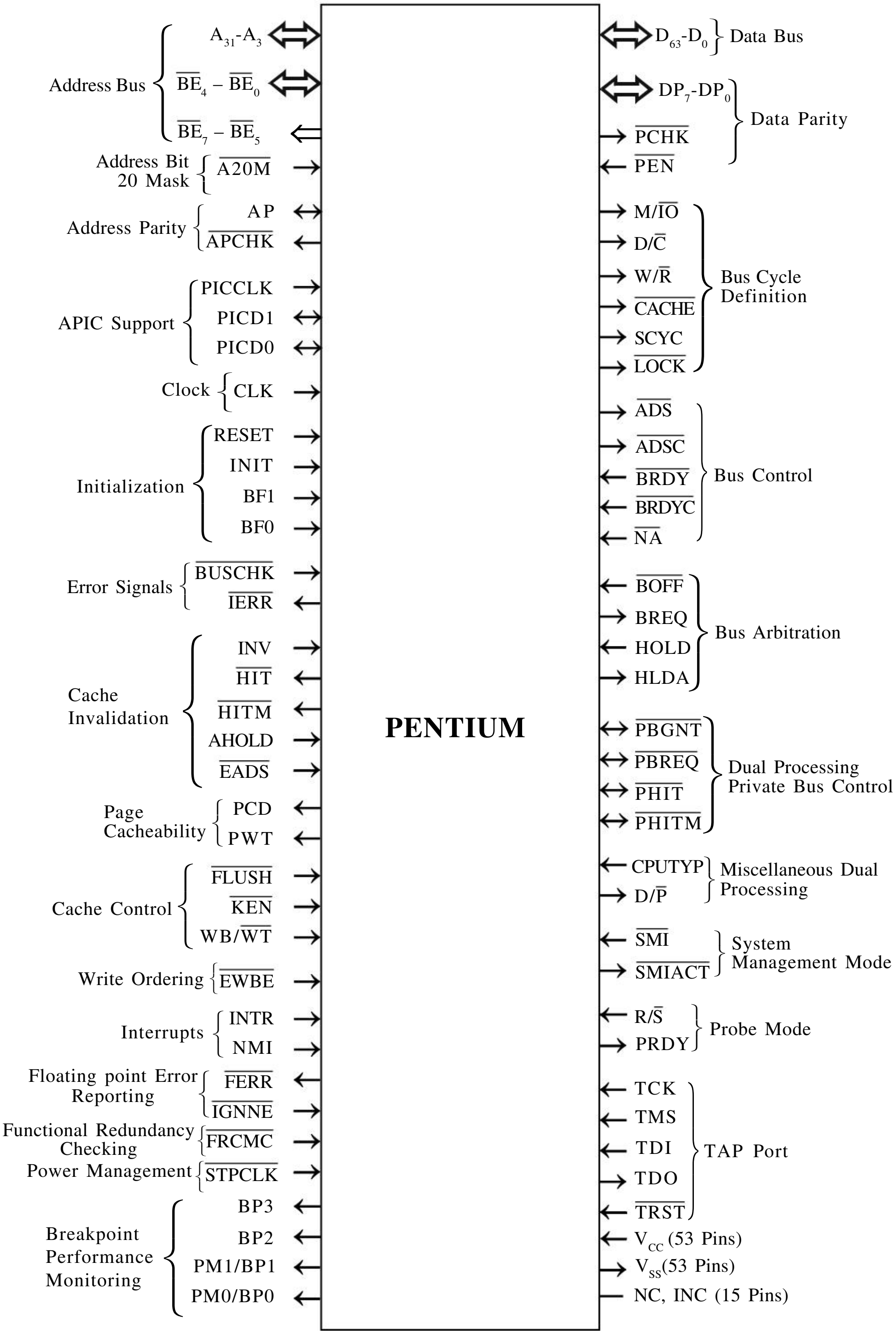


Fig. 8.24 : Functional grouping of the pins of a Pentium processor.

The Pentium has 64 data pins and so we can form a 64-bit data bus. The physical memory in the Pentium processor-based system can be organized as 8 banks. Each bank can have an address space of 512 Mb. The Pentium processor has 29 address pins (A_3-A_{31}) to address 512 Mb of address space and 8-bank select signals ($\overline{BE}_0$ to $\overline{BE}_7$) to select 8-memory banks.

The pins DP_0 - DP_7 are used for parity of the data bytes of the data bus and the functions of these pins are similar to that of data parity pins of 80486. The pins W/$\overline{R}$, D/$\overline{C}$, M/$\overline{IO}$, $\overline{LOCK}$, $\overline{BRDY}$, $\overline{A20M}$, INTR, NMI, HOLD, HLDA, $\overline{BOFF}$, BREQ, $\overline{PCHK}$, AHOLD, $\overline{EADS}$, $\overline{KEN}$, $\overline{FLUSH}$, PWT, PCD, $\overline{FERR}$ and $\overline{IGNNE}$ are common to 80486 and Pentium processors. The functions of these pins in Pentium processor are similar to that in 80486 processor.

The pin CPUTYP is used to distinguish the primary processor and dual processor. For primary processor CPUTYP is tied to ground (V_{SS}). For dual processor it is tied to V_{CC}. A brief description about the functions of each pin of Pentium processor are provided in Table-8.12. For detailed explanations the readers are advised to refer INTEL Pentium data sheet.

TABLE - 8.12 : SIGNALS OF PENTIUM MICROPROCESSOR

Signal	Function	Type
$\overline{A20M}$	Address 20 mask. Used to emulate the 1Mb address wrap around of 8086.	Input
A_{31}-A_3	29-bit address bus. Used to address 512 Mb memory space.	Bidirectional
$\overline{ADS}$	Address strobe. Indicates that a new valid bus cycle is currently being driven by the processor.	Output
$\overline{ADSC}$	Additional address strobe.	Output
AHOLD	Address hold. Used to get the address bus for an inquire cycle.	Input
AP	Address parity. Address parity pin for the address lines.	Bidirectional
$\overline{APCHK}$	Address parity check. The status of the address parity check is driven on this output.	Output
$\overline{BE}_7$ - $\overline{BE}_0$	Byte /Bank enables. Used to enable eight memory banks each of size 512 Mb.	Output
BF_1-BF_0	Bus-to-core frequency ratio. Used to configure processor bus-to-core frequency ratio.	Input
$\overline{BOFF}$	Backoff input. This input is used to force the processor off the bus in the next clock.	Input
BP_3-BP_0	Breakpoint signals. These signals externally indicates a breakpoint match.	Output
$\overline{BRDY}$	Burst ready. Transfer complete indication.	Input

Table - 8.12 continued...

Signal	Function	Type
$\overline{\text{BRDYC}}$	Additional burst ready input	Input
BREQ	Bus request. Indicates externally when a bus cycle is pending internally.	Output
$\overline{\text{BUSCHK}}$	Bus check. Allows the system to signal an unsuccessful completion of a bus cycle.	Input
$\overline{\text{CACHE}}$	Cacheability. External indication of internal cacheability.	Output
CLK	Clock. Fundamental timing source for processor.	Input
CPUTYP	Processor type definition pin. Used to configure as primary/dual processor.	Input
D/$\overline{\text{C}}$	Data/Code access. Distinguishes a data access from a code access.	Output
D_{63}-D_0	64-data lines. Forms the 64-bit data bus.	Bidirectional
DP_7-DP_0	Data parity. These are parity pins for the data bytes.	Bidirectional
D/$\overline{\text{P}}$	Dual/Primary processor indicator.	Output
$\overline{\text{EADS}}$	External address strobe. This input signals the processor to run an inquire cycle with the address on the bus.	Input
$\overline{\text{EWBE}}$	External write buffer empty. Provides the option of strong write ordering to the memory system.	Input
$\overline{\text{FERR}}$	Floating point error. The floating point error output is driven active when an unmasked floating point error occurs.	Output
$\overline{\text{FLUSH}}$	Cache flush. Writes all modified lines in the data cache back and flushes the code and data caches.	Input
$\overline{\text{FRCMC}}$	**F**unctional **R**edundancy **C**hecking **M**aster/**C**hecker configuration. Determines whether the processor is configured as a master or checker.	Input
$\overline{\text{HIT}}$	Inquire cycle hit/miss.	Output
$\overline{\text{HITM}}$	Inquire cycle hit/miss to a modified line.	Output
HLDA	Bus hold acknowledge. External indication that the processor outputs are floated.	Output
HOLD	Bus hold request. Used for DMA transfer.	Input

Table - 8.12 continued...

Signal	Function	Type
$\overline{\text{IERR}}$	Internal or functional redundancy check error. Alerts the system of internal parity errors and functional redundancy errors.	Output
$\overline{\text{IGNNE}}$	Ignore numeric exception. Determines whether or not numeric exceptions should be ignored.	Input
INIT	Initialization. Forces the processor to begin execution in a known state without flushing the caches or affecting the floating point state.	Input
INTR	Nonmaskable external interrupt request.	Input
INV	Invalidation request. Determines the final state of a cache line as a result of an inquire hit.	Input
$\overline{\text{KEN}}$	Cache enable. Indicates to the processor whether or not the system can support a cache line fill for the current cycle.	Input
$\overline{\text{LOCK}}$	Bus lock. Indicates to system that the current bus cycle should not be interrupted.	Output
M/$\overline{\text{IO}}$	Memory or IO indicator. Distinguishes a memory access from an IO access.	Output
$\overline{\text{NA}}$	Next address. Indicates that external memory is prepared for a pipelined cycle.	Input
NMI	Nonmaskable interrupt request.	Input
$\overline{\text{PBGNT}}$	Dual processor bus grant. Indicates to the LRM (**L**east **R**ecent **M**aster) processor that it will become the MRM (**M**ost **R**ecent **M**aster) processor in the next cycle.	Bidirectional
$\overline{\text{PBREQ}}$	Dual processor bus request. Indicates to MRM processor that LRM processor requires the bus.	Bidirectional
PCD	Page cacheability disable. Externally reflects the cacheability paging attribute bit in control register.	Output
$\overline{\text{PCHK}}$	Data parity check. Indicates the result of a parity check on a data read.	Output
$\overline{\text{PHIT}}$	Private inquire cycle hit/miss indication.	Bidirectional
$\overline{\text{PHITM}}$	Private inquire cycle hit/miss to a modified line indication.	Bidirectional
PICCLK	Processor interrupt controller clock. This pin drives the clock for APIC serial data bus operation.	Input
PICD1-PICD0	Processor interrupt controller data. These are the data pins for the 3-wire APIC bus.	Bidirectional

Table - 8.12 continued...

Signal	Function	Type
$\overline{\text{PEN}}$	Parity enable. This signal determines whether a machine check exception has to be taken for data parity error.	Input
PM1-PM0	**P**erformance **M**onitoring. Externally indicates the status of the performance monitor counter.	Output
PRDY	Probe ready. For use with INTEL debug port.	Output
PWT	Page write through status. Externally reflects the write through paging attribute bit in control register.	Output
R/$\overline{\text{S}}$	Run/Stop. For use with the INTEL debug port.	Input
RESET	Processor reset. Forces the processor to begin execution at a known state.	Input
SCYC	Split cycle indication. Indicates that a misaligned locked transfer is on the bus.	Output
$\overline{\text{SMI}}$	System management interrupt request.	Input
$\overline{\text{SMIACT}}$	System management interrupt active. Indicates that the processor is operating in SMM.	Output
$\overline{\text{STPCLK}}$	Stop clock request. Used to stop the internal processor clock and consume less power.	Input
TCK	Test clock input. Provides boundary scan clocking function.	Input
TDI	**T**est **D**ata **I**nput. Input pin to receive serial data and instructions.	Input
TDO	**T**est **D**ata **O**utput. Output serial test data and instructions.	Output
TMS	**T**est **M**ode **S**elect. Controls TAP controller state transitions.	Input
$\overline{\text{TRST}}$	Test reset. Allows the TAP controller to be asynchronously initialized.	Input
V_{CC}	Supply to processor (3.3-V).	Input
V_{SS}	Power supply ground.	Output
W/$\overline{\text{R}}$	**W**rite/**R**ead control. Distinguishes a write cycle from a read cycle.	Output
WB/$\overline{\text{WT}}$	**W**rite **B**ack/**W**rite **T**hrough. This pin allows a cache line to be defined as write back or write through on a line by line basis.	Input

8.6.2 Architecture of Pentium Processor

The Pentium processor has superscalar architecture which allows parallel execution of two instructions. The various functional blocks of Pentium processor are shown in Fig. 8.25. The functional units of Pentium processor are Bus unit, Paging unit, Prefetch buffer, Instruction decode, Control ROM, Execution unit with two integer pipelines (U-pipe and V-pipe), **B**ranch **T**arget **B**uffer (BTB), Code cache, Data cache, **F**loating **P**oint **U**nit (FPU), **D**ual **P**rocessing (DP) logic, and **A**dvanced **P**rogrammable **I**nterrupt **C**ontroller (APIC).

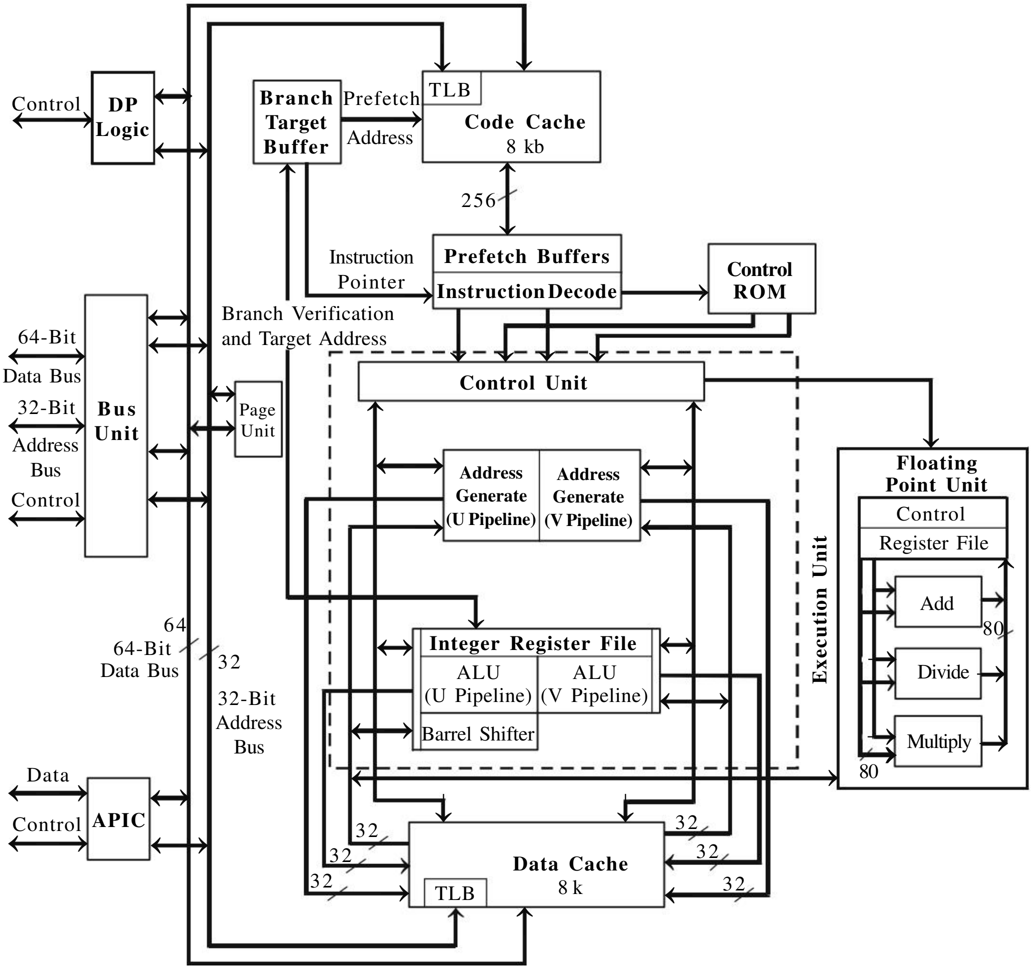


Fig. 8.25 : Architecture of a Pentium processor.

The bus unit takes care of the issuing control signals and fetching the code/data from the external memory and IO devices. The Pentium processor has 64-bit external data bus and supports burst read and burst writeback cycles. In addition, bus cycle pipelining has been added to allow two bus cycles to be in progress simultaneously. The paging unit contains optional extensions to the architecture which allow 2 Mb and 4 Mb page sizes.

The code cache, branch target buffer and prefetch buffers are responsible for getting instructions into the execution unit. Instructions are fetched from the code cache or from the external memory. Branch addresses are remembered by the branch target buffer. The code cache TLB (**T**ranslation **L**ookaside **B**uffer) translates linear addresses to physical addresses used by the code cache.

The Pentium processor has two independent pairs of 32-byte prefetch buffers which operate in conjunction with BTB (**B**ranch **T**arget **B**uffer). Only one prefetch buffer will be active at any one time. One prefetch buffer will fetch instructions sequentially until a branch instruction is encountered. When a branch instruction is fetched, the BTB will predict whether a branch will occur or not. If the BTB predicts that the branch will not take place then the prefetch buffer will continue fetching instructions linearly. If the BTB predicts that the branch will take place then the other prefetch buffer is enabled and begins to fetch instructions from branch target address. In this way the instructions needed for branching are also prefetched and kept ready for decode and execution.

If a branching is mispredicted then the instruction pipelines are flushed and the first buffer will start prefetch activity. The penalty for misprediction is three clock in U-pipe and four clock in V-pipe. Mispredicted calls and unconditional jump instructions have a three clock penalty in either pipe.

The execution unit has two parallel integer pipelines U-pipe and V-pipe with individual ALU for each pipe. The pipeline has five stages and they are **P**re**f**etch (PF), **D**ecode **S**tage-1(D1), **D**ecode Stage-**2** (D2), **E**xecute (E) and **W**rite**b**ack (WB). The U-pipe can execute all integer and floating point instructions. The V-pipe can execute simple integer instructions and the FXCH floating point instruction.

While executing (previous) instructions, the processor checks the next two instructions. If the execution of one instruction does not depend on the other then the first instruction is issued to U-pipe and the second instruction is issued to V-pipe, so that two instructions can be executed simultaneously. If it is not possible to execute two instructions simultaneously then the two instructions are issued to U-pipe one by one and no instruction is issued to V-pipe.

The control ROM contains microcodes to control the sequence of operations that must be performed to implement the Pentium processor architecture. The control ROM unit has direct control over both pipelines. The Pentium processor has an 8 kb code cache and 8 kb data cache. These caches are transparent to application software to maintain compatibility with previous INTEL processors. The data cache fully supports the MESI (**M**odified/**E**xclusive/**S**hared/**I**nvalid) cache consistency protocol. The code cache is write protected to prevent code corruption and so supports a subset of MESI protocol, the S (**S**hared) and I (**I**nvalid) states.

The cache is organized as a 2-way set associative cache. There are 128 sets in each cache with each set containing 2 lines. Each cache line is 32 bytes wide. The replacement in the cache is handled by LRU (**L**east **R**ecently **U**sed) mechanism.

The code cache is connected to prefetch buffer by a 256-bit bus. Hence, in one clock cycle the instruction cache can provide up to 32 bytes (32 ×8 = 256) of raw opcode to prefetch buffer. The data cache has two ports and each port is connected to pipelines by a 32-bit. Hence the data cache can provide data for two data references simultaneously. The caches are accessed with physical address and each cache has its own TLB (**T**ranslation **L**ookaside **B**uffer) to translate linear addresses to physical address.

The Pentium processor has a pipelined **F**loating **P**oint **U**nit (FPU) working independently. The FPU of Pentium processor is up to ten times faster than the FPU of 80486 processor for common operations including add, multiply and load.

The Pentium processor supports clock control. When the clock to the processor is stopped, power dissipation is virtually eliminated and this makes the Pentium processor a good choice for energy efficient designs. The Pentium processor supports fractional bus operation. This allows the processor core to operate at high frequencies, while communicating with the external bus at lower frequencies.

The Pentium processor contains an on-chip **A**dvanced **P**rogrammable **I**nterrupt **C**ontroller (APIC), which supports multiprocessor interrupt management, multiple IO subsystem support, 8259A compatiblity and inter-processor interrupt support.

The Pentium processor has in-built logic for dual processing mode of operation. In dual processor mode, two identical Pentium processors can be interfaced to a single system bus. The dual processor pair appears to the system bus as a single, unified processor. Multiprocessor operating systems properly schedule computing tasks between the two processors. This scheduling of task is transparent to software applications and the end user. Through a private bus, the two processors arbitrate for the external bus and maintains cache coherency. In dual processing mode, both the processor should have the same bus to core frequency ratio.

8.7 ADVANCED PENTIUM PROCESSORS

INTEL has released a number of advanced microprocessors after pentium with advanced features. (Please refer to Appendix-III for processors released by INTEL.) In the pentium series, the INTEL has released Pentium Pro, Pentium II, Pentium III and Pentium 4 processors. The Pentium Pro, Pentium II and Pentium III have a common architecture and the architecture of these processors are known as INTEL's P6 microarchitecture. The architecture of Pentium 4 is known as NetBurst Microarchitecture. The higher clock version of pentium 4 incorporates **H**yper **T**hreading (HT) technology. In all these advanced pentium processors, the basic data size and memory addressing capability has been retained, but the processors are provided with enhanced features and advanced/sophisticated method of instruction execution. The salient features of the advanced pentium processors are presented in the following sections:

8.7.1 Pentium PRO

The Pentium pro is a 32-bit processor with 64-bit data bus and 36-bit address bus to address up to 64 Gb of physical memory space. It was released in the year 1995 and consists of 5.5 million transistors. It is a 387 pin IC and available in PGA (**P**in **G**rid **A**rray) package. It is available with maximum internal clock ratings of 150/166/180/200 MHz.

The features of pentium pro processor are:

- Three-way superscalar architecture.
- Five parallel execution units and 12-stage super pipeline.
- Dual cavity PGA ceramic packages with a CPU die and a secondary cache die.
- Out of order execution and speculative execution.
- DIB (**D**ual **I**ndependent **B**us) architecture.
- Register renaming
- Error checking and correcting codes.
- Improved power management with two extra modes (Stop Grant and Auto HALT modes).
- Internal micro-ops similar to RISC like instructions.
- Transactional IO bus.
- Scalable up to four processors.
- Fault analysis/recovery.
- Integrated level two (secondary) cache of 256 k/512 k/1Mb.
- Internal thermal protection.
- Automatic selection of power supply voltage.

8.7.2 Pentium II

The Pentium II was released in 1997 and consists of 7.5 million transistors. It is actually a pentium pro processor with on-chip MMX (**M**ulti **M**edia E**x**tension). It is also a 32-bit processor with 64-bit data bus and 36-bit address bus to address up to 64 Gb of physical memory space. It is available with maximum internal clock ratings of 233 MHz to 450 MHz and in SEC (**S**ingle **E**dge **C**onnector) catridge packaging or as a boxed processor along with fan/heatsink.

The features of pentium II processor are:

- Supports the INTEL architecture with dynamic execution.
- Integrated primary (L1) 16 kb instruction cache and 16 kb write back data cache.
- Integrated 256 kb second level (L2) cache.
- Fully compatible with previous microprocessors.
- Supports MMX technology.
- Quick start and Deep sleep modes provide extremely low power dissipation.
- Low power GTL + processor system bus interface (GTL : **G**unning **T**ransceiver **L**ogic)
- Integrated math co-processor (Floating point unit compatible with IEEE std 754).
- Integrated thermal diode for measuring processor temperature.

8.7.3 Pentium III

The Pentium III was released in 1999 and consists of 9.5 million transistors. The higher clock version of pentium III consists of 28 million transistors. The Pentium III is a 32-bit processor with a 64-bit data bus and a 36-bit address bus to address up to 64 Gb of physical memory space. It is available with maximum internal ratings of 500 MHz to 1 GHz.

In the IC form, it is available as 370-pin IC in PGA (**P**in **G**rid **A**rray) package. The Pentium III is an advanced version of pentium II with improved MMX technology and processor serial number. The INTEL has incorporated a processor serial number in Pentium III which supports the concept of processor identification. Each pentium III processor has a 96-bit processor number accessible by software (of various applications) to identify a system. Some of the applications that may utilize processor serial number are membership authentication, data backup/restore protection, removable storage data protection, managed access to files, etc.

The features of pentium III processor are:

- Dynamic execution microarchitecture.
- Optimized for 32-bit applications running on advanced 32-bit operating systems.
- Fully compatible with previous microprocessors.
- Integrated high performance 16 kb instruction and 16 kb data, nonblocking level one cache.
- Integrated 512 kb full speed level two cache allows for low latency on read/store operation.
- 256-bit cache data bus provides extremely high throughput on read/store operation.
- Eight-way cache associativity provides improved cache hit rate on read/store operations.
- Error correcting code for system bus data.
- Data prefetch logic
- Internet streaming SIMD (**S**ingle **I**nstruction **M**ultiple **D**ata) Extensions for enhanced Video, Sound and 3D performance.
- System management mode and multiple low-power states.
- **F**lip **C**hip **P**in **G**rid **A**rray (FC-PGA2) packaging technology which offers improved handling protection and socketability.
- Intel processor serial number.

8.7.4 Pentium 4

The Pentium 4 processor was released in 2000 and consists of 42 million transistors. It is available with maximum internal clock ratings of 1.4 GHz to 2.8 GHz. The Pentium 4 processor with HT (**H**yper **T**hreading) Technology was released in the year 2002 and consists of 55 million transistors. It is available with maximum internal clock ratings of 2.4 GHz to 3.3 GHz. It is available as 478-pin IC in PGA (**P**in **G**rid **A**rray) package.

The features of Pentium 4 processor are:

- INTEL NetBurst microarchitecture
- **H**yper **T**hreading (HT) technology.
- Hyperpipelined technology which supports advanced dynamic execution and very deep out-of-order execution.
- Rapid execution engine-ALUs run at twice the processor core frequency.
- System bus frequency at 400/533/800 MHz.
- Binary compatible with applications running on previous members of INTEL processors.
- 8 kb level 1 data cache.
- Level 1 execution trace cache stores 12 k micro-ops and removes decoder latency from main execution loops.
- 512 kb advanced transfer cache with 8-way associativity and error correcting code.
- 144 streaming SIMD Extensions 2(SSE2) instructions. (SIMD : **S**ingle **I**nstruction **M**ultiple **D**ata).
- System management mode and mulitple low power states.

The features of INTEL NetBurst microarchitecture are hyper pipelined technology, a rapid execution engine, 400/533/800 MHz system bus, execution trace cache, advanced dynamic execution, advanced transfer cache, enhanced floating point and multimedia unit and streaming SIMD Extensions 2 (SSE2).

The hyper pipelined technology doubles the pipeline depth in the Pentium-4 processor with 512 kb L2 cache, allowing the processor to reach much higher core frequencies. The rapid execution engine allows the two integer ALUs in the processor to run at twice the core frequency, which allows many integer instructions to execute in 1/2 clock tick.

The **H**yper **T**hreading (HT) technology allows a single physical Pentium-4 processor to function as two logical processors. Each logical processor has its own architecture state, own set of general purpose registers and control registers to provide increased system performance in multitasking environments.

CHAPTER 9

8086 MICROPROCESSOR-BASED SYSTEM

9.1 DESIGNING A MICROPROCESSOR-BASED SYSTEM

Desiging of a 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 a 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 the floppy disk controller - INTEL 8272. The Hex-keyboard is normally interfaced to the 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 the 8086 processor using INTEL 8279 keyboard and display controller. The 8279 is a dedicated controller which takes care of keyboard scanning and display refreshing. A maximum of 16 number of 7-segment LEDs can be interfaced using one 8279 in an 8086-based system as multiplexed display.

Other output devices are LCD (**L**iquid **C**rystal **D**isplay), 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 8086/8088 systems.

Memory Requirement

The memory requirement of the system is split 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 a word size of 1-byte. Hence, the memory capacity of the system is specified in kilobytes.

The popular EPROM used in the 8086-based system are 2708 (1 k × 8), 2716 (2 k × 8), 2732 (4 k × 8), 2764 (8 k × 8) and 27256 (32 k × 8). The popular static RAM used in the 8086-based system are 6208 (1 k × 8), 6216 (2 k × 8), 6232 (4 k × 8), 6264 (8 k × 8) and 62256 (32 k × 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 have to be generated using decoders. The input to the decoders are the unused address lines and also to each memory location specific addresses should be allotted. [These techniques are discussed in memory interfacing.]

The EPROMs are mapped at the end of memory space in 8086-based system in order to store the monitor program in EPROM and to execute the monitor program upon power-on-reset. [Every system will be reset, when power supply is switched ON.]

In an 8086-based system, the interrupt vector addresses belongs to RAM locations in the beginning of address space. The 8086 processor has 256 types of interrupts. For each interrupt, four locations are reserved in the first 1 kilobyte address space. In these locations, the offset and base address of the subroutine program to be executed in response to the interrupts are stored.

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/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.

The 8086 microprocessor does not have an internal clock circuit. Hence, the clock has to be supplied from an external device. The INTEL 8284 clock generator can be employed to generate the clock required for 8086. The 8284 has an internal oscillator circuit. An external quartz crystal has to be connected to the 8284 to generate the clock signal. The frequency of the quartz crystal should be thrice the internal clock frequency of 8086. The 8284 generates the clock at crystal frequency and divides the clock by three and then supplies to 8086.

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 the 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 a 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 requests.

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

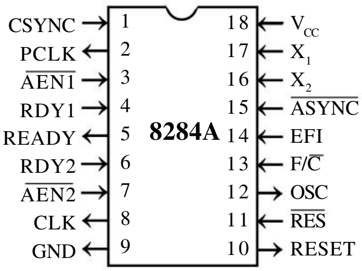
The specifications of the microprocessor itself depends on the applications for the proposed system and the nature of work it is going to perform. The input device, output device, memory requirement, peripheral requirement and the choice of CPU depend on the nature of work to be performed by the system.

9.2 CLOCK GENERATOR - INTEL 8284A

The clock generator 8284A is specially designed for 8086/8088 microprocessor. It generates the clock signal, READY signal and RESET signal required for 8086/8088 microprocessor. It also generates a TTL level peripheral clock signal which can be used for other peripheral devices in the system.

The INTEL 8284A is an 18-pin IC packed in DIP. The pin configuration of 8284A is shown in Fig. 9.1. The function of various pins are listed in Table-9.1.

A typical connection of 8284A with 8086 is shown in Fig. 9.2. The clock generator is used for generating clock signal and reset signal. It is assumed that the processor does not require wait states in bus cycles and so READY of 8086 is permanently tied to logic **high** (V_{CC}).



Pins	Description
CSYNC	Clock synchronization
PCLK	Peripheral clock
$\overline{\text{AEN1}}$, $\overline{\text{AEN2}}$	Address enable
RDY1, RDY2	Bus ready (Transfer complete)
READY	Ready signal to 8086/8088
CLK	Clock for 8086/8088
GND	Power supply/ground
RESET	Reset signal to 8086
$\overline{\text{RES}}$	Reset input to 8284A
OSC	Oscillator output
F/$\overline{\text{C}}$	Frequency/crystal selection
EFI	External frequency input
$\overline{\text{ASYNC}}$	Ready synchronization
X_1, X_2	Pins for crystal connection
V_{CC}	Power supply, +5-V

Fig. 9.1 : Pin configuration of 8284A.

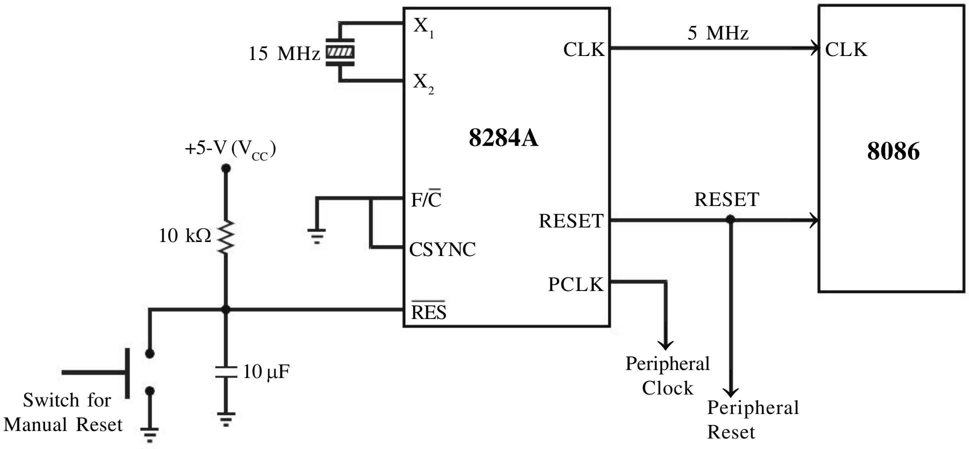


Fig. 9.2 : A typical connection of 8284A with 8086.

TABLE - 9.1 : PINS AND SIGNALS OF 8284A

Pin name	Function
CSYNC	When EFI is employed this clock synchronization input signal is used to synchronize the clocks of the various processors in a multimaster system. When crystal is connected to X_1 and X_2, this pin is grounded.
PCLK	It is a clock signal for peripheral devices in the system with a frequency of one-sixth the crystal frequency or EFI signal and has 50% duty cycle.
$\overline{AEN1}$, $\overline{AEN2}$	Address enables are provided to qualify (i.e., to allow) the bus ready signals RDY1 and RDY2 respectively. In a two-master system $\overline{AEN1}$ will qualify RDY1 and $\overline{AEN2}$ will qualify RDY2. In a single-master system $\overline{AEN1}$ and $\overline{AEN2}$ are tied to logic **low** (Ground).
RDY1, RDY2	These are active **high** signals from the devices located on the system data bus of each master to indicate that the data has been received or available. The RDY along with $\overline{AEN}$ are used to generate the READY signal for the 8086.
READY	The READY signal for the 8086 is supplied through this pin.
CLK	Clock signal for 8086. The frequency of this clock signal is one-third the frequency of crystal/EFI and has 33% duty cycle.
RESET	The reset signal for 8086 and other peripheral devices are supplied through this pin.
$\overline{RES}$	The reset input for 8284 and usually a RC network is connected to this pin to provide power-on reset.
OSC	A clock signal with same frequency as that of crystal/EFI. It can be used as EFI for other 8284s in a multimaster system.
$F/\overline{C}$	This is used to select crystal or EFI for clock generation. When EFI is used this pin is tied to logic **high** (V_{CC}). When a crystal is connected, this pin is tied to logic **low** (Ground).
EFI	This pin is used to supply external frequency input to 8284 when $F/\overline{C}$ is tied **high**. This external signal should be a square wave with frequency three times the frequency required for the CLK output.
$\overline{ASYNC}$	This signal is used to select one or two stages of synchronization for the RDY1 and RDY2 inputs. When this input is tied **low,** two stages of synchronization is provided and when this input is left open or tied **high,** one stage of synchronization is provided.
X_1, X_2	Pins for connecting quartz crystal. The frequency of crystal should be three times the desired processor clock frequency. When EFI is employed, X_1 should be tied to V_{CC} / GND and X_2 should be left open.
V_{CC}	Power supply input, +5-V
GND	Power supply ground

9.3 BUS CONTROLLER - INTEL 8288

The Bus controller-INTEL 8288 is specially designed for 8086/8088 microprocessor for use in maximum mode to generate bus control signals. It reads the status signals from the processor and generates bus control signal for memory and other IO devices.

The INTEL 8288 is a 20-pin IC packed in DIP. The pin configuration and internal block diagram of 8288 is shown in Fig. 9.3.

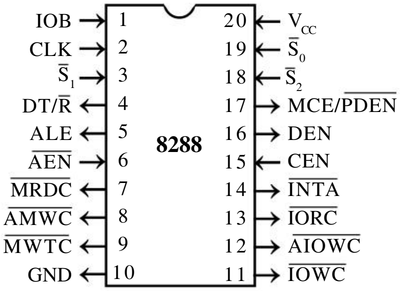


Fig. a : Pin configuration of INTEL 8288.

Pin	Description
$\overline{\text{AEN}}$	Address enable
$\overline{\text{AIOWC}}$	Advanced IO write control signal
ALE	Address latch enable
$\overline{\text{AMWC}}$	Advanced memory write control signal
CEN	Command enable
CLK	Clock input
DEN	Data enable
DT/$\overline{\text{R}}$	Data transmit/receive
$\overline{\text{INTA}}$	Interrupt acknowledge
IOB	IO bus mode
$\overline{\text{IORC}}$	IO read control signal
$\overline{\text{IOWC}}$	IO write control signal
MCE/$\overline{\text{PDEN}}$	Master cascade enable/ Peripheral data enable
$\overline{\text{MRDC}}$	Memory read control signal
$\overline{\text{MWTC}}$	Memory write control signal
$\overline{S}_0$, $\overline{S}_1$, $\overline{S}_2$	Input status signals
V_{CC}	Power supply, +5-V
GND	Ground, 0-V

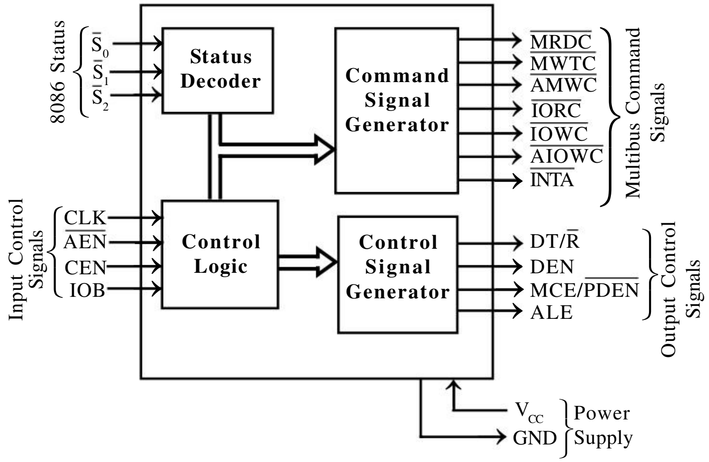


Fig. b : Internal block diagram of INTEL 8288.

Fig. 9.3: Bus controller-INTEL 8288.

The pins $\overline{S}_0$, $\overline{S}_1$, $\overline{S}_2$ are provided to receive the corresponding status bits from the 8086/8088 processor. The processor clock is directly connected to the CLK input of the 8288 in order to synchronize the activity of the bus controller with that of the processor. The signal output on ALE, DT/R and DEN are similar to that of 8086/8088 minimum mode signals for various bus cycles in order to enable the address latch and data transceivers. The bus controller issues the appropriate interrupt acknowledge signal through the $\overline{\text{INTA}}$ pin when the status signals are all zero (i.e., when $\overline{S}_0= \overline{S}_1= \overline{S}_2=0$).

The $\overline{\text{IORC}}$ and $\overline{\text{IOWC}}$ can be used as normal IO read and write control signals respectively. The $\overline{\text{AIOWC}}$ will provide extended IO write time, which can be used for writing IO devices requiring higher write time. The $\overline{\text{MRDC}}$ and $\overline{\text{MWTC}}$ can be used as normal memory read and write control signals respectively. The $\overline{\text{AMWC}}$ will provide extended memory write time, which can be used for writing memory devices requiring higher write time.

The $\overline{\text{AEN}}$, IOB and CEN are provided to configure the bus controller either for the uniprocessor or the multiprocessor system. In a uniprocessor system, $\overline{\text{AEN}}$ and IOB are tied to the ground (0-V) and CEN is tied to V_{CC}(+5-V). In a multiprocessor system, the $\overline{\text{AEN}}$ can be asserted **low/high** (by a bus arbiter such as INTEL 8289) to enable/disable the command outputs of 8288. For multiprocessor system IOB is tied to V_{CC}(+5-V). The signal output on MCE/$\overline{\text{PDEN}}$ depends on the mode, which is determined by the signal applied to IOB. In a uniprocessor mode, IOB is grounded and so the output of MCE/$\overline{\text{PDEN}}$ will be **high** and it is used to control the cascaded 8259s (cascaded interrupt controllers). In multiprocessor mode, IOB is tied to V_{CC} (+5-V) and in this mode the MCE/$\overline{\text{PDEN}}$ is asserted **low** during the IO read/write operation and this signal is used to enable IO bus data transceivers.

Note : In multiprocessor mode when an IO transfer is made, $\overline{\text{PDEN}}$ is active and DEN is inactive. For a memory transfer DEN is active and $\overline{\text{PDEN}}$ is inactive.

9.4 COPROCESSOR - INTEL 8087

The coprocessors has been specially designed to take care of mathematical calculations involving integer and floating point data. The coprocessor is also called math coprocessor or **N**umeric **D**ata **P**rocessor (NDP). A coprocessor is designed to work in parallel with a microprocessor. The INTEL has developed 80×87 series of coprocessors for 80×86 family of microprocessors. For example, INTEL has developed 8087 coprocessor for 8086/8088 processor, 80287 coprocessor for 80286 processor and so on. From 80486DX onwards INTEL has started integrating the coprocessor with the microprocessor and started fabricating microprocessors with an on-chip coprocessor.

The coprocessor-INTEL 8087 has been developed to work with 8086/8088 system in maximum mode. The 8087 coprocessor is a 40-pin IC packed in DIP. The pin configuration of 8087 is shown in Fig. 9.4.

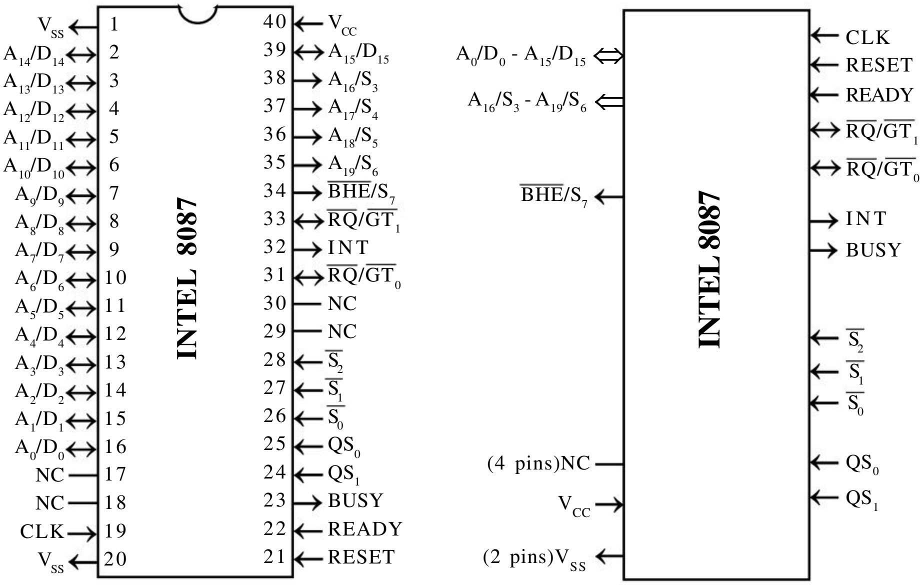


Fig. 9.4 : Pin configuration of INTEL 8087 coprocessor.

The 8087 has 16 numbers of multiplexed address/data pins and 4 numbers of multiplexed address/status pins. Hence, it can have 16-bit external data bus and 20-bit external address bus like 8086. (In case of 8088 system, the external data bus is 8-bit.) The processor clock, reset and ready signals are applied as clock, reset and ready signals for coprocessor. The BUSY output is used as $\overline{TEST}$ input of the processor. One of the Request/Grant ($\overline{RQ}/\overline{GT}$) signal can be connected to the corresponding pin of the processor to get the control of bus from the processor. During an internal error condition, the coprocessor asserts the INT output as **high**, which can be used to interrupt the processor to take appropriate action. The pins $\overline{S}_2$, $\overline{S}_1$ and $\overline{S}_0$ are used to receive the bus status signal from the processor and the pins QS_1 and QS_0 are used to receive the queue status from the processor. When used with 8086, the $\overline{BHE}$ signal is asserted to enable the upper memory bank.

Architecture of 8087

The internal architecture of 8087 is shown in Fig. 9.5. The 8087 has two funtional units: **C**ontrol **U**nit (CU) and **N**umeric **E**xecution **U**nit (NEU).

The control unit consists of data buffer, shared operands queue, addressing and bus tracking unit, exception pointers and control and status word registers.

The numeric execution unit consists of eight numbers of 80-bit register stack, microcode control unit, exponent module, programmable shifter, arithmetic module, temporary registers and shared operands queue.

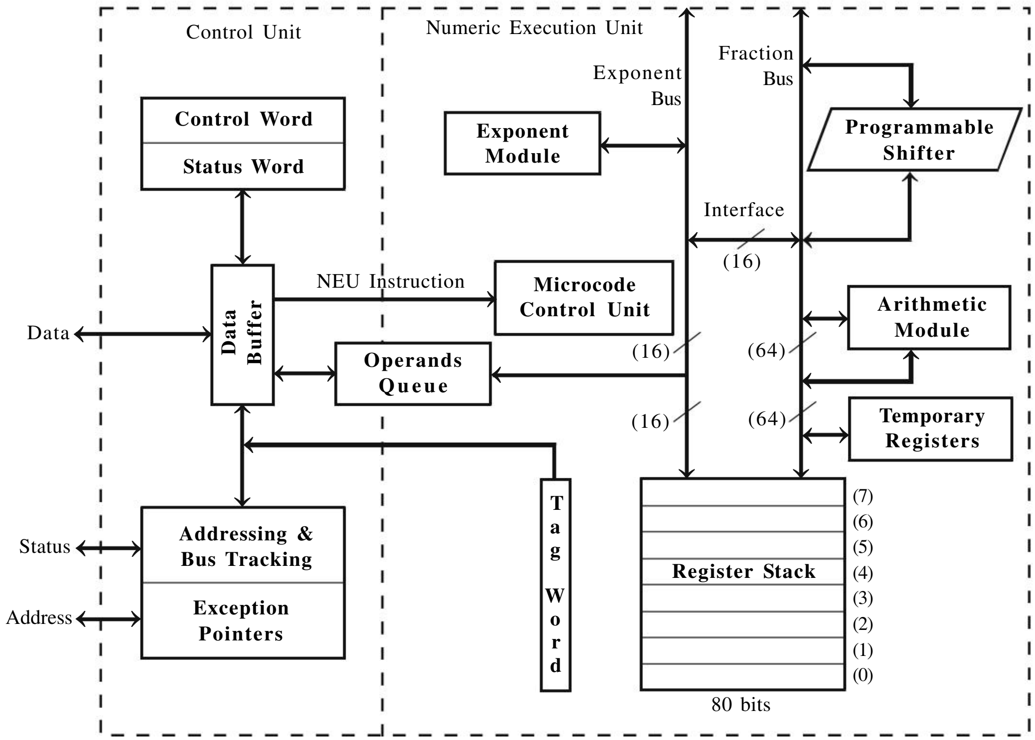


Fig. 9.5 : Architecture of coprocessor-INTEL 8087.

The NEU is responsible for the execution of the coprocessor instructions under the control of the coprocessor CU. The CU transfers the numeric instructions to the microcode control unit of NEU.

The 8087 internally operates on all numbers in the 80-bit temporary real format. The 8087 has eight numbers of 80-bit registers which are used as LIFO(**L**ast-**I**n-**F**irst-**O**ut) stack. This register stack holds the operands on which the coprocessor instruction operates. The 8087 has a 3-bit stack pointer to point to current top of stack. The stack pointer can hold binary values 000_2 to 111_2 to represent the eight registers of the stack. The stack operates as a circular stack of fixed size (8 elements) on the basis of LIFO access. Upon reset, the stack pointer is initialized with 000_2.

The coprocessor works with seven types of numeric data, which are divided into the following three classes:

i) Word integer (16 -bit) ii) Short integer (32 -bit) iii) Long integer (64 -bit)	Binary integers
iv) BCD (80 -bit)	Packed decimal number
v) Short real (32 -bit) vi) Long real (64 -bit) vii) Temporary real (80 -bit)	Real numbers

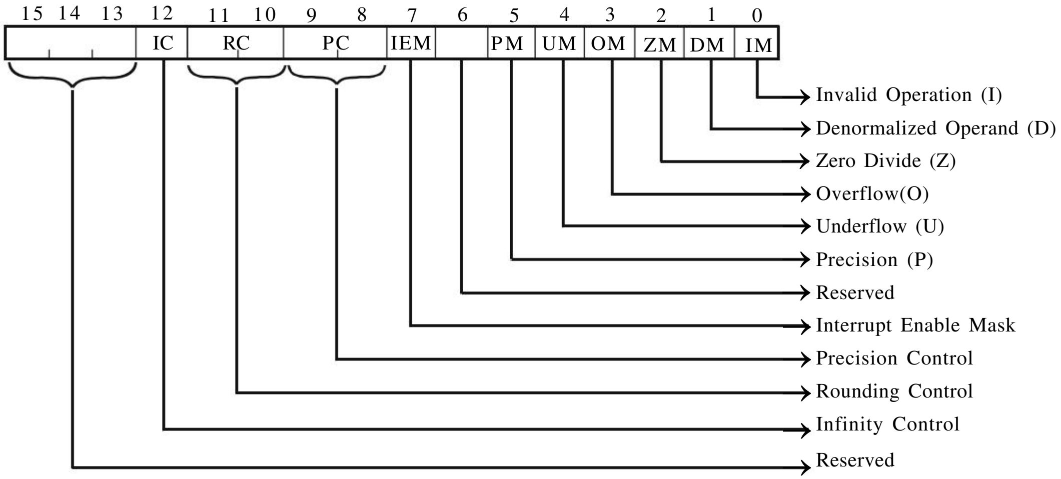


Fig. 9.6a : Format of control word.

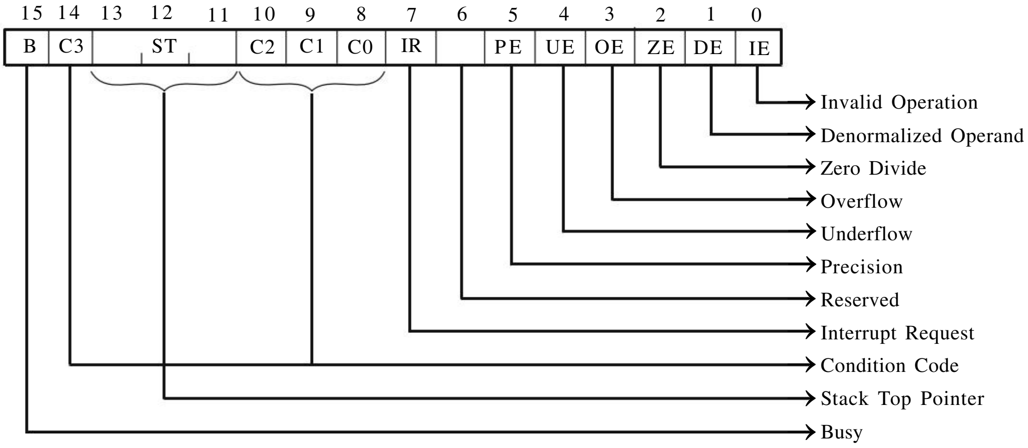


Fig. 9.6b : Format of status word.

Fig. 9.6 : Control and Status word format of the coprocessor 8087.

The 8087 has a 16-bit control word and a 16-bit status word. The format of the control word and the status word are shown in Fig. 9.6.

In order to write the control word to the control register, the processor has to write the control word to a memory location. Then the coprocessor has to read the control word from memory and load it in the control register. Similarly, to read the status word, first the coprocessor has to transfer/copy the content of status register to a memory location and then the processor has to read the status word from the memory location. Actually, the communication of data between the microprocessor and the coprocessor takes place through the main memory only.

Working of 8087 with 8086 Processor

The 8087 coprocessor has its own set of instructions. When a program is written the required instructions of 8087 are written along with the 8086 instructions. The instructions of 8087 always start with the letter F and this differentiates the 8087 instruction from the 8086 instruction in an assembly language program.

When the 8086 fetches instructions from the memory and stores them in its queue, the 8087 also reads these instructions and stores them in its internal queue. Here both the processor and coprocessor read and decode every instruction but execute only its own instruction. The most significant bits of all the 8087 instructions will be "11011" and this code is used to identify the 8087 instruction by the processor and coprocessor. Every instruction of the program is read and decoded by both the processor and the coprocessor. After decoding, if the processor identifies an 8087 instruction then it treats that instruction as NOP(**N**o-**op**eration). Similarly, if the coprocessor identifies an 8086 instruction then it treats that instruction as NOP(**N**o-**op**eration).

9.5 MINIMUM MODE 8086-BASED SYSTEM

In minimum mode, the 8086 processor itself generates all bus control signals and so there is no need for an external bus controller. The 8086 processor has multiplexed address/data pins and address/status pins. In a system, multiplexing is not allowed and so the multiplexed address lines has to be demultiplexed by using external latches and the latches are enabled by using the signal ALE supplied by the processor. In an 8086-based system, the data bus should be provided with data transceivers to drive the data on the bus. The signal DEN is used as enable and the signal DT/$\overline{\text{R}}$ is used as direction control for data transceivers.

The formation of address bus and data bus in the 8086-based minimum mode system is shown in Fig. 9.7. For minimum mode of operation the MN/$\overline{\text{MX}}$ is tied to V_{cc}(+5-V). The clock generator of the 8284 is used to generate the clock, reset and ready signals for the processor. A quartz crystal of frequency 15 MHz is connected to the X_1 and X_2 pins of 8284 so that the clock frequency supplied to the 8086 processor will be 5 MHz. An RC circuit is connected to the reset input of 8284 to provide power-ON reset. A switch is also connected across the capacitor to provide manual reset.

In the system shown in Fig. 9.7, three numbers of 8-bit latch 74LS573 are used as address latches and two numbers of 8-bit bidirectional buffer 74LS245 are used as data transceivers. The interfacing of memory and IO (or peripheral) devices are discussed in Chapters 4 and 7.

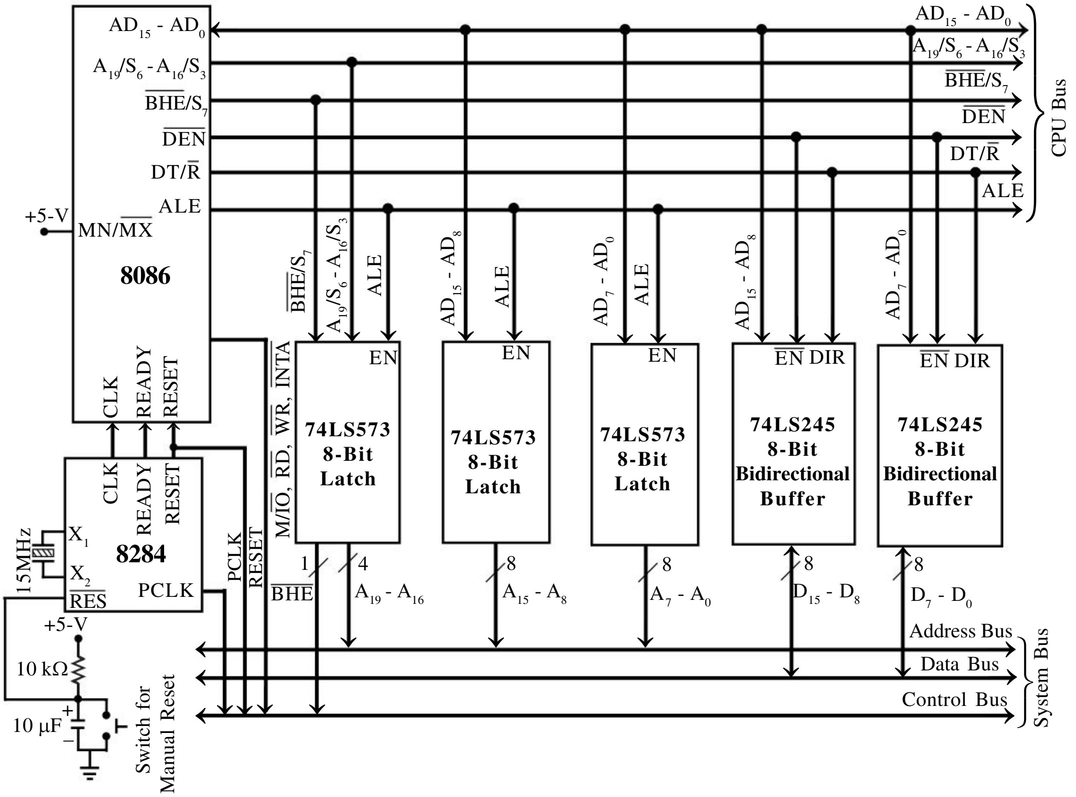


Fig. 9.7 : Formation of system bus in 8086-based minimum mode system.

9.6 MAXIMUM MODE 8086-BASED SYSTEM

In maximum mode 8086-based system, an external bus controller 8288 has to be employed to generate the bus control signals. The 8288 can be configured for uniprocessor or multiprocessor mode of operation using the signals, $\overline{\text{AEN}}$, IOB and CEN. The formation of address bus and data bus in 8086-based maximum mode system is shown in Fig. 9.8. For maximum mode of operation, the pin MN/$\overline{\text{MX}}$ of 8086 processor is tied to the ground.

The system shown in Fig. 9.8 employs a bus controller 8288 to generate bus control signals. Here the bus controller is configured for the uniprocessor mode of operation by grounding $\overline{\text{AEN}}$ and IOB, and by applying +5-V to CEN. (For multiprocessor mode of operation, IOB should be tied to +5-V and the signals $\overline{\text{AEN}}$ and CEN are supplied by a bus arbiter such as INTEL 8289.)

In 8086 processor, the address is multiplexed with the data or status signals. In a system, multiplexing is not allowed and so the multiplexed address lines of the CPU bus has to be demultiplexed by using external latches. In the system shown in Fig. 9.8, three numbers of 8-bit latch 74LS573 are employed to demultiplex the address lines. The signal ALE generated by the bus controller is used as enable for the latches.

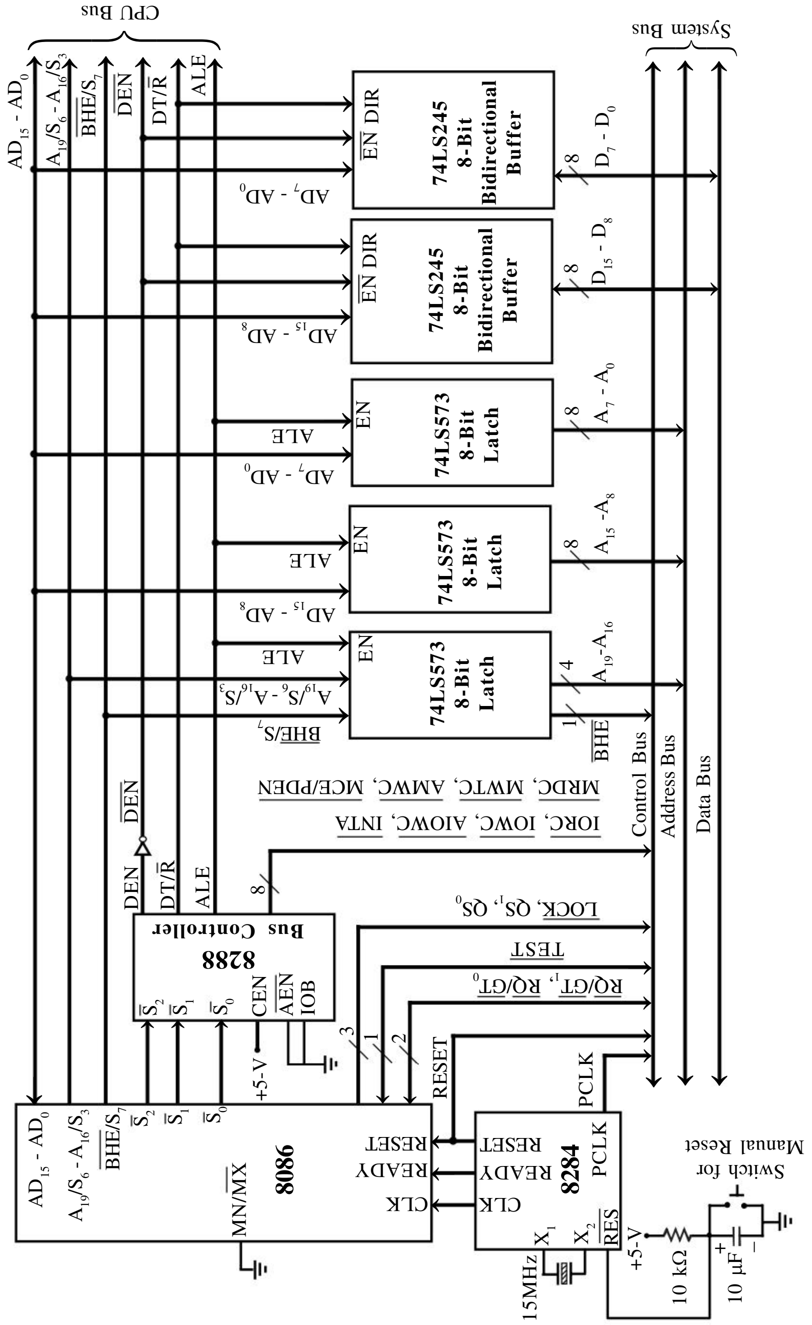


Fig. 9.8 : Formation of system bus in 8086-based maximum mode system.

In 8086-based system, the data bus should be provided with data transceivers to drive the data on the bus. In the system shown in Fig. 9.8, two numbers of bidirectional buffer 74LS245 are employed as data transceivers. The signals DEN and $DT/\overline{R}$ generated by the bus controller are used as enable and direction control of buffers respectively.

The system employs a clock generator INTEL 8284 to generate the clock, reset and ready signals for the 8086 processor. A quartz crystal of frequency 15 MHz is connected to the X_1 and X_2 pins of 8284 so that the clock frequency supplied to the 8086 processor will be 5 MHz. An RC circuit is connected to the reset input of the 8284 to provide power-ON reset. A switch is also connected across the capacitor to provide manual reset.

The bus controller generates separate read and write controls for memory and IO devices. It also generates extended write control signal for memory and IO devices requiring higher write time.

9.7 MULTIPROCESSOR CONFIGURATIONS

A multiprocessor system will have two or more processors that can execute instructions (or perform operations) simultaneously. In multiprocessor systems, the extra or added processors can be special purpose processors which are specifically designed to perform certain tasks efficiently or can be other general purpose processors. For example, a multiprocessor system can be formed by using an 8086 microprocessor and an 8087 coprocessor in order to impart efficient floating point arithmetic capability to the 8086-based system.

The multiprocessor systems offers the following advantages over single processor design:

1. Several low cost processors may be combined to fit the needs of an application while avoiding the expense of the unneeded capabilities of a centralized system.
2. The multiprocessor system provides room for expansion because it is easy to add more processors as the need arises.
3. In a multiprocessor system, implementation of modular processing of task can be achieved.
4. When a failure occurs, it is easier to replace only the faulty part/processor.

The two major issues in multiprocessor system design are bus contention and interprocessor communication. In a multiprocessor system, more than one processor will share the system memory and IO devices through a common system bus and so extra logic must be included to ensure that only one processor has access to the system bus at any one time. Also, there should be an unambiguous way of interprocessor communication so that one processor can despatch a task or return a result to another processor unambiguously.

The maximum mode 8086/8088 microprocessor has features for designing a multiprocessor system. Two types of multiprocessor configurations can be formed using 8086/8088 processor: closely coupled (or tightly coupled) configuration and loosely coupled configuration.

In an 8086-based system, only two or three processors can be connected to work in closely coupled configuration and in this, 8086/8088 is the host/master processor and other/supporting processor is the slave processor. The slave processor can be a coprocessor or an IO processor or general purpose processor. Two numbers of 8086/8088 processors cannot work in closely coupled configurations. In a closely coupled configuration, both the master and slave processors share the same bus control logic, clock generator, system memory and IO. In a closely coupled configurations each processor will not have its own local memory or IO. The general block diagram of a closely coupled configuration is shown in Fig. 9.9.

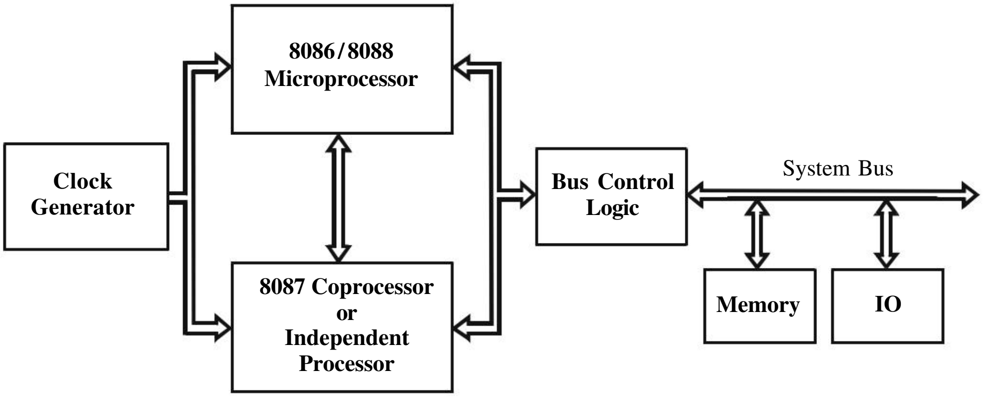


Fig. 9.9 : Closely coupled configuration.

In a closely coupled configuration, the bus access control is provided by the master/host processor and so the bus request of the supporting/slave processor is connected to the master (In case of 8086/8088 the signal $\overline{\text{RQ}}$, $\overline{\text{GT}}$ is used for bus request and grant.) In a closely coupled configuration, when the slave is a coprocessor, it interacts directly with the master and to a certain extent its functioning depends on the master. But when the slave is another processor then it can work independently.

In a loosely coupled configuration, a number of modules of microprocessor-based system (or masters) can be interfaced through a common system bus to work as a multiprocessor system. Each module in the loosely coupled configuration is an independent microprocessor-based system with its own clock source, and its own memory and IO devices interfaced through a local bus. Each module can also be a closely coupled configuration of a processor and coprocessor. The block diagram of a loosely coupled configuration using three modules of 8086/8088 is shown in Fig. 9.10.

Each module in a loosely coupled configuration functions independently and there is no direct connection between them. The modules can have access to system resources (system memory and IO) through the system bus. Each module has a local and system bus control logic which takes care of interprocessor communication and system bus allocation to the masters/modules competing for system resources.

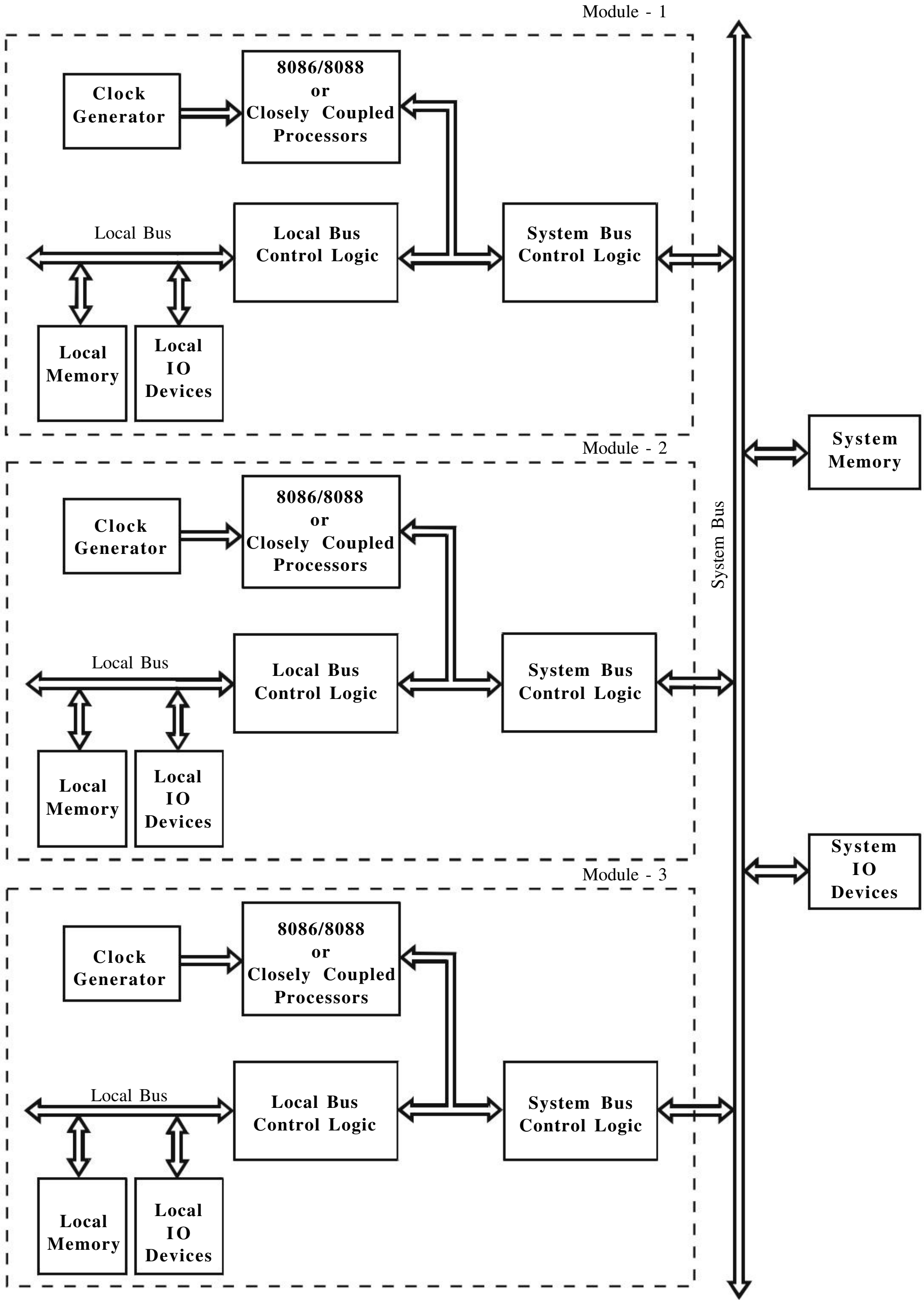


Fig. 9.10 : Loosely coupled configuration.

9.8 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 an 8086 microprocessor-based temperature control system is shown in Fig. 9.11.

The system consists of 8086 microprocessor in minimum mode 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 the 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 the keyboard. The 7-segment display has been provided to display the temperature of the plant at any time instant.

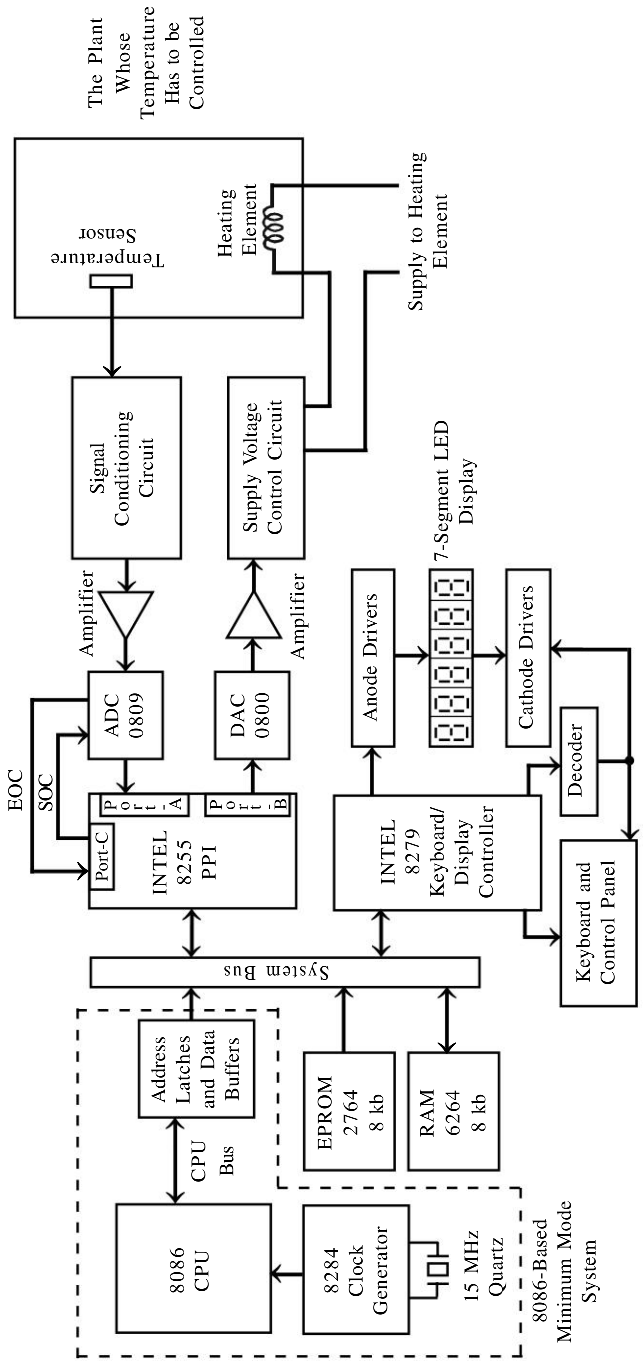
The temperature of the plant is measured using a temperature sensor. The different types of temperature sensors that can be used for temperature measurement are thermo-couples, thermistors, PN-junctions, IC sensors like AD590, etc. These sensors will convert the input temperature to proportional analog 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 a 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 **A**nalog-to-**D**igital **C**onverter (ADC) to convert the analog signal to proportional digital data. In this system, the ADC is interfaced to 8086 processor through port-A and port-C of 8255. The 8086 processor sends 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 8086 processor calculates the actual temperature using the input data and displays it on the 7-segment LED. The processor also compares the desired temperature with actual temperature (the operator can enter the desired temperature through the keyboard) and calculate the error (the difference between actual temperature and 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 8086 processor using different digital control algorithms (P/PI/PID/FUZZY logic control algorithms).


The Plant Whose Temperature Has to be Controlled
Temperature Sensor
Heating Element
Supply to Heating Element
Signal Conditioning Circuit
Supply Voltage Control Circuit
Amplifier
Amplifier
ADC 0809
DAC 0800
EOC
SOC
Port-C
Port-A
Port-B
INTEL 8255 PPI
Anode Drivers
7-Segment LED Display
Cathode Drivers
Decoder
INTEL 8279 Keyboard/Display Controller
Keyboard and Control Panel
System Bus
Address Latches and Data Buffers
CPU Bus
EPROM 2764 8 kb
RAM 6264 8 kb
8086 CPU
8284 Clock Generator
15 MHz Quartz
8086-Based Minimum Mode System

Note : Refer Fig. 9.7 for 8086-based minimum mode system.

Fig. 9.11 : 8086 microprocessor-based temperature control system.

The control circuit for power supply can be either a 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. 9.12.

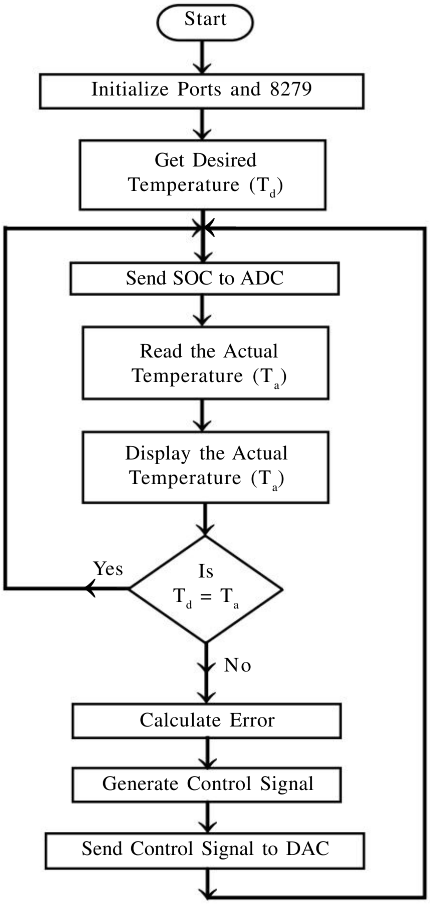

Start
Initialize Ports and 8279
Get Desired Temperature (T_d)
Send SOC to ADC
Read the Actual Temperature (T_a)
Display the Actual Temperature (T_a)
Yes
Is $T_d = T_a$
No
Calculate Error
Generate Control Signal
Send Control Signal to DAC

Fig. 9.12 : Flowchart for temperature control system.

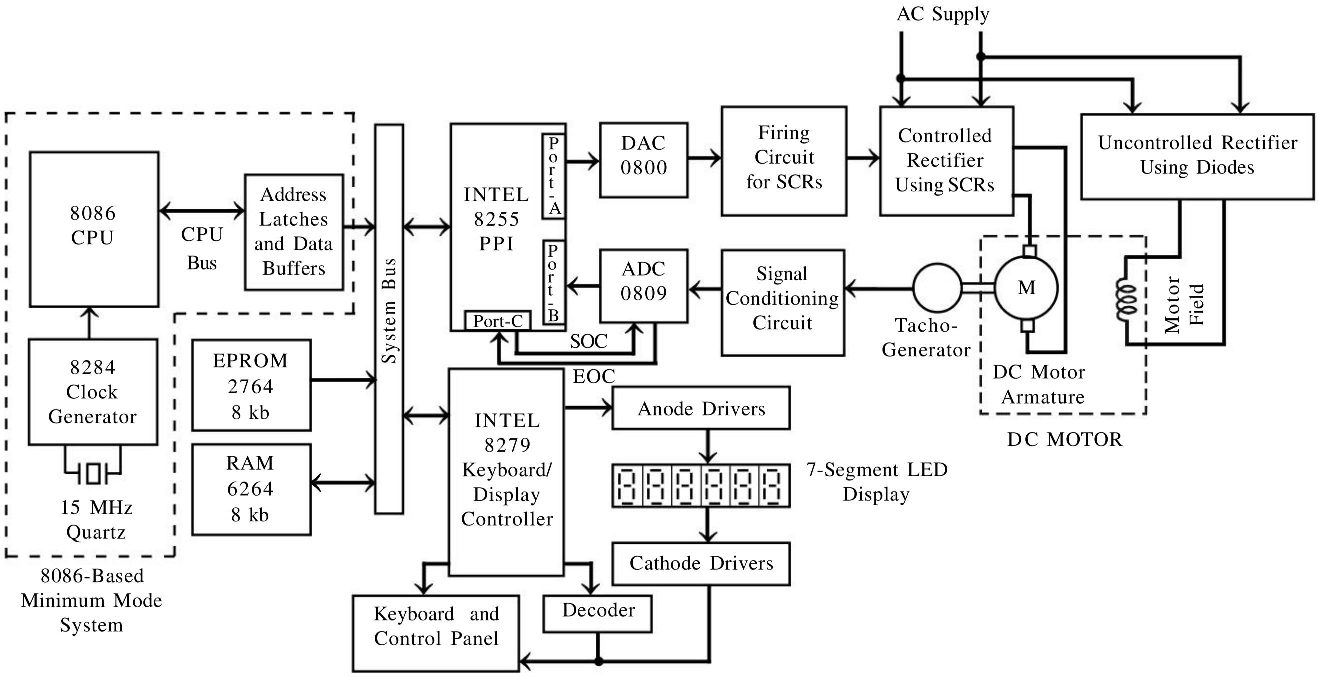
9.9 MOTOR SPEED CONTROL SYSTEM

The microprocessor-based speed control system can be used to automatically control the speed of a motor. A typical 8086 microprocessor-based dc motor speed control system is shown in Fig. 9.13. 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 the 8086 microprocessor in minimum mode as CPU. The system has EPROM for system program storage, and RAM for temporary data storage and stack. 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, a 7-segment LED display has been provided. The keyboard and 7-segment LED display has been interfaced to the 8086-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 the 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 the 8086 processor through 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 the actual speed.



Note : Refer Fig. 9.7 for 8086-based minimum mode system.

Fig. 9.13 : 8086 microprocessor-based dc motor speed control system.

The processor calculates the actual speed and displays it on LEDs. The processor also compares the actual speed with desired speed entered by the operator through the keyboard. If there is a difference then the 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. 9.14.

9.10 TRAFFIC LIGHT CONTROL SYSTEM

The traffic lights placed at the road crossings can be automatically switched ON/OFF in the desired sequence using a 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.

A typical traffic light control system (demonstration type) is shown in Fig. 9.15. The system has been developed using 8086 microprocessor in minimum mode 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.

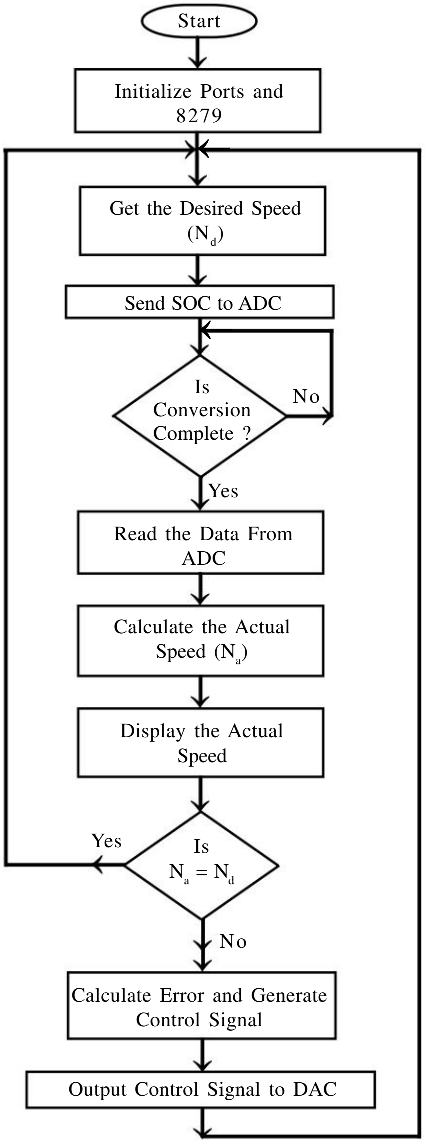
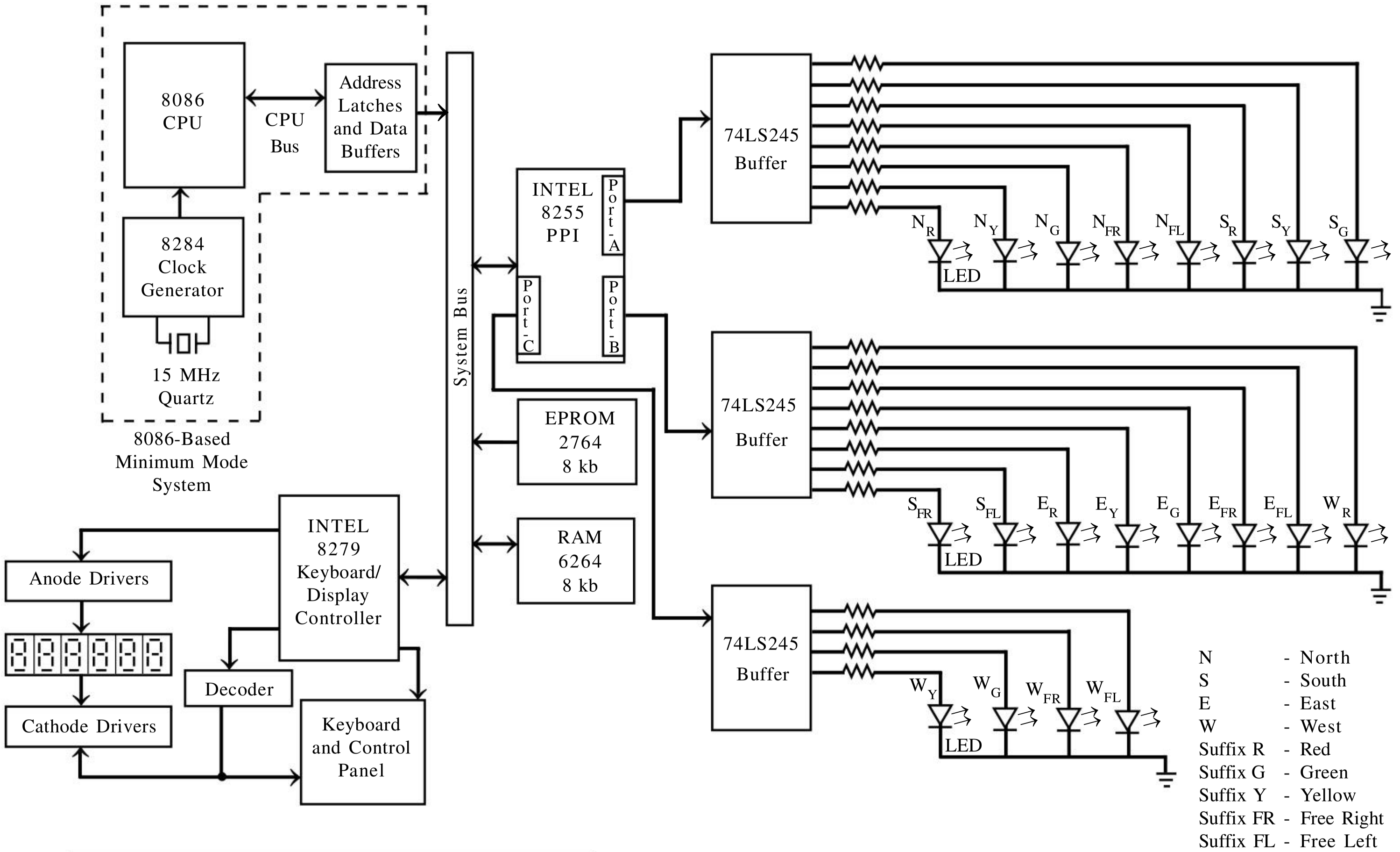

Start
Initialize Ports and 8279
Get the Desired Speed (N_d)
Send SOC to ADC
Is Conversion Complete ?
No
Yes
Read the Data From ADC
Calculate the Actual Speed (N_a)
Display the Actual Speed
Yes
Is $N_a = N_d$
No
Calculate Error and Generate Control Signal
Output Control Signal to DAC

Fig. 9.14 : Flowchart for a dc motor speed control system.

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. 9.15, Red/Yellow/Green LEDs are provided instead of lights (lamps). The LEDs are interfaced to the system through buffer (74LS245) and ports of 8255.



Note : Refer Fig. 9.7 for 8086-based minimum mode system.

Fig. 9.15 : 8086 microprocessor-based traffic light control demonstration system.

TABLE -9.2 : SWITCHING SCHEDULE FOR TRAFFIC LIGHTS

	ON/OFF status of traffic lights																			
Switching	PC_3	PC_2	PC_1	PC_0	PB_7	PB_6	PB_5	PB_4	PB_3	PB_2	PB_1	PB_0	PA_7	PA_6	PA_5	PA_4	PA_3	PA_2	PA_1	PA_0
Schedule	W_{FL}	W_{FR}	W_G	W_Y	W_R	E_{FL}	E_{FR}	E_G	E_Y	E_R	S_{FL}	S_{FR}	S_G	S_Y	S_R	N_{FL}	N_{FR}	N_G	N_Y	N_R
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.*

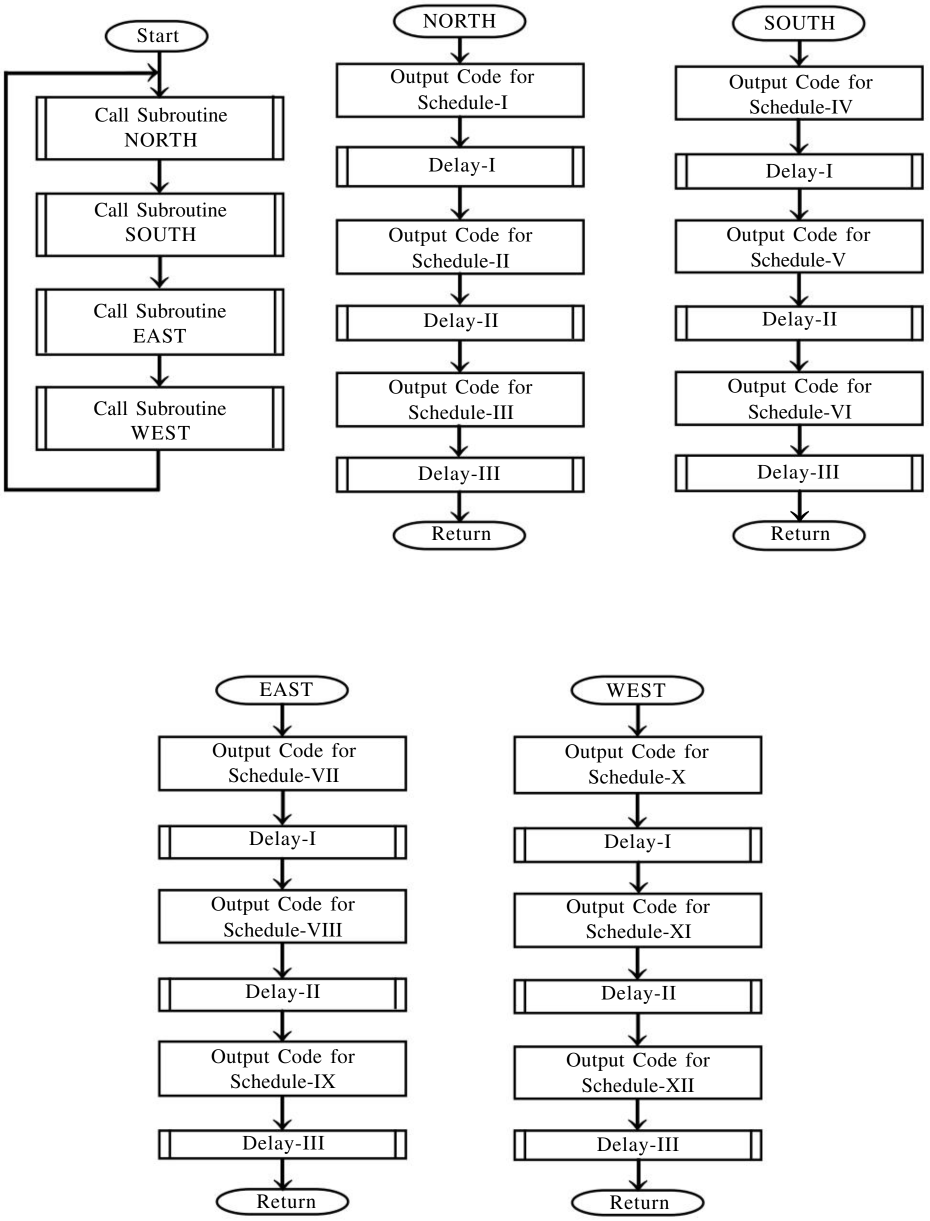

Start
Call Subroutine NORTH
Call Subroutine SOUTH
Call Subroutine EAST
Call Subroutine WEST
NORTH
Output Code for Schedule-I
Delay-I
Output Code for Schedule-II
Delay-II
Output Code for Schedule-III
Delay-III
Return
SOUTH
Output Code for Schedule-IV
Delay-I
Output Code for Schedule-V
Delay-II
Output Code for Schedule-VI
Delay-III
Return
EAST
Output Code for Schedule-VII
Delay-I
Output Code for Schedule-VIII
Delay-II
Output Code for Schedule-IX
Delay-III
Return
WEST
Output Code for Schedule-X
Delay-I
Output Code for Schedule-XI
Delay-II
Output Code for Schedule-XII
Delay-III
Return

Fig. 9.16 : 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. 9.17. A reverse biased diode is connected across the relay coil to prevent relay chattering (for free-wheeling action).

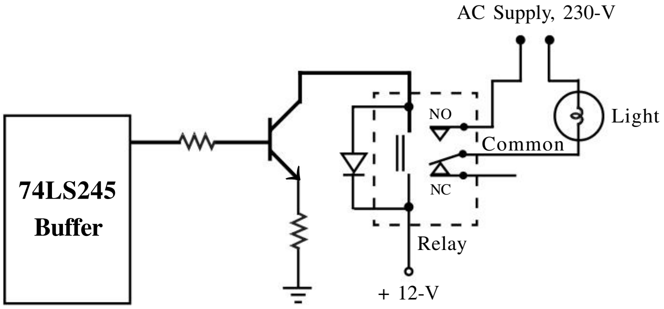


Fig. 9.17 : Switching circuit for traffic light.

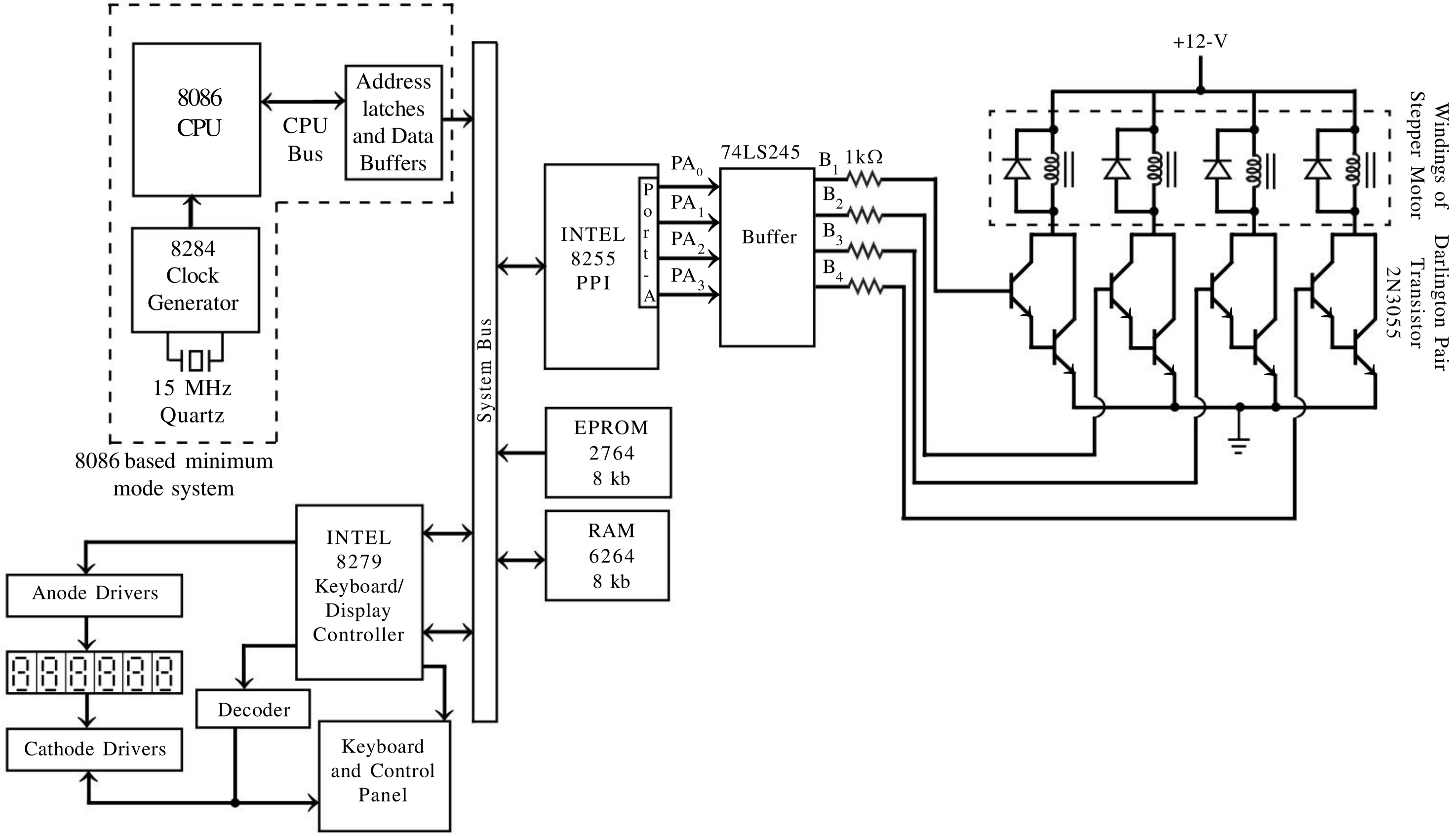
The microprocessor sends **high** through a port line to switch ON the light and **low** to switch OFF the light. A switching schedule (or sequence) can be developed as shown in Table-9.2. In this switching sequence, it is assumed that the traffic is allowed only in one direction at a time. In Table-9.2, "1" represents ON condition and "0" represents OFF condition. These 1's and 0's can be directly output to 8255 ports to switch ON/OFF the light. A flowchart for traffic light control program is shown in Fig. 9.16.

The processor can output the codes for switching the lights for schedule-I and then waits. After a specified time delay the processor outputs 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.

9.11 STEPPER MOTOR CONTROL SYSTEM

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° (i.e., 200 steps per revolution). This motor consists of four stator windings and requires four switching sequences as shown in



Note : Refer fig 9.7 for 8086 based minimum mode system.

Fig. 9.18 : An 8086 microprocessor-based stepper motor control system.

Table-9.3. 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-9.4.

A typical stepper motor control system is shown in Fig. 9.18, for a two-phase or four winding stepper motor. The system consists of an 8086 microprocessor in minimum mode 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 have been provided to display messages to the operator.

The windings of the stepper motor are connected to the collector of the 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 .9.19. The processor has to output a switching sequence and wait for 1 to 5 milliseconds before sending the next switching sequence. (The delay is necessary to allow the motor transients to die out.)

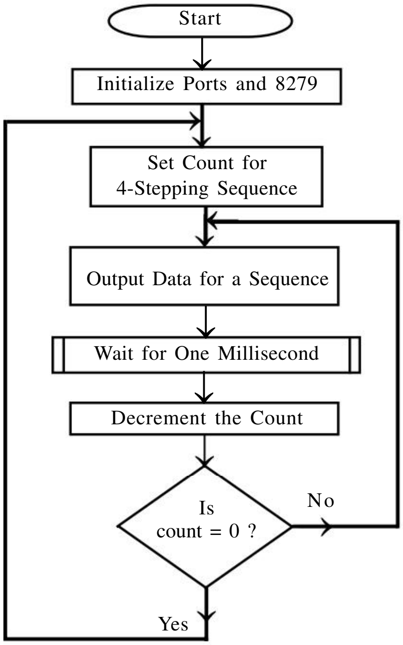

Start
Initialize Ports and 8279
Set Count for 4-Stepping Sequence
Output Data for a Sequence
Wait for One Millisecond
Decrement the Count
Is count = 0 ?
No
Yes

Fig. 9.19 : Flowchart for stepper motor control program.

TABLE-9.3 : SWITCHING SEQUENCE FOR FULL-STEP ROTATION

Switching sequence	Clockwise rotation				Anticlockwise rotation			
	PA_3	PA_2	PA_1	PA_0	PA_3	PA_2	PA_1	PA_0
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-9.4 : SWITCHING SEQUENCE FOR HALF-STEP ROTATION

Clockwise rotation				Anticlockwise rotation			
PA_3	PA_2	PA_1	PA_0	PA_3	PA_2	PA_1	PA_0
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

APPENDIX I : TEMPLATES FOR 8086 INSTRUCTIONS

S.No.	Mnemonic	Templates					
Group - I Data Transfer Instructions							
1.	Mov reg2/mem, reg1/mem						
	a) MOV reg2, reg1	1000 10dw	mod reg r/m				
	b) MOV mem, reg1 c) MOV reg2, mem	1000 10dw	mod reg r/m	l.b.disp	h.b.disp		
2.	MOV reg/mem, data						
	a) MOV reg, data	1100 011w	mod 000 r/m	l.b.data	h.b.data		
	b) MOV mem, data	1100 011w	mod 000 r/m	l.b.disp	h.b.disp	l.b.data	h.b.disp
3.	MOV reg, data	1011 wreg	l.b.data	h.b.data			
4.	MOV A, mem	1010 000w	l.b.disp	h.b.disp			
	a) MOV AL, mem						
	b) MOV AX, mem						
5.	MOV mem, A	1010 001w	l.b.disp	h.b.disp			
	a) MOV mem, AL						
	b) MOV mem, AX						
6.	MOV segreg, reg16/mem						
	a) MOV segreg, reg16	1000 1110	mod 0 sr r/m				
	b) MOV segreg, mem	1000 1110	mod 0 sr r/m	l.b.disp	h.b.disp		
7.	MOV reg16/mem, seg reg						
	a) MOV reg16, segreg	1000 1100	mod 0 sr r/m				
	b) MOV mem, segreg	1000 1100	mod 0 sr r/m	l.b disp	h.b.disp		
8.	PUSH reg16/mem						
	a) PUSH reg16	1111 1111	mod 110r/m				
	b) PUSH mem	1111 1111	mod 110 r/m	l.b disp	h.b.disp		
9.	PUSH reg16	0101 0 reg					
10.	PUSH segreg	000 sr 110					
11.	PUSHF	1001 1100					
12.	POP reg16/mem						
	a) POP reg16	1000 1111	mod 000r/m				
	b) POP mem	1000 1111	mod 000 r/m	l.b disp	h.b.disp		
13.	POP reg16	0101 1 reg					
14.	POP segreg	000 sr 111					
15.	POPF	1001 1101					

Appendix I continued ...

S.No.	Mnemonic	Templates					
16.	XCHG reg2/mem, reg1 a) XCHG reg2, reg1	1000 011w	mod reg r/m				
	b) XCHG mem, reg1	1000 011w	mod reg r/m	l.b disp	h.b.disp		
17.	XCHG AX, reg16	1001 0 reg					
18.	XLAT	1101 0111					
19.	IN A, [DX] a) IN AL, [DX] b) IN AX, [DX]	1110 110w					
20.	IN A, addr8 a) IN AL, addr8 b) IN AX, addr8	1110 010w	addr8				
21.	OUT [DX], A a) OUT [DX], AL b) OUT [DX], AX	1110 111w					
22.	OUT addr8, A a) OUT addr8, AL b) OUT addr8, AX	1110 011w	addr8				
23.	LEA reg16, mem	1000 1101	mod reg r/m	l.b.disp	h.b. disp		
24.	LDS reg16, mem	1100 0101	mod reg r/m	l.b.disp	h.b. disp		
25.	LES reg16, mem	1100 0100	mod reg r/m	l.b.disp	h.b. disp		
26.	LAHF	1001 1111					
27.	SAHF	1001 1110					
Group - II Arithmetic Instructions							
28.	ADD reg2/mem,reg1/mem a) ADD reg2, reg1	0000 00dw	mod reg r/m				
	b) ADD reg2, mem c) ADD mem, reg1	0000 00dw	mod reg r/m	l.b. disp	h.b. disp		
29.	ADD reg/mem, data a) ADD reg, data	1000 00sw	mod 000 r/m	l.b. data	h.b. data		
	b) ADD mem, data	1000 00sw	mod 000 r/m	l.b. disp	h.b. disp	l.b. data	h.b. data
30.	ADD A, data a) ADD AL, data8 b) ADD AX, data16	0000 010w	l.b. data	h.b. data			
31.	ADC reg2/mem,reg1/mem a) ADC reg2, reg1	0001 00dw	mod reg r/m				
	b) ADC reg2, mem c) ADC mem, reg1	001 00dw	mod reg r/m	l.b. disp	h.b.disp		

Appendix I continued ...

S.No.	Mnemonic	Templates					
32.	ADC reg/mem, data						
	a) ADC reg, data	1000 00sw	mod 010 r/m	l.b.data	h.b.data		
	b) ADC mem, data	1000 00sw	mod 010 r/m	l.b.disp	h.b.disp	l.b.data	h.b.data
33.	ADC A, data	0001 010w	l.b.data	h.b.data			
	a) ADC AL, data8						
	b) ADC AX, data16						
34.	AAA	0011 0111					
35.	DAA	0010 0111					
36.	SUB reg2/mem,reg1/mem						
	a) SUB reg2, reg1	0010 10dw	mod reg r/m				
	b) SUB reg2, mem c) SUB mem, reg1	0010 10dw	mod reg r/m	l.b.disp	h.b.disp		
37.	SUB reg/mem, data						
	a) SUB reg, data	1000 00sw	mod 101 r/m	l.b. data	h.b.data		
	b) SUB mem, data	000 00sw	mod 101 r/m	l.b.disp	h.b.disp	l.b.data	h.b.data
38.	SUB A, data	0010 110w	l.b.data	h.b.data			
	a) SUB AL, data8						
	b) SUB AX, data16						
39.	SBB reg2/mem,reg1/mem						
	a) SBB reg2, reg1	0001 10dw	mod reg r/m				
	b) SBB reg2, mem 0001 10dw c) SBB mem, reg1	mod reg r/m	l.b.disp	h.b.disp			
40.	SBB reg/mem, data						
	a) SBB reg, data	1000 00sw	mod 011 r/m	l.b.data	h.b.data		
	b) SBB mem, data	1000 00sw	mod 011 r/m	l.b.disp	h.b.disp	l.b.data	h.b.data
41.	SBB A, data	0001 110w	l.b.data	h.b.data			
	a) SBB AL, data8						
	b) SBB AX, data16						
42.	AAS	0011 1111					
43.	DAS	0010 1111					
44.	MUL reg/mem						
	a) MUL reg	1111 011w	mod 100 r/m				
	b) MUL mem	1111 011w	mod 100 r/m	l.b.disp	h.b.disp		
45.	IMUL reg/mem						
	a) IMUL reg	1111 011w	mod 101 r/m				
	b) IMUL mem	1111 011w	mod 101 r/m	l.b.disp	h.b.disp		

Appendix I continued ...

S.No.	Mnemonic	Templates					
46.	AAM	1101 0100	0000 1010				
47.	DIV reg/mem						
	a) DIV reg	1111 011w	mod 110 r/m				
	b) DIV mem	1111 011w	mod 110 r/m	l.b.disp	h.b.disp		
48.	IDIV reg/mem						
	a) IDIV reg	1111 011w	mod 111 r/m				
	b) IDIV mem	1111 011w	mod 111 r/m	l.b.disp	h.b.disp		
49.	AAD	1101 0101	0000 1010				
50.	NEG mem/reg						
	a) NEG reg	1111 011w	mod 011 r/m				
	b) NEG mem	1111 011w	mod 011 r/m	l.b.disp	h.b.disp		
51.	INC reg8/mem						
	a) INC reg8	1111 111w	mod 000 r/m				
	b) INC mem	1111 111w	mod 000 r/m	l.b.disp	h.b.disp		
52.	INC reg16	0100 0 reg					
53.	DEC reg8/mem						
	a) DEC reg8	1111 111w	mod 001 r/m				
	b) DEC mem	1111 111w	mod 001 r/m	l.b.disp	h.b.disp		
54.	DEC reg16	0100 1 reg					
55.	CBW	1001 1000					
56.	CWD	1001 1001					
57.	CMP reg2/mem, reg1/mem						
	a) CMP reg2, reg1	0011 10dw	mod reg r/m				
	b) CMP reg2, mem c) CMP mem, reg1	0011 10dw	mod reg r/m	l.b.disp	h.b.disp		
58.	CMP reg/mem, data						
	a) CMP reg, data	1000 00sw	mod 111 r/m	l.b.data	h.b.data		
	b) CMP mem, data	1000 00sw	mod 111 r/m	l.b.disp	h.b.disp	l.b.data	h.b.data
59.	CMP A, data	0001 110w	l.b.data	h.b.data			
	a) CMP AL, data8						
	b) CMP AX, data16						
Group - III Logical Instructions							
60.	AND reg2/mem, reg1/mem						
	a) AND reg2, reg1	0010 00dw	mod reg r/m				
	b) AND reg2, mem c) AND mem, reg1	0010 00dw	mod reg r/m	l.b.disp	h.b.disp		

Appendix I continued ...

S.No.	Mnemonic	Templates					
61.	AND reg/mem, data a) AND reg, data	1000 000w	mod 100 r/m	l.b.data	h.b.data		
	b) AND mem, data	1000 000w	mod 100 r/m	l.b.disp	h.b.disp	l.b.data	h.b.data
62.	AND A, data a) AND AL, data8 b) AND AX, data16	0010 010w	l.b.data	h.b.data			
63.	OR reg2/mem, reg1/mem a) OR reg2, reg1	0000 10dw	mod reg r/m				
	b) OR reg2, mem c) OR mem, reg1	0000 10dw	mod reg r/m	l.b.disp	h.b.disp		
64.	OR reg/mem, data a) OR reg, data	1000 000w	mod 001 r/m	l.b.data	h.b.data		
	b) OR mem, data	1000 000w	mod 001 r/m	l.b.disp	h.b.disp	l.b.data	h.b.data
65.	OR A, data a) OR AL, data8 b) OR AX, data16	0000 110w	l.b.data	h.b.data			
66.	XOR reg2/mem, reg1/mem a) XOR reg2, reg1	0011 00dw	mod reg r/m				
	b) XOR reg2, mem c) XOR mem, reg1	0011 00dw	mod reg r/m	l.b.disp	h.b.disp		
67.	XOR reg/mem, data a) XOR reg, data	1000 000w	mod 110 r/m	l.b.data	h.b.data		
	b) XOR mem, data	1000 000w	mod 110 r/m	l.b.disp	h.b.disp	l.b.data	h.b.data
68.	XOR A, data a) XOR AL, data8 b) XOR AX, data16	0011 010w	l.b.data	h.b.data			
69.	TEST reg2/mem, reg1/mem a) TEST reg2, reg1	1000 010w	mod reg r/m				
	b) TEST reg2, mem c) TEST mem, reg1	1000 010w	mod reg r/m	l.b.disp	h.b.disp		
70.	TEST reg/mem, data a) TEST reg, data	1111 011w	mod 000 r/m	l.b.data	h.b.data		
	b) TEST mem, data	1111 011w	mod 000 r/m	l.b.disp	h.b.disp	l.b.data	h.b.data
71.	TEST A, data a) TEST AL, data8 b) TEST AX, data16	1010 100w	l.b.data	h.b.data			

Appendix I continued ...

S.No.	Mnemonic	Templates			
72.	NOT reg/mem a) NOT reg	1111 011w	mod 010 r/m		
	b) NOT mem	1111 011w	mod 010 r/m	l.b.disp	h.b.disp
73.	SHL reg/mem or SAL reg/mem a) SHL reg or SAL reg i) SHL reg, 1 or SAL reg, 1 ii) SHL reg, CL or SAL reg, CL	1101 00vw	mod 100 r/m		
	b) SHL mem or SAL mem i) SHL mem, 1 or SAL mem, 1 ii) SHL mem, CL or SAL mem, CL	1101 00vw	mod 100 r/m	l.b.disp	h.b.disp
74.	SHR reg/mem a) SHR reg i) SHR reg, 1 ii) SHR reg, CL	1101 00vw	mod 101 r/m		
	b) SHR mem i) SHR mem,1 ii) SHR mem,CL	1101 00vw	mod 101 r/m	l.b.disp	h.b.disp
75.	SAR reg/mem a) SAR reg i) SAR reg, 1 ii) SAR reg, CL	1101 00vw	mod 111 r/m		
	b) SAR mem i) SAR mem, 1 ii) SAR mem, CL	1101 00vw	mod 111 r/m	l.b.disp	h.b.disp
76.	ROL reg/mem a) ROL reg i) ROL reg, 1 ii) ROL reg, CL	1101 00vw	mod 000 r/m		
	b) ROL mem i) ROL mem, 1 ii) ROL mem, CL	1101 00vw	mod 000 r/m	l.b.disp	h.b.disp
77.	RCL reg/mem a) RCL reg i) RCL reg, 1 ii) RCL reg, CL	1101 00vw	mod 010 r/m		
	b) RCL mem i) RCL mem, 1 ii) RCL mem, CL	1101 00vw	mod 010 r/m	l.b.disp	h.b.disp

Appendix I continued ...

S.No.	Mnemonic	Templates			
78.	ROR reg/mem a) ROR reg i) ROR reg, 1 ii) ROR reg, CL	1101 00vw	mod 001 r/m		
	b) ROR mem i) ROR mem, 1 ii) ROR mem, CL	1101 00vw	mod 001 r/m	l.b.disp	h.b.disp
79.	RCR reg/mem a) RCR reg i) RCR reg, 1 ii) RCR reg, CL	1101 00vw	mod 011 r/m		
	b) RCR mem i) RCR mem, 1 ii) RCR mem, CL	1101 00vw	mod 011 r/m	l.b.disp	h.b.disp
Group - IV String Manipulation Instructions					
80.	REP a) REPZ/REPE b) REPNZ/REPNE	1111 001z			
81.	MOVS a) MOVSB b) MOVSW	1010 010w			
82.	CMPS a) CMPSB b) CMPSW	1010 011w			
83.	SCAS a) SCASB b) SCASW	1010 111w			
84.	LODS a) LODSB b) LODSW	1010 110w			
85.	STOS a) STOSB b) STOSW	1010 101w			
Group - V Control Transfer Instructions					
86.	CALL disp16 (Call near - direct within segment)	1110 1000	l.b.disp	h.b.disp	
87.	CALL reg/mem (Call near - indirect within segment) a) CALL reg	1111 1111	mod 010 r/m		
	b) CALL mem	1111 1111	mod 010 r/m	l.b.disp	h.b.disp

Appendix I continued ...

S.No.	Mnemonic	Templates				
88.	CALL addr$_{offset}$, addr$_{base}$ (Call far-direct intersegment)	1001 1010	l.b.offset	h.b.offset	l.b.base	h.b.base
89.	CALL mem (Call far-indirect intersegment)	1111 1111	mod 011 r/m	l.b.disp	h.b.disp	
90.	RET (Return from call within segment)	1100 0011				
91.	RET data16 (Return from call within segment adding immediate data to SP)	1100 0010	l.b. data	h.b.data		
92.	RET (Return from intersegment call)	1100 1011				
93.	RET data16 (Return from intersegment call adding immediate data to SP)	1100 1010	l.b. data	h.b.data		
94.	JMP disp16 (Unconditional jump near-direct within segment)	1110 1001	l.b. disp	h.b.disp		
95.	JMP disp8 (Unconditional jump short-direct within segment)	1110 1011	disp8			
96.	JMP reg/mem (Unconditional jump near-indirect within segment)					
	a) JMP reg	1111 1111	mod 100 r/m			
	b) JMP mem	1111 1111	mod 100 r/m	l.b.disp	h.b.disp	
97.	JMP addr$_{offset}$, addr$_{base}$ (Unconditional jump far-direct intersegment)	1110 1010	l.b.offset	h.b.offset	l.b.base	h.b.base
98.	JMP mem (Unconditional jump far-indirect intersegment)	1111 1111	mod 101 r/m	l.b.disp	h.b.disp	
99.	JE/JZ disp8	0111 0100	disp8			
100.	JL/JNGE disp8	0111 1100	disp8			
101.	JLE/JNG disp8	0111 1110	disp8			
102.	JB/JNAE/JC disp8	0111 0010	disp8			
103.	JBE/JNA disp8	0111 0110	disp8			
104.	JP/JPE disp8	0111 1010	disp8			
105.	JNB/JAE/JNC disp8	0111 0011	disp8			
106.	JNBE/JA disp8	0111 0111	disp8			
107.	JNP/JPO disp8	0111 1011	disp8			

Appendix I continued ...

S.No.	Mnemonic	Templates			
108.	JNO disp8	0111 0001	disp8		
109.	JNS disp8	0111 1001	disp8		
110.	JO disp8	0111 0000	disp8		
111.	JS disp8	0111 1000	disp8		
112.	JNE/JNZ disp8	0111 0101	disp8		
113.	JNL/JGE disp8	0111 1101	disp8		
114.	JNLE/JG disp8	0111 1111	disp8		
115.	JCXZ disp8	1110 0011	disp8		
116.	LOOP disp8	1110 0010	disp8		
117.	LOOPZ/LOOPE disp8	1110 0001	disp8		
118.	LOOPNZ/LOOPNE disp8	1110 0000	disp8		
119.	INT type	1100 1101	type		
120.	INT 3	1100 1100			
121.	INTO	1100 1110			
122.	IRET	1100 1111			
Group - VI Processor Control Instructions					
123.	CLC	1111 1000			
124.	CMC	1111 0101			
125.	STC	1111 1001			
126.	CLD	1111 1100			
127.	STD	1111 1101			
128.	CLI	1111 1010			
129.	STI	1111 1011			
130.	HLT	1111 0100			
131.	WAIT	1001 1011			
132.	ESC opcode, mem/reg				
	a) ESC opcode, mem	1101 1opc	mod opc r/m	l.b.disp	h.b.disp
	b) ESC opcode, reg	1101 1opc	mod opc r/m		
133.	LOCK	1111 0000			
134.	NOP	1001 0000			
135.	Segment override prefix	001 sr 110			

TABLE - A - 1 : ONE BIT SPECIAL INDICATOR

Special bit value	Meaning
w = 0	8-bit operation.
w = 1	16-bit operation.
d = 0	The register specified by reg field is source operand.
d = 1	The register specified by reg field is destination operand.
sw = 00	8-bit operation with an 8-bit immediate data.
sw = 01	16-bit operation with a 16-bit immediate data.
sw = 11	16-bit operation with a sign extended 8-bit immediate operand.
v = 0	Shift/rotate operation is performed one time.
v = 1	The content of CL is count value for number of shift/rotate operations to be performed.
z = 0	Repeat execution of string instruction until ZF = 0.
z = 0	Repeat execution of string instruction until ZF = 1.

TABLE - A - 2 : CODES FOR "mod" FIELD

Code for mod field	Name of the mode
00	Memory mode with no displacement
01	Memory mode with 8-bit displacement
10	Memory mode with 16-bit displacement
11	Register mode

TABLE - A - 3 : CODES FOR "reg" FIELD

Code for reg field	Name of the register represented by the code when w = 0 or 1	
	When w = 0	When w = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

TABLE - A - 4 : CODES FOR "sr" FIELD

Code for sr field	Segment register
00	ES
01	CS
10	SS
11	DS

TABLE - A - 5 : CODES FOR "r/m" FIELD

Code for r/m field	Effective address calculation when mod = 00/01/10		
	mod = 00	mod = 01	mod = 10
000	[BX + SI]	[BX + SI + disp8]	[BX + SI + disp16]
001	[BX + DI]	[BX + DI + disp8]	[BX + DI + disp16]
010	[BP + SI]	[BP + SI + disp8]	[BP + SI + disp16]
011	[BP + DI]	[BP + DI + disp8]	[BP + DI + disp16]
100	[SI]	[SI + disp8]	[SI + disp16]
101	[DI]	[DI + disp8]	[DI + disp16]
110	[disp16]	[BP + disp8]	[BP + disp16]
111	[BX]	[BX + disp8]	[BX + disp16]

TABLE - A - 6 : MEANINGS OF VARIOUS TERMS USED IN THE OPERAND FIELD OF INSTRUCTIONS AND IN TEMPLATES

Term	Meaning
reg/reg1/reg2	8-bit or 16-bit register
reg8	8-bit register
reg16	16-bit register
segreg, sr	segment register
mem	8-bit or 16-bit memory
mem8	8-bit memory
mem16	16-bit memory
data	8-bit or 16-bit data

Term	Meaning
data8	8-bit data
data16	16-bit data
addr8	8-bit address
$addr_{offset}$	16-bit offset/effective address
$addr_{base}$	16-bit base address
disp8	8-bit signed displacement
disp16	16-bit displacement
opc	opcode

APPENDIX II : DOS AND BIOS INTERRUPTS

TABLE - 1 : DOS INTERRUPTS

Interrupt number	Function code	Dedicated operation
INT 20H	-	Program Terminate
	Character Input/Output	
INT 21H	01H	Read character from standard input device.
INT 21H	02H	Write character to standard output device.
INT 21H	03H	Read character from auxiliary input device.
INT 21H	04H	Write character to auxiliary output device.
INT 21H	05H	Write character to printer.
INT 21H	06H	Console input or output.
INT 21H	07H	Unfiltered character input without echo.
INT 21H	08H	Read character without echo.
INT 21H	09H	Display string.
INT 21H	0AH	Buffered string input.
INT 21H	0BH	Get input status.
INT 21H	0CH	Clear input buffer and read.
	File Operations	
INT 21H	0FH	Open file using FCB (**F**ile **C**ontrol **B**lock).
INT 21H	10H	Close file using FCB.
INT 21H	11H	Find first matching file using FCB.
INT 21H	12H	Find next matching file using FCB.
INT 21H	13H	Delete file using FCB.
INT 21H	16H	Create/Truncate file using FCB.
INT 21H	17H	Rename file using FCB.
INT 21H	23H	Get file size in records using FCB.
INT 21H	29H	Parse file name.
INT 21H	3CH	Create/Truncate file.
INT 21H	3DH	Open file.
INT 21H	3EH	Close file.
INT 21H	41H	Delete file.
INT 21H	43H	Set/Get file attributes - CHMOD.
INT 21H	45H	Duplicate a file handle.
INT 21H	46H	Force duplication of handle.
INT 21H	4EH	DOS first find (matching file).
INT 21H	4FH	DOS next find (matching file).
INT 21H	56H	Rename file.
INT 21H	57H	Get/Set file time and date.
INT 21H	5AH	Create unique temporary file.
INT 21H	5BH	Create new file.
	Record Operations	
INT 21H	14H	Read file sequentially using FCB.
INT 21H	15H	Write file sequentially using FCB.
INT 21H	1AH	Set disk transfer area address.

Appendix II continued...

Interrupt number	Function code	Dedicated operation
INT 21H	21H	Random record read using FCB.
INT 21H	22H	Random file read/write using FCB.
INT 21H	24H	Set random record number.
INT 21H	27H	Read random file block.
INT 21H	28H	Write random file block.
INT 21H	2FH	Get current disk transfer area address.
INT 21H	3FH	Read file or device.
INT 21H	40H	Write to file or device.
INT 21H	42H	Move file pointer.
INT 21H	5CH	Lock/unlock file access.
	Directory Operations	
INT 21H	39H	Create new subdirectory.
INT 21H	3AH	Delete subdirectory.
INT 21H	3BH	Set current directory.
INT 21H	47H	Get present working directory.
	Disk Management	
INT 21H	0DH	Reset disk.
INT 21H	0EH	Set DOS default disk.
INT 21H	19H	Get DOS default disk drive.
INT 21H	1BH	Get FAT information for default drive.
INT 21H	1CH	Get FAT information for specified drive.
INT 21H	2EH	Set or reset verify flag.
INT 21H	36H	Get disk free space.
INT 21H	54H	Get DOS verify switch.
	Process Management	
INT 21H	00H	Program terminate.
INT 21H	26H	Create program segment prefix.
INT 21H	31H	Terminate and stay resident.
INT 21H	4BH	Execute or load program.
INT 21H	4CH	Terminate with return code.
INT 21H	4DH	Get return code.
INT 21H	62H	Get PSP (Program Segment Prefix) pointer.
	Memory Management	
INT 21H	48H	Allocate memory.
INT 21H	49H	Release memory.
INT 21H	4AH	Modify memory allocation.
INT 21H	58H	Get or set allocation strategy.
	Network Functions	
INT 21H	5EH	Get machine name and Get/Set printer setup.
INT 21H	5FH	Get redirection entry.
	Time and Date Functions	
INT 21H	2AH	Get DOS system date.
INT 21H	2BH	Set DOS system date.
INT 21H	2CH	Get DOS system time.
INT 21H	2DH	Set DOS system time.

Appendix II continued...

Interrupt number	Function code	Dedicated operation
	Miscellaneous System Functions	
INT 21H	25H	Set interrupt vector.
INT 21H	30H	Get DOS version number.
INT 21H	32H	Get DOS disk information.
INT 21H	33H	Get/set Ctrl-Break flag, Get boot drive.
INT 21H	34H	DOS re-entrance status address.
INT 21H	35H	Get interrupt vector.
INT 21H	37H	Set/Get switch character.
INT 21H	38H	Get/Set country dependent information.
INT 21H	44H	Device I/O control.
INT 21H	59H	Get extended error information.
INT 21H	63H	Get load byte table.
INT 22H	-	Terminate handler pointer.
INT 23H	-	Ctrl-C handler pointer.
INT 24H	-	Critical error handler pointer.
INT 25H	-	Absolute disk read.
INT 26H	-	Absolute disk write.
INT 27H	-	Terminate and stay resident.
INT 28H to INT 2EH	-	Reserved.
INT 2FH	01H	Print spooler.
INT 2FH	10H	Share.

TABLE - 2 : BIOS INTERRUPTS

Interrupt number	Function code	Dedicated operation
	BIOS Video Driver Services	
INT 10H	00H	Set video mode.
INT 10H	01H	Set cursor shape.
INT 10H	02H	Set cursor position.
INT 10H	03H	Read cursor position.
INT 10H	04H	Read light pen position.
INT 10H	05H	Set active video page.
INT 10H	06H	Scroll/Initialize rectangle window up.
INT 10H	07H	Scroll/Initialize rectangle window down.
INT 10H	08H	Read character and attribute at cursor.
INT 10H	09H	Write character and attribute at cursor.
INT 10H	0AH	Write character only at cursor.
INT 10H	0BH	Set colour palette.
INT 10H	0CH	Set pixel.
INT 10H	0DH	Get pixel.
INT 10H	0EH	Write text in teletype mode.
INT 10H	0FH	Get video mode.
INT 10H	10H	Set colour palette registers.

Appendix VII continued...

Interrupt number	Function code	Dedicated operation
INT 10H	13H	Display string.
INT 10H	0FEH	Get video buffer pointer.
INT 10H	0FFH	Update video buffer.
INT 11H	-	Get machine configuration.
INT 12H	-	Get conventional memory size.
	BIOS Floppy Disk Services	
INT 13H	00H	Reset disk system.
INT 13H	01H	Get disk system status.
INT 13H	02H	Read disk sector.
INT 13H	03H	Write disk sector.
INT 13H	04H	Verify disk sectors.
INT 13H	05H	Format disk track.
	BIOS Serial Communication Port Services	
INT 14H	00H	Initialize communication port.
INT 14H	01H	Write communication port.
INT 14H	02H	Read communication port.
INT 14H	03H	Read communication port status.
INT 15H	-	AT services.
	BIOS Keyboard Driver Services	
INT 16H	00H	Read keyboard character.
INT 16H	01H	Read keyboard status.
INT 16H	02H	Read keyboard flags.
	BIOS Printer Driver Services	
INT 17H	00H	Write to printer.
INT 17H	01H	Initialize printer port.
INT 17H	02H	Read printer status.

APPENDIX III : LIST OF MICROPROCESSORS RELEASED BY INTEL

MICROPROCESSOR	DATE OF INTRODUCTION	NUMBER OF TRANSISTORS	CLOCK SPEED
4004	15th 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	8th 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	17th Oct, 1985	275,000	16/20/25/33 MHz
INTEL386 SX	16th Jun, 1988	275,000	16/20/25/33 MHz
INTEL386 SL	15th Oct, 1990	855,000	20/25 MHz
INTEL486 DX	10th Apr, 1989	1.2 million	25/33/50 MHz
INTEL486 SX	16th Sep, 1991	900,000	16/20/25 MHz
INTEL486 SX	21st Sep, 1992	1.185 million	33 MHz
INTEL486 SL	4th Nov, 1992	1.4 million	20/25/33 MHz
INTELDX 2	3rd Mar, 1992	1.2 million	50/66 MHz
INTELDX 4	7th Mar, 1994	3.2 million	75/100 MHz
Pentium	22nd Mar, 1993	3.1 million	60/66 MHz
Pentium	7th Mar, 1994	3.2 million	75/90/100/120 MHz
Pentium	Jun, 1995	3.3 million	133/150/166/200 MHz
Pentium Pro	1st Nov, 1995	5.5 million	150/166/180/200 MHz
Pentium (MMX)	8th Jan, 1997	4.5 million	166/200/233 MHz
Mobile Pentium (MMX)	9th Sep, 1997	4.5 million	200/233/266/300 MHz
Pentium II	7th May, 1997	7.5 million 450 MHz	233/266/300/333/350/400/
Mobile Pentium II	2nd Apr, 1998	7.5 million	233/266/300 MHz
Mobile Pentium II	25th Jan, 1999	27.4 million	333/366/400 MHz
Pentium II Xeon	29th Jun, 1998	7.5 million	400/450 MHz
Celeron	15th Apr, 1998	7.5 million	266/300 MHz
Celeron	24th Aug, 1998	19 million	333 MHZ to 2.7 GHz
Mobile Celeron	25th Jan, 1999	18.9 million	266 MHz to 2.4 GHz
Pentium III	26th Feb, 1999	9.5 million	450/500/550/600 MHZ

Appendix Ill continued...

MICROPROCESSOR	DATE OF INTRODUCTION	NUMBER OF TRANSISTORS	CLOCK SPEED
Pentium III	25th Oct, 1999	28 million	500 MHz to 1 GHz
Pentium III Xeon	17th Mar, 1999	9.5 million	500/550 MHz
Pentium III Xeon	25th Oct, 1999	28 million	600 to 900 MHz
Mobile Pentium III	25th Oct, 1999	28 million	400 MHz to 1 GHz
Mobile Pentium III	30th Jul, 2001	44 million	1/1.06/1.13/1.2/1.33 GHz
Pentium 4	20th Nov, 2000	42 million	1.4/1.5/1.6/1.7/1.8/1.9/2 GHz
Pentium 4	27th Aug, 2001	55 million	2 to 2.8 GHz
Pentium 4 (HT Technology)	14th Nov, 2002	55 million	2.4 to 3.3 GHz
Mobile Pentium 4	4th Mar, 2002	55 million	1.5 to 3.2 GHz
INTEL Xeon	21st May, 2001	42 million	1.4/1.5/1.7/2 GHz
INTEL Xeon	9th Jan, 2002	52 million	1.8/2/2.2/2.4/2.6/2.8 GHz
INTEL Xeon	18th Nov, 2002	108 million	1.4 to 3.2 GHz
INTEL Itanium	May, 2001	25 million	733/800 MHz
INTEL Itanium 2	8th Jul, 2002	220 million	900 MHz/1 GHz
INTEL Itanium 2	30th Jun, 2003	410 million	1/1.4/1.5 GHz
INTEL Pentium-M	12th 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.

GENERAL INDEX

CHIP INDEX