

**M. Sc. (IT) Previous Year**

**MIT-02**

**COMPUTER ORGANIZATION AND  
ARCHITECTURE**



**मध्यप्रदेश भोज (मुक्त) विश्वविद्यालय – भोपाल**

**MADHYA PRADESH BHOJ (OPEN) UNIVERSITY - BHOPAL**

### **Reviewer Committee**

1. Dr. Amit Kumar Mandle  
Assistant Professor  
IEHE, Bhopal
2. Dr. Romsha Sharma  
Professor  
Shri Sathya Sai College for Women, Bhopal
3. Dr. K. Mani Kandan Nair  
Department of Computer Science  
Makhanlal Chaturvedi National University of  
Journalism & Communication, Bhopal

### **Advisory Committee**

1. Dr. Jayant Sonwalkar  
Hon'ble Vice Chancellor  
Madhya Pradesh Bhoj (Open) University, Bhopal
2. Dr. L.S. Solanki  
Registrar  
Madhya Pradesh Bhoj (Open) University, Bhopal
3. Dr. Kishor John  
Director  
Madhya Pradesh Bhoj (Open) University, Bhopal
4. Dr. Amit Kumar Mandle  
Assistant Professor  
IEHE, Bhopal
5. Dr. Romsha Sharma  
Professor  
Shri Sathya Sai College for Women, Bhopal
6. Dr. K. Mani Kandan Nair  
Department of Computer Science  
Makhanlal Chaturvedi National University of  
Journalism & Communication, Bhopal

### **COURSE WRITERS**

**B. Basavaraj**, HOD, Department of Electronics & Communication Engineering, SJR College of Science, Arts and Commerce, Bangalore

**Units:** (1.0-1.2, 1.6, 1.8-1.9, 1.11-1.16, 2.5.2-2.5.8)

**Vivek Kesari**, Assistant Professor, Galgotia's GIMT, Institute of Management & Technology, Greater Noida, NOIDA

**Units:** (2.3, 2.5-2.5.1, 4.3-4.5, 4.6, 4.9.3, 4.10, 4.11)

**Sanjay Saxena**, Renowned Author of over 130 Books on Information Technology, New Delhi

**Units:** (2.6, 2.7.2, 2.10-2.10.2, 2.10.5, 3.0-3.2)

**Dr Deepti Mehrotra**, Professor & Head, Accreditation, Ranking & Quality Assurance, AMITY School of Engineering and Technology, AMITY University, NOIDA

**Units:** (1.3-1.5, 1.7, 1.10, 2.0-2.2, 2.4, 2.7-2.7.1, 2.8-2.9, 2.10.3-2.10.4, 2.11-2.15, 3.2.1-3.2.5, 3.3, 3.3.1-3.3.2, 3.3.3-3.8, 4.0-4.2, 4.5.1, 4.7-4.7.1, 4.8.1-4.8.7, 4.9-4.9.2, 4.9.4-4.9.7, 4.10.1, 5.2.3-5.2.6, 5.3, 5.5-5.10)

**Vikas Singhal**, Senior Lecturer, JSS Academy of Technical Education (JSSATE), NOIDA

**Dr Deepti Mehrotra**, Professor & Head, Accreditation, Ranking & Quality Assurance, AMITY School of Engineering and Technology, AMITY University, NOIDA

**Units:** (4.7.2, 4.8, 4.12-4.16, 5.4)

**Vikas Singhal**, Senior Lecturer, JSS Academy of Technical Education (JSSATE), NOIDA

**Units:** (5.0-5.2.2)

Copyright © Reserved, Madhya Pradesh Bhoj (Open) University, Bhopal

All rights reserved. No part of this publication which is material protected by this copyright notice may be reproduced or transmitted or utilized or stored in any form or by any means now known or hereinafter invented, electronic, digital or mechanical, including photocopying, scanning, recording or by any information storage or retrieval system, without prior written permission from the Registrar, Madhya Pradesh Bhoj (Open) University, Bhopal

Information contained in this book has been published by VIKAS® Publishing House Pvt. Ltd. and has been obtained by its Authors from sources believed to be reliable and are correct to the best of their knowledge. However, the Madhya Pradesh Bhoj (Open) University, Bhopal, Publisher and its Authors shall in no event be liable for any errors, omissions or damages arising out of use of this information and specifically disclaim any implied warranties or merchantability or fitness for any particular use.

Published by Registrar, MP Bhoj (open) University, Bhopal in 2020



VIKAS®

VIKAS® is the registered trademark of Vikas® Publishing House Pvt. Ltd.

VIKAS® PUBLISHING HOUSE PVT. LTD.

E-28, Sector-8, Noida - 201301 (UP)

Phone: 0120-4078900 • Fax: 0120-4078999

Regd. Office: A-27, 2nd Floor, Mohan Co-operative Industrial Estate, New Delhi 1100 44

• Website: www.vikaspublishing.com • Email: helpline@vikaspublishing.com

---

# SYLLABI-BOOK MAPPING TABLE

## Computer Organization and Architecture

---

Syllabi	Mapping in Book
<p><b>Unit I</b> <b>Information Representation:</b> Number System, Floating Point Representation, Integer Representation, Character Codes. <b>Logic Gates:</b> Boolean Algebra, Boolean Expression Simplification, <b>Basic Building Blocks and Circuits</b>, Combinational Circuits, Arithmetic Circuits, Combinational Circuits and Sequential Circuits, Registers, Counters.</p>	<p><b>Unit-1:</b> Information Representation, Logic Gates, Boolean Algebra, Circuits, Registers and Counters <b>(Pages 3-100)</b></p>
<p><b>Unit - II</b> <b>Register Manipulations and Micro-Operations:</b> Register Transfer, Bus System, Micro-Operations. <b>Computer Organization and Design Concepts:</b> Instruction and Instruction Code, Computer Instructions, Timing and Controls, Instruction Cycle, Memory Reference Instructions, Input/Output and Interrupts, Complete Computer Description, Machine Language, Design Control Unit.</p>	<p><b>Unit-2:</b> Register, Micro-Operations and Design Concepts <b>(Pages 101-159)</b></p>
<p><b>Unit - III</b> <b>Basic Computer Programming:</b> Assembly Language, The Assembler, Program Loops, Programming Arithmetic &amp; Logic, Subroutines, Input/Output Programming <b>Micro-Programming:</b> Micro-Programmed Control, Address Sequencing, Micro-Program Example, Design of Control Unit.</p>	<p><b>Unit-3:</b> Computer Programming and Micro-Programming <b>(Pages 161-217)</b></p>
<p><b>Unit - IV</b> <b>CPU Organization :</b> Central Processing Unit, General Register Organization, Stack Organization, Instruction Formats, Addressing Modes, Data Transfer and manipulation, Micro Programmed Control, Reduced Instruction Set Computer <b>Input-Output Organization:</b> Peripheral Devices, Input-Output Interface, Asynchronous Data Transfer, Mode of Transfer, Priority Interrupt, Direct Memory Access (DMA), Input-Output Processor (I-OP), Serial Communication. <b>Memory Organization:</b> Memory Unit, Types of Memory, Associative Memory, Building Large Memories, Cache Memory, Virtual Memory, Parallel Processing, Methods of Parallel Processing, Overcoming Pipelining Conflicts, Flynn's Classification, Array Processors.</p>	<p><b>Unit-4:</b> CPU, Input-Output and Memory Organizations <b>(Pages 219-342)</b></p>
<p><b>Unit - V</b> <b>Pipeline and Vector Processing :</b> Pipeline Processing, Vector Processing, Array Processing. <b>Multiprocessing :</b> Multiprocessors, Interconnection Structure, Interprocess Arbitration, Interprocessor communication and synchronization, Cache Coherence.</p>	<p><b>Unit-5:</b> Pipeline, Vector Processing and Multiprocessing <b>(Pages 343-422)</b></p>

---





---

# CONTENTS

---

<b>INTRODUCTION</b>	<b>1</b>
<b>UNIT 1 INFORMATION REPRESENTATION, LOGIC GATES, BOOLEAN ALGEBRA, CIRCUITS, REGISTERS AND COUNTERS</b>	<b>3-100</b>
1.0 Introduction	
1.1 Objectives	
1.2 Number System	
1.2.1 Decimal Number System	
1.2.2 Binary Number System	
1.3 Floating Point Representation	
1.3.1 Integer Representation	
1.3.2 1's Complement Representation	
1.3.3 2's Complement Representation	
1.3.4 Complements	
1.4 Character Codes	
1.5 Logic Gates	
1.5.1 NOT Gate	
1.5.2 AND Gate	
1.5.3 OR Gate	
1.5.4 XOR Gate	
1.6 Boolean Algebra	
1.6.1 Logical AND Operation	
1.6.2 Logical OR Operation	
1.6.3 Logical Complementation (Inversion)	
1.6.4 Basic Laws of Boolean Algebra	
1.6.5 De Morgan's Theorems	
1.6.6 Realization of Expression using Gates	
1.6.7 Combinational Logic	
1.7 Boolean Expression Simplification	
1.7.1 Algebraic Simplification	
1.7.2 Karnaugh Map	
1.7.3 Steps for Forming Karnaugh Map	
1.7.4 Simplification of Expressions using Karnaugh Map	
1.7.5 Simplification using Karnaugh Map	
1.7.6 Simplest SOP Expressions	
1.7.7 Getting POS Expressions	
1.8 Combinational Circuits	
1.8.1 Half-Adder	
1.8.2 Full-Adder	
1.8.3 Parallel Binary Adder	
1.8.4 Subtractors	
1.8.5 Decoders	
1.8.6 Multiplexer	
1.8.7 Code Converters	
1.9 Arithmetic Circuits	
1.9.1 Binary Addition	
1.9.2 Binary Subtraction	
1.9.3 Binary Multiplication	
1.9.4 Binary Division	
1.10 Combinational Circuits and Sequential Circuits	
1.10.1 Analysis of a Combinational Circuit	

- 1.11 Registers and Counters
  - 1.11.1 Serial-In-Serial-Out Shift Registers
  - 1.11.2 Serial-In-Parallel-Out Shift Registers
  - 1.11.3 Parallel-In-Serial-Out Shift Registers
  - 1.11.4 Parallel-In-Parallel-Out Registers
  - 1.11.5 Bidirectional Shift Registers
  - 1.11.6 Applications of Shift Registers
- 1.12 Answers to ‘Check Your Progress’
- 1.13 Summary
- 1.17 Key Terms
- 1.15 Self-Assessment Questions and Exercises
- 1.16 Further Reading

**UNIT 2 REGISTER, MICRO-OPERATIONS AND DESIGN CONCEPTS 101-159**

- 2.0 Introduction
- 2.1 Objectives
- 2.2 Register Transfer
- 2.3 Bus System
  - 2.3.1 Bus Organization
  - 2.3.2 Multiple Bus Organization
- 2.4 Micro-Operations
  - 2.4.1 Arithmetic Micro-Operations
  - 2.4.2 Logic Micro-Operations
  - 2.4.3 Shift Micro-Operations
- 2.5 Instruction and Instruction Code
  - 2.5.1 Instruction Execution
  - 2.5.2 Binary Coded Decimal (BCD) Code
  - 2.5.3 Excess-3 Code
  - 2.5.4 Gray Code
  - 2.2.5 Alphanumeric Codes
  - 2.5.6 Error-Detecting Codes
  - 2.5.7 Error-Correcting Codes
  - 2.5.8 Hamming Codes
- 2.6 Computer Instruction
  - 2.6.1 Instruction Representation
- 2.7 Timing and Controls
  - 2.7.1 Functions of Control Unit
  - 2.7.2 Instruction Cycle
- 2.8 Memory Reference Instructions
  - 2.8.1 Memory Reference Format
- 2.9 Input/Output and Interrupts
- 2.10 Complete Computer Description
  - 2.10.1 Basic Anatomy of the Computer
  - 2.10.2 Data Representation within the Computer
  - 2.10.3 Design of a Basic Computer
  - 2.10.4 Components of a Computer System
  - 2.10.5 Machine Language
- 2.11 Answers to ‘Check Your Progress’
- 2.12 Summary
- 2.13 Key Terms
- 2.14 Self-Assessment Questions and Exercises
- 2.15 Further Reading

**UNIT 3    COMPUTER PROGRAMMING AND MICRO-PROGRAMMING    161-217**

- 3.0 Introduction
- 3.1 Objectives
- 3.2 Assembly Language
  - 3.2.1 Assembler
  - 3.2.2 Program Loops
  - 3.2.3 Programming Arithmetic and Logic
  - 3.2.4 Subroutines
  - 3.2.5 Input/Output Programming
- 3.3 Micro-Programmed Control
  - 3.3.1 Address Sequencing
  - 3.3.2 Micro-Program Example
  - 3.3.3 Design of Control Unit
- 3.4 Answers to 'Check Your Progress'
- 3.5 Summary
- 3.6 Key Terms
- 3.7 Self-Assessment Questions and Exercises
- 3.8 Further Reading

**UNIT 4    CPU, INPUT-OUTPUT AND MEMORY ORGANIZATIONS    219-342**

- 4.0 Introduction
- 4.1 Objectives
- 4.2 Central Processing Unit
  - 4.2.1 Fundamental Concepts
  - 4.2.2 Organization of Registers in Different Computers
- 4.3 General Register Organization
  - 4.3.1 Control Word
- 4.4 Stack Organization
  - 4.4.1 Register Stack
  - 4.4.2 Memory Stack
  - 4.4.3 Reverse Polish Notation
  - 4.4.4 Evaluation of Arithmetic Expression
- 4.5 Instruction Formats
  - 4.5.1 Addressing Modes
- 4.6 Data Transfer and Manipulation
  - 4.6.1 Data Manipulation Instructions
- 4.7 Micro-Programmed Control
  - 4.7.1 Execution of Complete Instruction
  - 4.7.2 Reduced Instruction Set Computer
- 4.8 Peripheral Device
  - 4.8.1 Input-Output Interface
  - 4.8.2 Asynchronous Data Transfer
  - 4.8.3 Mode of Transfer
  - 4.8.4 Priority Interrupt
  - 4.8.5 Direct Memory Access (DMA)
  - 4.8.6 Input/Output Processor
  - 4.8.7 Serial Communication
- 4.9 Memory Unit
  - 4.9.1 Types of Memory
  - 4.9.2 Flash Memory
  - 4.9.3 Associative Memory
  - 4.9.4 Cache Memory
  - 4.9.5 Interleaving

- 4.9.6 Hit Rate and Miss Penalty
- 4.9.7 Virtual Memory
- 4.10 Parallel Processing
  - 4.10.1 Overcoming Pipelining Conflicts
- 4.11 Flynn's Classification
  - 4.11.1 Array Processors
- 4.12 Answers to 'Check Your Progress'
- 4.13 Summary
- 4.14 Key Terms
- 4.15 Self-Assessment Questions and Exercises
- 4.16 Further Reading

## **UNIT 5 PIPELINE, VECTOR PROCESSING AND MULTIPROCESSING 343-422**

- 5.0 Introduction
- 5.1 Objectives
- 5.2 Pipeline Processing
  - 5.2.1 Arithmetic Pipeline
  - 5.2.2 Instruction Pipeline
  - 5.2.3 Linear Pipeline
  - 5.2.4 RISC Pipelines
  - 5.2.5 Vector Processing
  - 5.2.6 Array Processing
- 5.3 Multiprocessors
  - 5.3.1 Interconnection Structure
  - 5.3.2 Characteristics of Multiprocessors
  - 5.3.3 Interprocess Arbitration
- 5.4 Interprocessor communication and Synchronization
  - 5.4.1 Racing Problem
  - 5.4.2 Problems of Critical Section
  - 5.4.3 Critical Section Algorithms
  - 5.4.4 Hardware Support for Mutual Exclusion
  - 5.4.5 Swap Instruction
  - 5.4.6 Binary Semaphore
  - 5.4.7 Implementation of Semaphores with a Waiting Queue
  - 5.4.8 Conditional Critical Region (CCR)
  - 5.4.9 Classical Problems in Concurrent Programming
  - 5.4.10 Readers and Writers Problem
  - 5.4.11 Deadlocks
  - 5.4.12 Resource Allocation Graph (RAG)
  - 5.4.13 Methods for Handling Deadlocks
  - 5.4.14 Introduction to File System and IO
  - 5.4.15 Organizing Files
- 5.5 Cache Coherence
- 5.6 Answers to 'Check Your Progress'
- 5.7 Summary
- 5.8 Key Terms
- 5.9 Self-Assessment Questions and Exercises
- 5.10 Further Reading

---

## INTRODUCTION

---

A computer is an electronic device for storing and processing data, typically in binary form, according to instructions given to it in a variable program. In computer engineering, computer architecture is a set of rules and methods that describe the functionality, organization, and implementation of computer systems. The architecture of a system refers to its structure in terms of separately specified components of that system and their interrelationships.

Computer organization helps optimize performance-based products, such as the processing power of processors. Computer organization also helps plan the selection of a processor for a particular project. Multimedia projects may need very rapid data access, while virtual machines may need fast interrupts. Sometimes certain tasks need additional components as well. Computer organization and features also affect power consumption and processor cost.

Fundamentally, the computer architecture involves Instruction Set Architecture (ISA) design, microarchitecture design, logic design, and its implementation.

The discipline of computer architecture includes three main subcategories, namely the Instruction Set Architecture (ISA), the microarchitecture and the systems design. The term ISA refers to the machine code that a processor reads and acts upon as well as the word size, memory address modes, processor registers, and data type. Microarchitecture is also known as ‘Computer Organization’, as it describes how a particular processor will implement the ISA. Systems design includes all of the other hardware components within a computing system, such as data processing other than the CPU, for example Direct Memory Access (DMA), virtualization, and multiprocessing.

Instruction sets have shifted over the years, from initially very simple to sometimes very complex in various respects. A Complex Instruction Set Computer or CISC is a computer where single instructions can execute several low level operations and/or are capable of multi-step operations or addressing modes within single instructions. The term was retroactively coined in contrast to Reduced Instruction Set Computer (RISC). However, the choice of instruction set architecture may greatly affect the complexity of implementing high performance devices. The prominent strategy, used to develop the first RISC processors, was to simplify instructions to a minimum of individual semantic complexity combined with high encoding regularity and simplicity.

This book is divided into five units that attempt to give the students a fair idea of basic computer organization and design, information representation, logic gates, basic building blocks and circuits, computer organization and design concepts, basic computer programming, micro-programming, CPU organization, input-output organization, memory organization, pipeline and vector processing, and multiprocessing. The book follows the Self-Instructional Mode or SIM format wherein each unit begins with an ‘Introduction’ to the topic followed by an outline of the ‘Objectives’. The detailed content is then presented in a simple and structured manner interspersed with Answers to ‘Check Your Progress’ questions. A list of ‘Key Terms’, a ‘Summary’ and a set of ‘Self-Assessment Questions and Exercises’ is also provided at the end of each unit for effective recapitulation.

## NOTES



---

# UNIT 1 INFORMATION REPRESENTATION, LOGIC GATES, BOOLEAN ALGEBRA, CIRCUITS, REGISTERS AND COUNTERS

---

**Information  
Representation Logic  
Gates, Boolean Algebra,  
Circuits, Registers and  
Counters**

NOTES

## Structure

- 1.0 Introduction
- 1.1 Objectives
- 1.2 Number System
  - 1.2.1 Decimal Number System
  - 1.2.2 Binary Number System
- 1.3 Floating Point Representation
  - 1.3.1 Integer Representation
  - 1.3.2 1's Complement Representation
  - 1.3.3 2's Complement Representation
  - 1.3.4 Complements
- 1.4 Character Codes
- 1.5 Logic Gates
  - 1.5.1 NOT Gate
  - 1.5.2 AND Gate
  - 1.5.3 OR Gate
  - 1.5.4 XOR Gate
- 1.6 Boolean Algebra
  - 1.6.1 Logical AND Operation
  - 1.6.2 Logical OR Operation
  - 1.6.3 Logical Complementation (Inversion)
  - 1.6.4 Basic Laws of Boolean Algebra
  - 1.6.5 De Morgan's Theorems
  - 1.6.6 Realization of Expression using Gates
  - 1.6.7 Combinational Logic
- 1.7 Boolean Expression Simplification
  - 1.7.1 Algebraic Simplification
  - 1.7.2 Karnaugh Map
  - 1.7.3 Steps for Forming Karnaugh Map
  - 1.7.4 Simplification of Expressions using Karnaugh Map
  - 1.7.5 Simplification using Karnaugh Map
  - 1.7.6 Simplest SOP Expressions
  - 1.7.7 Getting POS Expressions
- 1.8 Combinational Circuits
  - 1.8.1 Half-Adder
  - 1.8.2 Full-Adder
  - 1.8.3 Parallel Binary Adder
  - 1.8.4 Subtractors
  - 1.8.5 Decoders
  - 1.8.6 Multiplexer
  - 1.8.7 Code Converters

**NOTES**

- 1.9 Arithmetic Circuits
  - 1.9.1 Binary Addition
  - 1.9.2 Binary Subtraction
  - 1.9.3 Binary Multiplication
  - 1.9.4 Binary Division
- 1.10 Combinational Circuits and Sequential Circuits
  - 1.10.1 Analysis of a Combinational Circuit
- 1.11 Registers and Counters
  - 1.11.1 Serial-In-Serial-Out Shift Registers
  - 1.11.2 Serial-In-Parallel-Out Shift Registers
  - 1.11.3 Parallel-In-Serial-Out Shift Registers
  - 1.11.4 Parallel-In-Parallel-Out Registers
  - 1.11.5 Bidirectional Shift Registers
  - 1.11.6 Applications of Shift Registers
- 1.12 Answers to ‘Check Your Progress’
- 1.13 Summary
- 1.17 Key Terms
- 1.15 Self-Assessment Questions and Exercises
- 1.16 Further Reading

---

## **1.0 INTRODUCTION**

---

A number system that uses only two digits, 0 and 1, is called the binary number system. A binary fraction can be represented by a series of 1 and 0 to the right of a binary point. In digital computers, the binary numbers are represented by a set of binary storage devices such as flip flops. A binary number can be converted into decimal number by multiplying the binary 1 or 0 by the weight corresponding to its position and adding all the values.

Floating point representation has a fractional part and is known as the floating-point number. Floating point number are those numbers, which include ‘Decimals’ or ‘Fractional Parts’ and ‘Integer Values’. In integer representation the sign information has to be encoded along with the magnitude to represent the integers completely.

A logic gate is a device that performs Boolean logic on one or more binary inputs before producing a single binary output. Computers often output more than a single binary digit and conduct more than simple Boolean logic operations on supplied data and a combinational circuit is a digital logic circuit in which the output is determined by the combination of inputs at any given time, regardless of the condition of the inputs previously. Combinational circuits are built around the digital logic gate.

Arithmetic circuits in computers and calculators perform arithmetic and logical operations. All arithmetic operations take place in the arithmetic unit of a computer. A register is a group of flip-flops used to store or manipulated data or both. Each flip-flop is capable of storing one bit of information.

In this unit, you will study about the number system, floating point representation, integer representation, character codes, logic gates, Boolean algebra, Boolean expression simplification, combinational circuits, arithmetic circuits, combinational circuits and sequential circuits, registers and counters.



---

## 1.1 OBJECTIVES

---

After going through this unit, you will be able to:

- Discuss about the different types of number system
- Explain the floating point representation
- Analyse the techniques of integer representation
- Describe the various character codes of computer
- Elaborate on the different types of logic gates
- Discuss about the Boolean algebra
- Describe the simplification techniques of Boolean Expression
- Define combinational circuits
- Understand the significance of the arithmetic circuits
- Differentiate between combinational circuits and sequential circuits
- Explain the term registers
- Discuss the basic features of counters

## NOTES

---

## 1.2 NUMBER SYSTEM

---

A number is an idea that is used to refer amounts of things. People use number words, number gestures and number symbols. Number words are said out loud. Number gestures are made with some part of the body, usually the hands. Number symbols are marked or written down. A number symbol is called a **numeral**. The number is the idea we think of when we see the numeral or when we see or hear the word.

On hearing the word number, we immediately think of the familiar decimal number system with its 10 digits; 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. These numerals are called **Arabic numerals**. Our present number system provides modern mathematicians and scientists with great advantages over those of previous civilisations, and is an important factor in our advancement. Since fingers are the most convenient tools nature has provided, human beings use them in counting. So the decimal number system followed naturally from this usage.

A number of base, or radix, **r** is a system that uses distinct symbols of **r** digits. Numbers are represented by a string of digit symbols. To determine the quantity that the number represents, it is necessary to multiply each digit by an integer power of **r**, and then form the sum of all the weighted digits. It is possible to use any whole number greater than one as a base in building a numeration system. The number of digits used is always equal to the base.

There are four systems of arithmetic, which are often used in digital systems. These systems are:

**NOTES**

1. Decimal
2. Binary
3. Hexadecimal
4. Octal

In any number system, there is an ordered set of symbols known as **digits**. Collection of these digits makes a number which in general has two parts, integer and fractional, and are set apart by a radix point (.). Hence, a number system can be represented as

$$N_b = \underbrace{a_{n-1} a_{n-2} a_{n-3} \dots a_1 a_0}_{\text{Integer portion}} \cdot \underbrace{a_{-1} a_{-2} a_{-3} \dots a_{-m}}_{\text{Fractional portion}}$$

where **N** = a number

**b** = radix or base of the number system

**n** = number of digits in integer portion

**m** = number of digits in fractional portion

**a<sub>n-1</sub>** = Most Significant Digit (MSD)

**a<sub>-m</sub>** = Least Significant Digit (LSD)

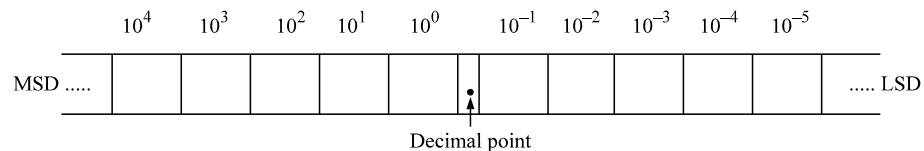
and  $0 \leq (a_i \text{ or } a_j) \leq b-1$

**Base or Radix:** The base or radix of a number is defined as the number of different digits which can occur in each position in the number system.

### 1.2.1 Decimal Number System

The number system which utilizes ten distinct digits, i.e., 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 is known as decimal number system. It represents numbers in terms of groups of tens (as shown in Figure 1.1).

We would be forced to stop at 9 or to invent more symbols if it were not for the use of positional notation. It is necessary to learn only 10 basic numbers and positional notational system in order to count any desired figure.



**Fig. 1.1 Decimal Position Values as Powers of 10**

The decimal number system has a base or radix of 10. Each of the ten decimal digits 0 through 9, has a place value or weight depending on its position. The weights are units, tens, hundreds and so on. The same can be written as the power of its base as [10<sup>0</sup>, 10<sup>1</sup>, 10<sup>2</sup>, 10<sup>3</sup>,...], etc. Thus, the number 1993 represents quantity equal to 1000 + 900 + 90 + 3. Actually, this should be written as {1 × 10<sup>3</sup> + 9 × 10<sup>2</sup> + 9 × 10<sup>1</sup> + 3 × 10<sup>0</sup>}. Hence, 1993 is the sum of all digits multiplied by their weights.

Each position has a value 10 times greater than the position to its right.

**Example 1.1:** The number 379 actually stands for the following representation:

**Solution:**

$$\begin{array}{r}
 100 \quad 10 \quad 1 \\
 10^2 \quad 10^1 \quad 10^0 \\
 3 \quad 7 \quad 9 \\
 3 \times 100 + 7 \times 10 + 9 \times 1 \\
 \therefore 379_{10} = 3 \times 100 + 7 \times 10 + 9 \times 1 \\
 = 3 \times 10^2 + 7 \times 10^1 + 9 \times 10^0
 \end{array}$$

In this example, 9 is the Least Significant Digit (LSD) and 3 is the Most Significant Digit (MSD).

**Example 1.2:** The number 1936.469 can be written as

**Solution:**

$$\begin{aligned}
 1936.469_{10} &= 1 \times 10^3 + 9 \times 10^2 + 3 \times 10^1 + 6 \times 10^0 + 4 \times 10^{-1} \\
 &\quad + 6 \times 10^{-2} + 9 \times 10^{-3} \\
 &= 1000 + 900 + 30 + 6 + 0.4 + 0.06 + 0.009 = \mathbf{1936.469}
 \end{aligned}$$

It is seen that powers are numbered to the left of the decimal point starting with 0 and to the right of decimal point starting with -1.

The general rule for representing numbers in the decimal system by using positional notation is as follows:

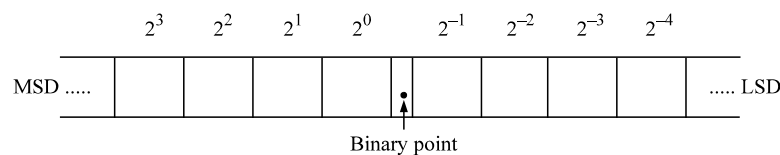
$$\mathbf{a_{n-1} a_{n-2} \dots a_2 a_1 a_0 = a_{n-1} 10^{n-1} + a_{n-2} 10^{n-2} + \dots + a_2 10^2 + a_1 10^1 + a_0 10^0}$$

where **n** is the number of digits to the left of the decimal point.

### 1.2.2 Binary Number System

A number system that uses only two digits, 0 and 1 is called the **binary number system**. The binary number system is also called a **base two system**. The two symbols 0 and 1 are known as **bits** (binary digits).

The binary system groups numbers by two's and by powers of two (as shown in Figure 1.2). The word binary comes from a Latin word meaning two at a time.



**Fig. 1.2 Binary Position Values as a Power of 2**

The weight or place value of each position can be expressed in terms of 2, and as  $2^0, 2^1, 2^2$ , etc. The least significant digit has a weight of  $2^0 (= 1)$ . The second position to the left of the least significant digit is multiplied by  $2^1 (= 2)$ . The third position has weight equal to  $2^2 (= 4)$ . Thus, the weights are in the ascending powers of 2 or 1, 2, 4, 8, 16, 32, 64, 128, etc.

The numeral  $10_2$  (one, zero, base two) stands for two, the base of the system.

In binary counting, single digits are used for none and one. Two-digit numbers are used for  $10_2$  and  $11_2$  [2 and 3 in decimal numerals]. For the next counting

**NOTES**

number,  $100_2$  (4 in decimal numerals) three digits are necessary. After  $111_2$  (7 in decimal numerals) four digit numerals are used until  $1111_2$  (15 in decimal numerals) is reached, and so on. In a binary numeral every position has a value 2 times the value of the position to its right.

A binary number with 4 bits, is called a **nibble** and binary number with 8 bits is known as a byte.

**Example 1.3:** The number  $1011_2$  actually stands for the following representation:

**Solution:** 
$$1011_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

$$= 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1$$

$$\therefore 1011_2 = 8 + 0 + 2 + 1 = 11_{10}$$

In general,

$$[b_n b_{n-1} \dots b_2 b_1 b_0]_2 = b_n 2^n + b_{n-1} 2^{n-1} + \dots + b_2 2^2 + b_1 2^1 + b_0 2^0$$

**Example 1.4:** The binary number 10101.011 can be written as

**Solution:** 
$$\begin{array}{ccccccc} 1 & 0 & 1 & 0 & 1 & . & 0 & 1 & 1 \\ 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & . & 2^{-1} & 2^{-2} & 2^{-3} \\ \text{(MSD)} & & & & & & \text{(LSD)} & & \end{array}$$

$$\therefore 10101.011_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$+ 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$$

$$= 16 + 0 + 4 + 0 + 1 + 0 + 0.25 + 0.125 = 21.375_{10}$$

In each binary digit, the value increases in powers of two starting with 0 to the left of the binary point and decreases to the right of the binary point starting with power -1.

**Why Binary Number System is Used**

Binary number system is used in digital computers because all electrical and electronic circuits can be made to respond to the two states concept. For instance: a switch can be either opened or closed, only two possible states exist. A transistor can be made to operate either in cut-off or saturation; a magnetic tape can be either magnetised or non-magnetised; a signal can be either HIGH or LOW; a punched tape can have a hole or no hole. In all of the above illustrations, each device is operated in any one of the two possible states and the intermediate condition does not exist. Thus, 0 can represent one of the states and 1 can represent the other. Hence, binary numbers are convenient to use in analysing or designing digital circuits.

**Binary Fraction**

A binary fraction can be represented by a series of 1 and 0 to the right of a binary point. The weights of digit positions to the right of the binary point are given by  $2^{-1}$ ,  $2^{-2}$ ,  $2^{-3}$  and so on.

**Example 1.5:** The binary fraction 0.1011 can be written as

**Solution:** 
$$0.1011 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4}$$

$$= 1 \times 0.5 + 0 \times 0.25 + 1 \times 0.125 + 1 \times 0.0625$$

$$0.1011_2 = 0.6875_{10}$$

### Mixed Numbers

Mixed numbers contain both integer and fractional parts. The weights of mixed numbers are

$$2^3 \quad 2^2 \quad 2^1 \quad \cdot \quad 2^{-1} \quad 2^{-2} \quad 2^{-3} \quad \text{etc.}$$

↑  
Binary Point

**Example 1.6:** A mixed binary numbers 1011.101 can be written as

**Solution:**  $1011.101_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$

$$= 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1 + 1 \times 0.5 + 0 \times 0.25 + 1 \times 0.125$$

$\therefore [1011.101]_2 = [11.625]_{10}$

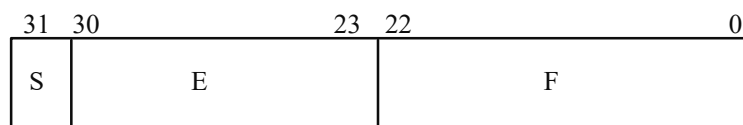
When different number systems are used, it is customary to enclose the number within big brackets and the subscripts indicate the type of the number system.

## 1.3 FLOATING POINT REPRESENTATION

In floating point representation integers can be represented on a computer. However, there is another type of number, which has a fractional part and is known as the floating-point number. Floating-point numbers are those numbers, which include ‘Decimals’ or ‘Fractional Parts’ and ‘Integer Values’, for example, representation of decimal 5.25 can be given as 101.01 in binary. The problem is how to show the decimal point since you are limited to using only 0 or 1 in a binary system. The most widely used presentation is the IEEE 754 floating-point presentation, which presents standards for both 32-bit and 64-bit floating-point numbers.

According to the IEEE standard, the leftmost bit represents the sign of the number. Sign bit 0 is used for the positive sign and 1 for the negative sign. Then the rest is divided into two parts, namely, exponent and mantissa. Since this presentation does not want to ‘Waste’ a sign bit for the exponent, it decides to use the lower half of the exponential value as negative and the upper half as positive. This will become clearer later with examples.

The 32-bit presentation in the IEEE floating-point standard will be as follows:



Here S = Sign bit, in the leftmost position; S = 1 for -ve number and 0 for +ve number.

E = The ‘Exponent’ in the next 8-bits.

F = The normalized fractional part in the last 23-bits.

**NOTES**

An example of this might look like 0 00101011 100000000000000000000000.

For converting the above to a decimal value, we have to use the following formula:

$$V = (-1)^S \times 2^{E-127} \times 1.F$$

It is to be noted that there is 127 in the exponent term. Why? Since E is made up of 8-bits, it ranges from 00000000 = 0 to 11111111 = 255 and  $\frac{1}{2}$  of 255 is 127.5 or simply 127. In this way, E = 127 will give the exponent 0; anything above 127 will result in a positive exponent and anything below 127 will result in a negative exponent. The above is known as **excess 127** presentations.

For example, if a 32-bit word contains the floating-point number 0 10000000 100000000000000000000000, this means S = 0, E = 128 and F = 0.5. Therefore, this is representing the decimal value,

$$V = (-1)^0 \times 2^{128-127} \times 1.5 = 1 \times 2^1 \times 1.5 = 3.0$$

For example,

Represent -2.5 in the IEEE754 32-bit floating-point standard.

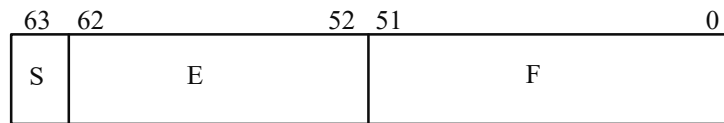
Since its negative number, S = 1. Then convert 2.5 to binary, which comes out to be 10.1. Normalize this to  $1.01 \times 2^1$ . To fit this into  $V = (-1)^S \times 2^{E-127} \times 1.F$ , E = 128 and F = 01000...0. Therefore, the representation of -2.5 is,

1
10000000  
 010000000000000000000000

For example,

Represent 0.875 in IEEE 32-bit presentation. Since this is a positive number, S = 0. Convert 0.875 to binary, which comes out to be 0.111 that normalizes to  $1.11 \times 2^{-1}$ . This means E = 126 (binary 01111110) and F = 1100000...0. Therefore, the binary representation of 0.875 is 0 01111110 110000000000000000000000.

The IEEE standard for 64-bit floating-point representation is similar to the 32-bit standard. The format is shown along with its conversion formula.



$$V = (-1)^S \times 2^{E-1023} \times 1.F$$

Since the integer value 1 is assumed to be invisibly stored with each number in both the 32- and 64-bit representations, you may wonder how the number 0.0 can be represented.

For this, there is an exception. Whenever **both** E and F are **all** 0s,  $V = 0.0$ .

That is 0 00000000 000000000000000000000000 represents 0.0.

How do you add or subtract floating-point values? Obviously, the adders you have learned earlier will not work here. This is similar to when you add or subtract numbers with decimal points where you have to first line up the decimals.

Likewise, to add or subtract floating-point numbers, you will have to first modify the F of one of the two numbers by shifting left or right to make it equal to the E of the other number. Then and only then can you add the two F's.

### 1.3.1 Integer Representation

The integers that were discussed so far are unsigned integers; signed integers have been ignored till now. However, the problem with signed integers is how to present a sign. The sign information has to be encoded along with the magnitude to represent the integers completely. The following are the three possible techniques for representing signed integers:

- Signed Magnitude Representation
- Diminished Radix-Complement Representation
- Radix-Complement Representation

#### Signed Magnitude Representation

In the Signed Magnitude (SM) representation, the Most Significant Bit (MSB) is used to represent the sign as follows:

‘1’ is used for a ‘-’ (Negative Sign)

‘0’ is used for a ‘+’ (Positive Sign)

#### Format of SM Representation

SM number in 8-bits looks like smmmmmmm, where ‘s’ at the MSB location represents sign and the other 7-bits represent the magnitude.

*Note:* For positive numbers, this presentation is the same as the unsigned binary representation for any number.

#### Signed Magnitude Examples (8-Bits)

The following examples show the SM (Signed Magnitude) representation and their corresponding hex numbers:

$$\begin{aligned}
 -5 &= (10000101)_2 = (85)_{16} \\
 +5 &= (00000101)_2 = (05)_{16} \\
 +127 &= (01111111)_2 = (7F)_{16} \\
 -127 &= (11111111)_2 = (FF)_{16} \\
 +0 &= (00000000)_2 = (00)_{16} \\
 -0 &= (10000000)_2 = (80)_{16}
 \end{aligned}$$

With N-bits, you can represent the signed integers that range from,

$$-\{2^{(N-1)} - 1\} \text{ to } \{2^{(N-1)} - 1\}.$$

For example, with 8-bits, you can represent the signed integers ranging from -127 to +127. Signed magnitude is easy to understand and can be encoded easily and hence, finds its application in digital electronics.

## Problems with Signed Magnitude Representation

One problem with signed magnitude representation is that it has two ways of representing 0 (–0 and +0). Another problem is that addition of  $N + (-N)$  does not give result as zero. For example,

$$-5 + 5 = (85)_{16} + (05)_{16} = (8A)_{16} = -10$$

In order to remove this ambiguity, new representations are proposed. They are one's complement (1's) representation and two's complement (2's) representation for signed integers. These complements are used to represent only the signed integers and for the positive numbers, this representation is same as that of simple binary representation.

### 1.3.2 1's Complement Representation

1's complement is another way to represent signed integers. In this presentation, for the case of a negative number, first get the binary representation of its magnitude, and then complement each bit, i.e., replace 1 with 0 and vice versa.

**Example 1.7** Represent –5 in 1's complement in 8-bits.

**Solution:**

**Step 1:** First, find the magnitude 5 in 8-bits.

$$(00000101)_2 = (05)_{16}$$

**Step 2:** Then complement each bit as per rule.

$$(11111010)_2 = (FA)_{16}$$

Hence,  $(FA)_{16}$  is the 8-bit one's complement representation of –5.

### Examples

The following examples show 1's complement representation and their corresponding hex for some numbers:

$$\begin{aligned} -5 &= (11111010)_2 = (FA)_{16} \\ +5 &= (00000101)_2 = (05)_{16} \\ +127 &= (01111111)_2 = (7F)_{16} \\ -127 &= (10000000)_2 = (80)_{16} \\ +0 &= (00000000)_2 = (00)_{16} \\ -0 &= (11111111)_2 = (FF)_{16} \end{aligned}$$

In N-bits representation, you can represent the signed integers ranging from  $-\{2^{(N-1)} - 1\}$  to  $\{2^{(N-1)} - 1\}$ . In 8-bits, you can represent the signed integers from –127 to +127. However, the above representation has a problem in presenting 0, since there will be two ways of representing 0 like –0 and +0. However, addition of  $N + (-N)$  now gives the result as zero.

$$\begin{aligned} -5 + 5 &= (FA)_{16} + (05)_{16} \\ &= (FF)_{16} = -0 \end{aligned}$$

Some more observations for using 1's complement are given as follows:



For example,  $K + 0 = K$  will work if you use  $+0$  and it will not satisfy the result if you use  $-0$ , in place of 0.

$$5 + (+0) = (05)_{16} + (00)_{16} = (05)_{16} = 5 \text{ (Correct)}$$

$$5 + (-0) = (05)_{16} + (FF)_{16} = (04)_{16} = 4 \text{ (Wrong)}$$

So, the two's complement representation comes into the picture.

### 1.3.3 2's Complement Representation

2's complement is used to represent signed integers, especially negative integers. Whenever you need to encode a negative number, first get the binary representation of its magnitude, complement each bit and then add 1. For example, 2's complement of  $-5$  in 8-bits can be obtained as follows:

**Step 1:** The magnitude 5 in 8-bits

$$(00000101)_2 = (05)_{16}$$

**Step 2:** Taking complement of each bit

$$(11111010)_2 = (FA)_{16}$$

**Step 3:** Adding one will result 2's complement

$$11111011 = (FA)_{16} + 1 = (FB)_{16}$$

So,  $(11111011)_2$  in binary and  $(FB)_{16}$  in Hex is the 8-bit two's complement representation of  $-5$ .

The following examples show 2's complement representation and their corresponding Hex number.

$$-5 = (11111011) = (FB)_{16}$$

$$+5 = (00000101) = (05)_{16}$$

$$+127 = (01111111) = (7F)_{16}$$

$$-127 = (10000001) = (81)_{16}$$

$$-128 = (10000000) = (80)_{16}$$

$$+0 = (00000000) = (00)_{16}$$

$$-0 = (00000000) = (00)_{16}$$

In  $N$ -bits, you can represent the signed integers ranging from  $-2^{(N-1)}$  to  $2^{(N-1)} - 1$ . It is to be noted that the negative range extends one more than the positive range. For example, in 8-bits, you can represent the signed integers from  $-128$  to  $+127$ .

It is to be noted that the 2's complement representation of any signed number has none of the drawbacks like the signed magnitude or one's complement representation, since there is only one representation for zero, i.e., for  $+0$  and  $-0$ , and  $N + (-N) = 0$ .

$$\text{For example, } -5 + 5 = (FB)_{16} + (05)_{16} = (00)_{16} = 0$$

### 1.3.4 Complements

Complements are used in digital computers for simplifying the subtraction operation and for logic manipulations. There are two types of complements for each base  $R$  system, which are as follows:

## NOTES

- Diminished Radix-Complement Representation [(R – 1)’s Complement]
- Radix-Complement Representation [R’s Complement]

### (R – 1)’s Complement Representation

For positive numbers, you need to represent by sign and magnitude and for negative numbers – N, you need to represent it by  $\tilde{N}$ , the (R – 1)’s complement where,

$$\tilde{N} = (R^n - R^{-m}) - N$$

**n** = Total number of digits in integer part of the number N

**m** = Total number of digits in fractional part of the number N

For example,

$$9\text{'s complements of } (52520)_{10} = (10^5 - 10^0 - 52520) = (10^5 - 1 - 52520) = 47479$$

$$9\text{'s complements of } (0.3267)_{10} = (10^0 - 10^{-4} - 0.3267) = (0.9999 - 0.3267) = 0.6732$$

### R’s Complement Representation

For positive numbers, you need to represent by sign and magnitude and for negative numbers – N, you need to represent it by  $\tilde{N}$ , the (R)’s complement where

$$\tilde{N} = R^n - N$$

**n** = Total number of digits in the integer part of the number N

For example,

$$10\text{'s complements of } (52520)_{10} = (10^5 - 52520) = 47480$$

$$10\text{'s complements of } (0.3267)_{10} = (10^0 - 0.3267) = (1.0 - 0.3267) = 0.6733$$

---

## 1.4 CHARACTER CODES

---

Most of the processing in computers and other digital circuits are done in the binary formats. Various binary codes are used to represent data, which may be numerals, alphabets or special characters. A user must be very careful about the code being used while interpreting information available in the binary format. For example,

1000001 represents  $(65)_{10}$  in straight binary.

1000001 represents  $(41)_{10}$  in BCD.

1000001 represents A in ASCII code.

Some commonly used codes are as follows:

- Straight Binary Codes
- Natural BCD Codes
- Excess-3 Codes
- Gray Codes
- Alphanumeric Codes
- Error Codes

## 1. Straight Binary Codes

One binary digit (one bit) can take on values 0 and 1. This is used to represent numbers using a natural (straight) binary form as discussed earlier. For example,  $(65)_{10}$  in straight binary is represented by 1000001. Examples of other binary representations are: {Black = 0, White = 1}, {True = 1, False = 0}, {On = 1, Off = 0}.

Similarly, two binary digits (two bits) can represent four different values like 00, 01, 10 and 11.

In general, N-bits (or N binary digits) can represent  $2^N$  different values.

For example, 4-bits can represent  $2^4$  or 16 different values. N-bits can take on unsigned decimal values from 0 to  $2^N - 1$ .

## 2. Natural BCD Codes

BCD stands for Binary Coded Decimal. In BCD codes, decimal digits 0 through 9 are represented by their natural binary equivalents using four bits, and each decimal digit of a decimal number is represented by this four bits code individually. It is also known as 8, 4, 2, 1 codes where 8, 4, 2, 1 are the weights of the four bits of the decimal digits similar to the straight binary number system. The given Table 1.1 shows the BCD representation for the decimal number:

**Table 1.1 BCD Representation for Decimal Number**

Decimal Number	BCD Codes
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

From the table, it is clear that BCD codes are only valid for decimal numbers less than 10. So, if the need arises to represent a decimal value greater than 9 in the BCD format, then each digit in the decimal number is to be represented individually by the corresponding BCD code.

For example,

$(23)_{10}$  is represented by 0010 0011 in BCD

$(08)_{10}$  is represented by 0000 1000 in BCD

$(921)_{10}$  is represented by 1001 0010 0001 in BCD

Similarly, if you have a BCD number presentation and you have to find its equivalent integer value, then you have to make a group of 4-bits starting from the LSB bit to the MSB bit. If the last group does not have 4-bits, then pad with zeros.

**NOTES**

For example,

01010010 represents  $52_{10}$

01110011 represents  $73_{10}$

**3. Excess-3 Codes**

This is another form of BCD code in which each decimal digit is coded into a 4-bit binary code. The code for each decimal digit is obtained by adding decimal 3 to the corresponding BCD code. For example, decimal 2 is coded as  $0010 + 0011 = 0101$  in the excess-3 code.

It is not a weighted code like straight binary or BCD code. Also, it is a self-complementing code, which means that 1's complement of the coded number yields 9's complement of the number itself. For example, the excess-3 code of decimal 2 is 0101 and its 1's complement is 1010, which is excess-3 code for decimal 7. It also represents the 9's complement of 2.

This property helps in performing subtraction operation in digital systems. The complete Table 1.2 is as follows:

**Table 1.2 Excess 3 Codes**

Decimal Number	BCD Codes	Excess-3 Code
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100

**4. Gray Codes**

It is a very useful code in which a decimal number is represented in the binary form in such a way that each Gray code differs from the preceding and the succeeding numbers by a single bit. It is not a weighted code. It is also known as reflected code.

**Construction of Gray Code**

A 1-bit Gray code has two codes 0 and 1 representing decimal numbers 0 and 1. An n-bit ( $n \geq 2$ ) Gray code will have first  $2^{n-1}$  Gray codes with n – 1 bits (Least Significant Bits or LSB) written in order with a leading 0 appended; and the last  $2^{n-1}$  Gray codes with n – 1 bits (LSB) written in the reverse order (mirror image) with a leading 1 appended.

For example,

**1-Bit Gray Code**

Decimal Number	Gray Code
0	0
1	1

**2-Bit Gray Code**

Decimal Number	Gray Code
0	00
1	01
2	11
3	10

**3-Bit Gray Code**

Decimal Number	Gray Code		
0	0	0	0
1	0	0	1
2	0	1	1
3	0	1	0
4	1	1	0
5	1	1	1
6	1	0	1
7	1	0	0

**NOTES**

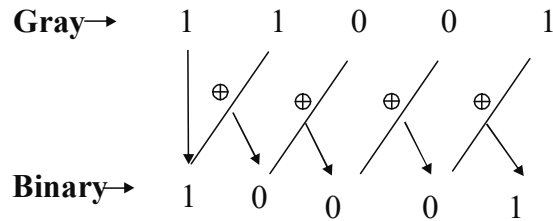
**Conversion from Binary to Gray**

The MSB of Gray code is the same as the MSB of the corresponding binary code. EXOR is the MSB to the next bit of the binary number. This will give the next bit of the Gray code number. Continue the EXOR operation on each bit of the binary to the next bit to its right to get the Gray code for that position shown as follows:

		+	+	-	+	+	+
<b>Binary</b>	1	1	0	1	0	0	1
	↓	↓	↓	↓	↓	↓	↓
<b>Gray</b>	1	0	1	1	1	0	1

**Conversion from Gray to Binary**

The MSB of binary code is the same as the MSB of the corresponding Gray code. EXOR is the result of the binary code to the next bit of the Gray code number to get the next bit of the binary code number. Continue the EXOR operation on the result bit of the binary code to the next bit of the Gray code to get the binary code for that position as follows:



⊕ X-OR

## 5. Alphanumeric Codes

Sometimes, digital systems are required to handle data that may consist of numerals, letters and special symbols. If you use an n-bit binary code, you can represent  $2^n$  elements using this code. Therefore, to represent 10 digits 0 through 9 and 26 alphabets A through Z, you need a minimum of 6-bits.

In some applications, sometimes it is required to represent more than 64 characters including the lower-case letters and the special control characters for the transmission of the digital information. For this reason, make use of the new codes, namely:

- Extended BCD Interchange Code (EBCDIC)
- American Standard Code for Information Interchange (ASCII)

### American Standard Code for Information Interchange (ASCII)

This is required for representing more than 64 characters. It is a 7-bit code, so a maximum of 128 different characters can be represented by this code. For example, 1000001 is the ASCII representation of alphabet A.

### Extended BCD Interchange Code (EBCDIC)

It is also used to represent more than 64 characters and it is an 8-bit code. The 8-bit (Most Significant Bit or MSB) is invariably added for parity. A maximum of 128 different characters can be represented by this code. A parity bit is an extra bit included with the message to make the total number of 1's either odd or even, depending on the parity need either even or odd. Odd parity makes the total number of 1's odd including parity bit and even parity makes the total number of 1's even, including parity bit. For example, 11000001 is the EBCDIC representation of alphabet A.

The table for the EBCDIC is as follows:

Character	6-bit Internal Code	8-bit EBCDIC
A	010001	11000001
B	010010	11000010
C	010011	11000011
D	010100	11000100
E	010101	11000111
F	010110	11000110
H	011000	11001000
I	011001	11001001

**NOTES**

J	100001	11010001
K	100010	11010010
L	100011	11010011
M	100100	11010100
N	100101	11010101
O	100110	11010110
P	100111	11010111
Q	101000	11011000
R	101001	11011001
S	110010	11100010
T	110011	11100011
U	110100	11100100
V	110101	11100101
W	110110	11100110
X	110111	11100111
Y	111000	11101000
Z	111001	11101001
0	000000	11110000
1	000001	11110001
2	000010	11110010
3	000011	11110011
4	000100	11110100
5	000101	11110101
6	000110	11110110
7	000111	11110111
8	001000	11111000
9	001001	11111001
blank	110000	01000000
⊙	011011	01001011
(	111100	01001101
+	010000	01001110
\$	101011	01011011
*	101100	01011100
,	011100	01011101
–	100000	01100000
,	110001	01101011
,	111011	01101011
=	001011	01111110

**Check Your Progress**

1. What is binary number system?
2. Define mixed numbers.
3. How do you add or subtract floating-point values?
4. What are the three possible techniques for representing signed integers?
5. State the character codes of computers.

## 1.5 LOGIC GATES

A logic gate is an idealized model of computation or physical electronic device implementing a Boolean function, a logical operation performed on one or more binary inputs that produces a single binary output. Depending on the context, the term may refer to an ideal logic gate, one that has for instance zero rise time and unlimited fan-out, or it may refer to a non-ideal physical device.

Logic gates are primarily implemented using diodes or transistors acting as electronic switches, but can also be constructed using vacuum tubes, electromagnetic relays (relay logic), fluidic logic, pneumatic logic, optics, molecules, or even mechanical elements. With amplification, logic gates can be cascaded in the same way that Boolean functions can be composed, allowing the construction of a physical model of all of Boolean logic, and therefore, all of the algorithms and mathematics that can be described with Boolean logic.

Logic circuits include such devices as multiplexers, registers, Arithmetic Logic Units (ALUs), and computer memory, all the way up through complete microprocessors, which may contain more than 100 million gates. In modern practice, most gates are made from MOSFETs (Metal–Oxide–Semiconductor Field-Effect Transistors).

Compound logic gates AND-OR-Invert (AOI) and OR-AND-Invert (OAI) are often employed in circuit design because their construction using MOSFETs is simpler and more efficient than the sum of the individual gates.

### 1.5.1 NOT Gate

The basic NOT gate has only one input and one output. The output is always the opposite or negation of the input. The following is the truth table for NOT gate:

Table 1.3 Truth Table for NOT Gate

A	F
0	1
1	0

Symbol:  $F = A'$

The following is the figure of NOT gate representation:

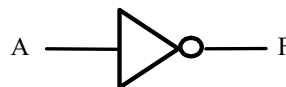


Fig. 1.3 NOT Gate

### 1.5.2 AND Gate

A basic AND gate consists of two inputs and an output. In the AND gate, the output is 'High' or gate is 'On' only if both the inputs are 'High'. The relationship between the input signals and the output signals is often represented in the form of a **truth table**. It is nothing but a tabulation of all possible input combinations and the resulting outputs. For the AND gate, there are four possible combinations of



input states:  $\{A=0, B=0\}$ ;  $\{A=0, B=1\}$ ;  $\{A=1, B=0\}$ ; and  $\{A=1, B=1\}$ .  
In the truth table, these are listed as follows:

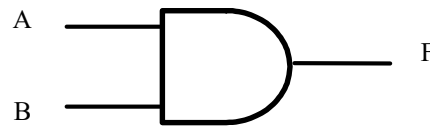
**Table 1.4 Truth Table for AND Gate**

A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

In Table 1.4, F represents the output of two inputs in the AND gate with input signals A and B.

**Symbol:**  $F = A \cdot B$  (where ‘.’ implies AND operation)

The following figure represents the AND gate:



**Fig. 1.4 AND Gate**

### 1.5.3 OR Gate

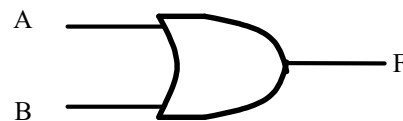
A basic OR gate is a two input, single output gate. Unlike the AND gate, the output is 1 when any one of the input signals is 1. The OR gate output is 0 only when both the inputs are 0. The truth table for the OR gate is as follows:

**Table 1.5 Truth Table for OR Gate**

A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

**Symbol:**  $F = A + B$  (where ‘+’ implies OR operation)

The following figure represents OR gate:



**Fig. 1.5 OR Gate**

### 1.5.4 XOR Gate

A gate related to the OR gate is the XOR gate or exclusive OR gate in which the output is 1 when one, and only one, of the inputs is 1. In other words, the XOR output is 1 if the inputs are different. The truth table for the XOR gate is as follows:

NOTES

NOTES

Table 1.6 Truth Table for XOR Gate

A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

Symbol:  $F = A \oplus B$  (where ‘ $\oplus$ ’ implies XOR operation)

The following figure represents XOR gate:

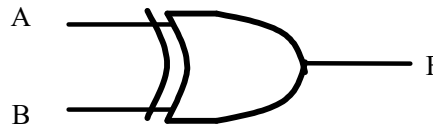


Fig. 1.6 XOR Gate

## 1.6 BOOLEAN ALGEBRA

Basic logic functions and operations are the AND function (logical multiplication), the OR function (logical addition) and the NOT operation (logical complementation).

### 1.6.1 Logical AND Operation

The logical AND operation between two Boolean variables, **A** and **B** is written as  $Y = A \cdot B$ . The common symbol for this operation is the multiplication sign ( $\cdot$ ). Table 1.7 shows that the result of logically ANDing the variables **A** and **B** is logical 0 for all cases except when both **A** and **B** are logical 1. Normally, the dot denoting the AND function is omitted and  $A \cdot B$  is written as **AB**.

### 1.6.2 Logical OR Operation

The logical OR operation between two Boolean variables, **A** and **B** is written as  $Y = A + B$  and can be represented by a truth table. Table 1.8 shows that the result of ORing the variables **A** and **B** is logical 1 when either **A** or **B** (or both) are logical 1. The common symbol for logical addition is the plus sign ( $+$ ).

Table 1.7 Logical AND Operation

A	B	$Y = A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

Table 1.8 Logical OR Operation

A	B	$Y = A + B$
0	0	0
0	1	1
1	0	1
1	1	1

### 1.6.3 Logical Complementation (Inversion)

The logical inverse operation changes logical 1 to logical 0 and vice versa. It is also called the NOT operation. The common symbol for this operation is a bar

over the function or variable. Several notations are used to indicate the NOT operation, such as adding asterisks, stars, primes, etc., 'NOT **A**' or the 'Complement of **A**' will be written as  $\bar{A}$  or  $A'$ .

A Boolean function is an algebraic expression formed with binary variables, the logic operation symbols, parentheses and equal sign. A Boolean function can be transformed from an algebraic expression into a logic diagram composed of AND, OR, NOT (inverter) gates.

The purpose of Boolean algebra is to facilitate the analysis and design of digital circuits. It provides a convenient tool to:

- (i) Express in algebraic form a truth table relationship between variables.
- (ii) Express in algebraic form the input-output relationship of logic diagram.
- (iii) Find simpler circuits for the same function.

### 1.6.4 Basic Laws of Boolean Algebra

Three logic functions (AND, OR and NOT or complement) provide the foundation for all digital systems analysis and design. Logic operations can be expressed and minimized mathematically using, laws and theorems of Boolean algebra. It is a convenient and systematic method of expressing and analysing the operation of digital circuits and systems. The following are the basic laws of Boolean algebra.

**Boolean addition:** The basic rules of Boolean addition are given as follows:

$$\begin{aligned} 0 + 0 &= 0 \\ 0 + 1 &= 1 \\ 1 + 0 &= 1 \\ 1 + 1 &= 1 \end{aligned}$$

Boolean addition is same as the logical OR addition.

**Boolean Multiplication:** The basic rules of Boolean multiplication method are given as follows:

$$\begin{aligned} 0.0 &= 0 \\ 0.1 &= 0 \\ 1.0 &= 0 \\ 1.1 &= 1 \end{aligned}$$

Boolean multiplication is same as the logical AND operation.

**Properties of Boolean Algebra:** Boolean algebra is a mathematical system consisting of a set of two or more distinct elements, two binary operators denoted by the symbols (+) and (.), and one unary operator denoted by the symbol either bar (–) or prime ('). This satisfies the commutative, associative, distributive, absorption, consensus and idempotency properties of the Boolean algebra.

NOTES

**Commutative Laws:** Boolean addition is commutative and is given by,

$$\mathbf{A + B = B + A} \quad \dots(1.1)$$

$$\mathbf{A \cdot B = B \cdot A} \quad \dots(1.2)$$

These laws indicate that the order in which we OR or AND two variables is not important; the result is the same.

**Associative Laws:**

$$\mathbf{A + (B + C) = (A + B) + C} \quad \dots(1.3)$$

$$\mathbf{A \cdot (B \cdot C) = (A \cdot B) \cdot C} \quad \dots(1.4)$$

These laws state that we can group the variables in an AND expression or OR expression any way we want.

**Distributive Laws:**

$$\mathbf{A \cdot (B + C) = A \cdot B + A \cdot C} \quad \dots(1.5)$$

$$\mathbf{A + B \cdot C = (A + B) \cdot (A + C)}$$

These laws state that an expression can be expanded by multiplying term by term just the same as in ordinary algebra.

This theorem also indicates that you can factor an expression. That is, if you have a sum of two (or more) terms, each of which contains a common variable. The common variable can be factored out just as in ordinary algebra.

$$\mathbf{A + (B \cdot C) = (A + B) \cdot (A + C)} \quad \dots(1.6)$$

**Proof:**

$$\begin{aligned} \mathbf{A + B \cdot C} &= \mathbf{A \cdot 1 + B \cdot C} && (\because \mathbf{A \cdot 1 = A}) \\ &= \mathbf{A \cdot (1 + B) + BC} && (\because \mathbf{1 + B = 1}) \\ &= \mathbf{A \cdot 1 + A \cdot B + BC} && (\because \mathbf{A(B + C) = AB + AC}) \\ &= \mathbf{A \cdot (1 + C) + AB + BC} && (\because \mathbf{1 + C = 1}) \\ &= \mathbf{A \cdot 1 + AC + AB + BC} \\ &= \mathbf{A \cdot A + AC + AB + BC} && (\because \mathbf{A \cdot A = A}) \\ &= \mathbf{A(A + C) + B(A + C)} \end{aligned}$$

$$\therefore \mathbf{A + (B \cdot C) = (A + B) \cdot (A + C)}$$

**Absorption Laws:**

(i)  $\mathbf{A + AB = A} \quad \dots(1.7)$

**Proof:**  $\mathbf{A + AB = A \cdot 1 + AB = A(1 + B)}$

$$\therefore \mathbf{A + AB = A \cdot 1 = A}$$

(ii)  $\mathbf{A \cdot (A + B) = A} \quad \dots(1.8)$

**Proof:**  $\mathbf{A \cdot (A + B) = A \cdot A + A \cdot B}$   
 $\mathbf{= A + AB = A(1 + B) = A \cdot 1}$

$$\therefore \mathbf{A \cdot (A + B) = A}$$

(iii)  $\mathbf{A + \bar{A} \cdot B = A + B} \quad \dots(1.9)$

**Proof:**  $A + \bar{A} \cdot B = (A + \bar{A})(A + B) \quad [ \because A + BC = (A + B)(A + C) ]$   
 $= 1 \cdot (A + B) \quad ( \because A + \bar{A} = 1 )$

$\therefore A + \bar{A} \cdot B = A + B$

**(iv)**  $A(\bar{A} + B) = AB \quad \dots(1.10)$

**Proof:**  $A(\bar{A} + B) = (A\bar{A}) + (AB)$

$\therefore A(\bar{A} + B) = AB \quad ( \because A\bar{A} = 0 )$

**Consensus Laws:**

**(i)**  $AB + \bar{A}C + BC = AB + \bar{A}C \quad \dots(1.11)$

**Proof:**  $AB + \bar{A}C + BC = AB + \bar{A}C + BC \cdot 1$   
 $= AB + \bar{A}C + BC(A + \bar{A}) \quad ( \because A + \bar{A} = 1 )$   
 $= AB + \bar{A}C + ABC + \bar{A}BC$   
 $= AB(1 + C) + \bar{A}C(1 + B)$

$\therefore AB + \bar{A}C + BC = AB + \bar{A}C \quad ( \because 1 + C = 1 )$

**(ii)**  $(A + B)(\bar{A} + C)(B + C) = (A + B)(\bar{A} + C) \quad \dots(1.12)$

**Proof:**  $(A + B)(\bar{A} + C)(B + C) = (A + B)(\bar{A} + C)(B + C + 0)$   
 $= (A + B)(\bar{A} + C)(B + C + A\bar{A})$   
 $= (A + B)(\bar{A} + C)(B + C + A)(B + C + \bar{A})$   
 $[ \because A + BC = (A + B)(A + C) ]$   
 $= (A + B)(A + B + C)(\bar{A} + C)(\bar{A} + C + B)$

$\therefore (A + B)(\bar{A} + C)(B + C) = (A + B)(\bar{A} + C) \quad [ \because A(A + B) = A ]$

The other basic laws of Boolean algebra are given in Table 1.9.

**Truth Table 1.9 Laws of Boolean Algebra**

Boolean laws			
2.13	(a) $A + 0 = A$	(b) $A \cdot 1 = A$	
2.14	(a) $A + 1 = 1$	(b) $A \cdot 0 = 0$	
2.15	(a) $A + A = A$	(b) $A \cdot A = A$	Idempotency
2.16	(a) $A + \bar{A} = 1$	(b) $A \cdot \bar{A} = 0$	Full set, null set
2.17	(a) $\bar{0} = 1$	(b) $\bar{1} = 0$	
2.18	(a) $\bar{\bar{A}} = A$	(b) $\bar{\bar{A}} = A$	Double inversion or involution

**Principle of Duality:** From the above properties and laws of Boolean algebra, it is seen that they are grouped in pairs as (a) and (b). One expression can be obtained from the other by replacing every 0 with 1, every 1 with 0, every (+) with (.) and every (.) with (+). Any pair of expression satisfying this property is called **dual expression**. This characteristic of Boolean algebra is called the **principle of duality**.

### 1.6.5 De Morgan's Theorems

A great mathematician De Morgan has contributed with two of the most important theorems of Boolean algebra. De Morgan's theorems are extremely useful in simplifying expression in which product or sum of variables are complemented. The two theorems are:

**Theorem I:**  $\overline{A+B+C} \dots\dots\dots = \bar{A} \cdot \bar{B} \cdot \bar{C} \dots\dots\dots$

**Theorem II:**  $\overline{A \cdot B \cdot C} \dots\dots\dots = \bar{A} + \bar{B} + \bar{C} \dots\dots\dots$

The complement of an OR sum equals the AND product of the complements. The complement of an AND product is equal to the OR sum of the complements.

These two theorems can be easily proved by checking each one for all values of **A, B, C**, etc.

The complement of any Boolean expression, a part of any expression, may be found by means of these theorems. In these rules, two steps are used to form a complement.

1. The + symbols are replaced with • symbols and • symbols with + symbols.
2. Each of the term in the expression is complemented.

#### Implications of De Morgan's Theorems

Consider **Theorem I**,  $\overline{A+B} = \bar{A} \cdot \bar{B}$

The equation in the left can be seen as the output of a NOR gate whose inputs are **A** and **B**. The right-hand side of the equation is the result of the first inverting both **A** and **B** and then putting them through an AND gate. These two representations are equivalent and are illustrated in Figure 1.7. Hence, an AND gate with inverters on each of its inputs is equivalent to a NOR gate.

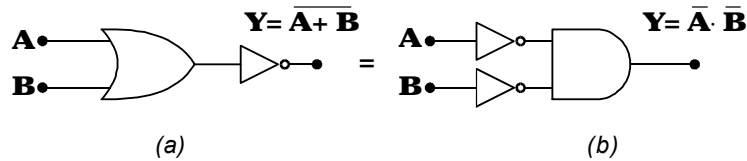


Fig. 1.7 De Morgan's Theorems

Consider **Theorem II**,  $\overline{A \cdot B} = \bar{A} + \bar{B}$

The left-hand side of the equation can be implemented by a NAND gate with inputs **A** and **B**. The right-hand side can be implemented by first inverting inputs **A** and **B** and then putting them through an OR gate. These two equivalent representations are shown in Figure. 1.7(a). The OR gate with inverters on each of its inputs is equivalent to the NAND gate. When the OR gate with inverted inputs is used to represent the NAND function, it is usually drawn as shown in Figure 1.7(b).

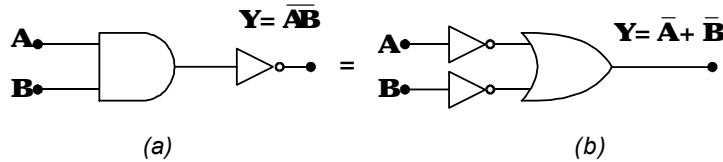


Fig. 1.8 OR Gate used to Represent the NAND Function

De Morgan's theorems can be proved for any number of variables and proof of these two theorems for 2-input variables is shown in Table 1.10.

Table 1.10 DeMorgan's Theorems

A	B	$\bar{A}$	$\bar{B}$	$A+B$	$A \cdot B$	$\overline{A+B}$	$\bar{A}\bar{B}$	$\overline{A \cdot B}$	$\bar{A} + \bar{B}$	$\overline{\bar{A}\bar{B}}$
0	0	1	1	0	0	1	1	1	1	1
0	1	1	0	1	0	0	1	1	1	0
1	0	0	1	1	0	0	1	1	1	0
1	1	0	0	1	1	0	0	0	0	0

### 1.6.6 Realization of Expression using Gates

Some examples of realization of expression using gates are given below:

Example 1.8: Simplify  $\overline{\overline{AB} + \overline{ABC} + A(B + \overline{AB})}$

$$\begin{aligned}
 \text{Solution: } \overline{\overline{AB} + \overline{ABC} + A(B + \overline{AB})} &= \overline{\overline{AB} + \overline{ABC} + \overline{AB} + \overline{A\overline{B}}} \\
 &= \overline{A(\overline{B} + \overline{BC}) + \overline{AB} + \overline{A\overline{B}}} \\
 &= \overline{A(\overline{B} + \overline{C}) + \overline{AB} + \overline{A\overline{B}}} \\
 &= \overline{\overline{AB} + \overline{AC} + A(\overline{B} + \overline{B})} \\
 &= \overline{\overline{AB} + \overline{AC} + A} \\
 &= \overline{\overline{AB} + \overline{AC} + A} \\
 &= (\overline{\overline{AB}}) \cdot (\overline{\overline{AC}}) + \overline{A} \\
 &= (\overline{A} + \overline{B}) \cdot (\overline{A} + \overline{C}) + \overline{A} \\
 &= \overline{A} + \overline{AC} + \overline{AB} + \overline{BC} + \overline{A} \\
 &= \overline{A}(1 + \overline{C} + \overline{B}) + \overline{BC} + \overline{A} \\
 &= \overline{A} + \overline{BC} + \overline{A} = 1 + \overline{BC} \\
 &= \overline{1} = 0
 \end{aligned}$$

Example 1.9: Show that  $Y = \overline{ABC} + \overline{A\overline{B}C} + \overline{A\overline{B}\overline{C}}$  can be simplified to  $Y = A(\overline{B} + \overline{C})$

$$\begin{aligned}
 \text{Solution: } Y &= \overline{ABC} + \overline{A\overline{B}C} + \overline{A\overline{B}\overline{C}} \\
 &= \overline{AC(B + \overline{B})} + \overline{A\overline{B}C} = \overline{AC}1 + \overline{A\overline{B}C} \\
 Y &= \overline{A(C + \overline{BC})} = \overline{A(C + \overline{B})} \\
 &= \overline{A}(\overline{C + \overline{B}}) = \overline{A}(\overline{C} + \overline{\overline{B}}) \\
 &= \overline{A}(\overline{C} + B) = \overline{A}(B + \overline{C})
 \end{aligned}$$

### NOTES

NOTES

**Example 1.10:** Using Boolean algebra simplify the following expression:

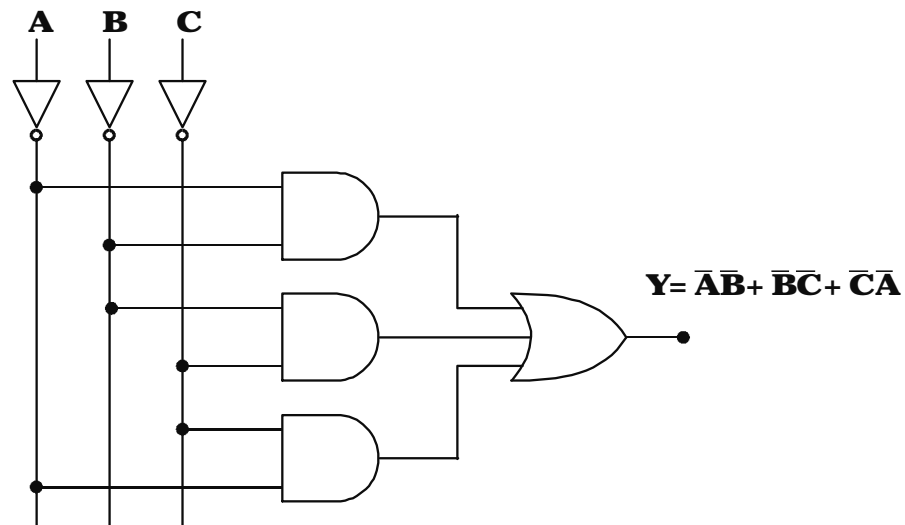
$$Y = \overline{A}BC + \overline{A}B\overline{C} + \overline{A}B\overline{C} + \overline{A}BC$$

Realize the simplified expression for the above equation using basic logic gates.

**Solution:**

$$\begin{aligned} Y &= \overline{A}BC + \overline{A}B\overline{C} + \overline{A}B\overline{C} + \overline{A}BC \\ &= \overline{B}\overline{C}(A + \overline{A}) + \overline{A}B\overline{C} + \overline{A}BC \\ &= \overline{B}\overline{C}1 + \overline{A}B\overline{C} + \overline{A}BC \quad (\because A + \overline{A} = 1) \\ &= \overline{B}\overline{C} + \overline{A}B\overline{C} + \overline{A}BC \\ &= \overline{B}(\overline{C} + \overline{A}C) + \overline{A}BC \\ &= \overline{B}(\overline{C} + \overline{A}) + \overline{A}BC \quad (\because \overline{C} + \overline{A}C = \overline{C} + \overline{A}) \\ &= \overline{B}\overline{C} + \overline{A}\overline{B} + \overline{A}BC \\ &= \overline{B}\overline{C} + \overline{A}(\overline{B} + BC) \\ &= \overline{B}\overline{C} + \overline{A}(\overline{B} + C) \\ &= \overline{B}\overline{C} + \overline{A}\overline{B} + \overline{A}C = \overline{A}\overline{B} + \overline{B}\overline{C} + \overline{C}\overline{A} \end{aligned}$$

The logic circuit for the above simplified expression is given in the figure.



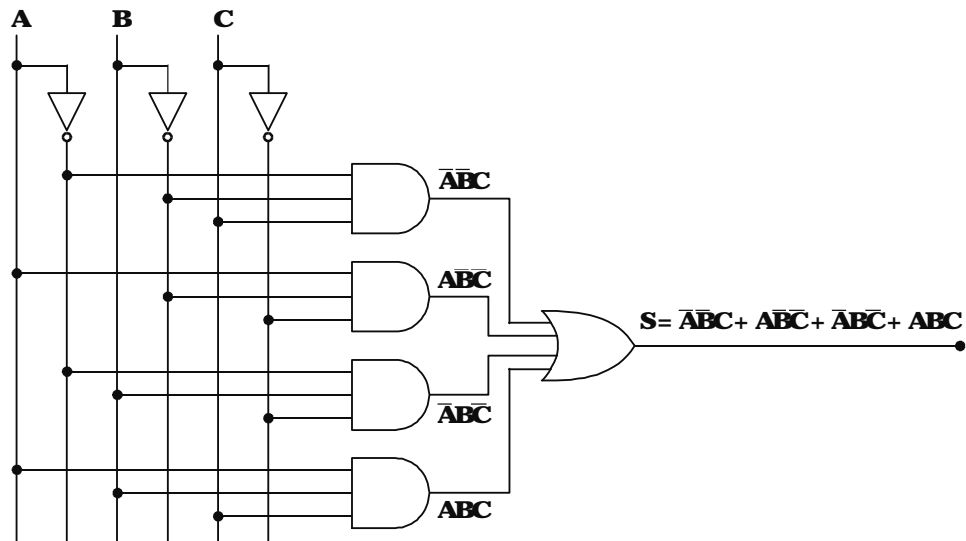
**Example 1.11:** Expand the term  $S = A \oplus B \oplus C$  using Boolean theorems. Realize the above expression using basic logic gates.

**Solution:**

$$\begin{aligned} S &= A \oplus B \oplus C \\ &= (\overline{A}B + \overline{B}A) \oplus C \\ &= (\overline{A}B + \overline{B}A) \cdot C + \overline{C} \cdot (\overline{A}B + \overline{B}A) \\ &= (\overline{A}B) \cdot (\overline{B}A) \cdot C + \overline{A}B\overline{C} + \overline{B}A\overline{C} \\ &= [(\overline{A} + \overline{B}) \cdot (\overline{B} + \overline{A})] C + \overline{A}B\overline{C} + \overline{B}A\overline{C} \\ &= [(A + \overline{B}) \cdot (B + \overline{A})] C + \overline{A}B\overline{C} + \overline{B}A\overline{C} \\ &= (\overline{A}B + \overline{A}\overline{A} + \overline{B}B + \overline{B}\overline{A}) \cdot C + \overline{A}B\overline{C} + \overline{B}A\overline{C} \\ &= (\overline{A}B + \overline{B}A) C + \overline{A}B\overline{C} + \overline{B}A\overline{C} \quad (\because \overline{A}\overline{A} = 0) \\ S &= \overline{A}B\overline{C} + \overline{B}A\overline{C} + \overline{A}\overline{B}C + \overline{A}B\overline{C} \end{aligned}$$



The logic circuit for the simplified expression is shown in the figure.

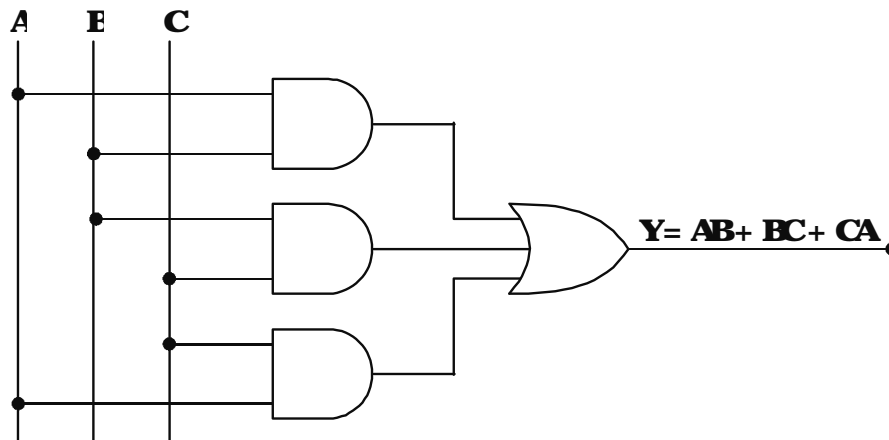


**Example 1.12:** Simplify the following Boolean expression and realize using the basic logic gates:  $ABC + \bar{A}BC + A\bar{B}C + \bar{A}\bar{B}C$

**Solution:**

$$\begin{aligned}
 Y &= ABC + \bar{A}BC + A\bar{B}C + \bar{A}\bar{B}C \\
 &= AC(B + \bar{B}) + \bar{A}BC + \bar{A}\bar{B}C \\
 &= AC1 + \bar{A}BC + \bar{A}\bar{B}C && (\because B + \bar{B} = 1) \\
 &= A(C + \bar{B}C) + \bar{A}\bar{B}C \\
 &= A(C + B) + \bar{A}\bar{B}C && (\because C + \bar{B}C = C + B) \\
 &= AC + AB + \bar{A}\bar{B}C \\
 &= AC + B(A + \bar{A}C) \\
 &= AC + B(A + C) \\
 &= AC + AB + BC = AB + BC + CA
 \end{aligned}$$

The above Boolean expression is realized using the basic logic gates as shown in the figure.



**Example 1.13:** Apply De Morgan's theorems to each of the following expressions:

(i)  $\overline{\overline{A+B+C}}$       (ii)  $\overline{\overline{A+B+CD}}$       (iii)  $\overline{\overline{(A+B)CD+E+F}}$

Solution: (i)  $\overline{\overline{A+B+C}} = \overline{\overline{A+B}} \cdot \overline{\overline{C}}$   
 $= \overline{A+B} \cdot \overline{C} = AC + BC$

(ii)  $\overline{\overline{A+B+CD}} = \overline{\overline{A+B}} \cdot \overline{\overline{CD}}$   
 $= \overline{A+B} \cdot \overline{CD}$   
 $= (\overline{A+B}) \cdot \overline{CD} = \overline{A+B} \cdot \overline{CD}$

(iii)  $\overline{\overline{(A+B)CD+E+F}} = \overline{\overline{(A+B)(\overline{C+D})+E+F}}$   
 $= \overline{\overline{AC+AD+BC+BD+E+F}}$   
 $= \overline{(\overline{AC})(\overline{AD})(\overline{BC})(\overline{BD})(\overline{E})(\overline{F}))}$   
 $= (\overline{A+C})(\overline{A+D})(\overline{B+C})(\overline{B+D})(\overline{E})(\overline{F})$   
 $= [\overline{AA} + \overline{AD} + \overline{CA} + \overline{CD}][\overline{BB} + \overline{BD} + \overline{CB} + \overline{CD}]\overline{E}\overline{F}$   
 $= (\overline{A} + \overline{AD} + \overline{CA} + \overline{CD})(\overline{B} + \overline{BD} + \overline{CB} + \overline{CD})\overline{E}\overline{F}$   
 $\quad\quad\quad (\because \overline{AA} = \overline{A})$   
 $= [\overline{A}(1+D) + \overline{CA} + \overline{CD}][\overline{B}(1+D) + (\overline{CB} + \overline{CD})]\overline{E}\overline{F}$   
 $= [\overline{A} + \overline{CA} + \overline{CD}][\overline{B} + \overline{CB} + \overline{CD}]\overline{E}\overline{F} \quad (\because 1+A=1)$   
 $= [\overline{A}(1+C) + \overline{CD}][\overline{B}(1+C) + \overline{CD}]\overline{E}\overline{F}$   
 $= [\overline{A} + \overline{CD}][\overline{B} + \overline{CD}]\overline{E}\overline{F}$   
 $= [\overline{AB} + \overline{ACD} + \overline{BCD} + \overline{CD}]\overline{E}\overline{F}$   
 $= [\overline{AB} + \overline{CD}(\overline{A+B} + \overline{CD})]\overline{E}\overline{F}$   
 $= [\overline{AB} + \overline{CD}(\overline{A+B} + 1)]\overline{E}\overline{F}$   
 $= [\overline{AB} + \overline{CD}]\overline{E}\overline{F}$

**Example 1.14:** Simplify the following functions using relevant theorems. Mention the theorems used.

(i)  $AB + \overline{AC} + \overline{BC}$       (ii)  $(A + \overline{BC})(\overline{AB} + C)$   
 (iii)  $\overline{(A+B)(\overline{A+C})(\overline{B+C})}$       (iv)  $AB + ABC + \overline{AB} + \overline{ABC}$

Solution: (i)  $AB + \overline{AC} + \overline{BC} = AB + C(\overline{A+B})$   
 $= AB + \overline{CAB}$   
 $= AB + \overline{CAB}$  (By De Morgan's theorem)

$\overline{AB} = \overline{A+B}$

$$\begin{aligned} \text{Let } AB = X, \text{ you have } \overline{AB} + \overline{CAB} &= \overline{X} + \overline{CX} \\ &= \overline{X} + \overline{C} \\ &= \overline{AB} + \overline{C} \end{aligned}$$

$$\begin{aligned} \text{(ii)} \quad (A + \overline{BC})(\overline{AB} + C) &= \overline{A}\overline{B} + \overline{A}C + \overline{A}\overline{B}C + \overline{B}CC \\ &= \overline{A}\overline{B} + \overline{A}C + \overline{A}\overline{B}C + \overline{B}C \quad (\because \overline{A}\overline{A} = \overline{A}) \\ &= \overline{A}\overline{B} + \overline{A}C(1 + \overline{B}) + \overline{B}C \\ &= \overline{A}\overline{B} + \overline{A}C + \overline{B}C \quad (\because 1 + \overline{B} = 1) \end{aligned}$$

$$\begin{aligned} \text{(iii)} \quad \overline{(A + B)(\overline{A} + \overline{C})(\overline{B} + C)} &= \overline{(\overline{A}\overline{B})(\overline{A} + \overline{C})(\overline{B} + C)} \\ &= \overline{(\overline{A}\overline{B})(\overline{A}\overline{B} + \overline{A}C + \overline{C}\overline{B} + \overline{C}C)} \\ &= \overline{\overline{A}\overline{B} \cdot \overline{A}\overline{B} + \overline{A}\overline{B}AC + \overline{A}\overline{B} \cdot \overline{C}\overline{B} + 0} \quad (\because \overline{C}C = 0) \\ &= \overline{\overline{A}\overline{B} + \overline{A}\overline{B}C + \overline{A}\overline{B}\overline{C}} \quad (\because \overline{A}\overline{B} \cdot \overline{A}\overline{B} = \overline{A}\overline{B} \text{ and } \overline{B}\overline{B} = \overline{B}) \\ &= \overline{\overline{A}\overline{B}(1 + C) + \overline{A}\overline{B}\overline{C}} \\ &= \overline{\overline{A}\overline{B} + \overline{A}\overline{B}\overline{C}} \quad (\because 1 + C = 1) \\ &= \overline{\overline{A}\overline{B}(1 + \overline{C})} \quad (\because 1 + \overline{C} = 1) \\ &= \overline{\overline{A}\overline{B}} \end{aligned}$$

$$\begin{aligned} \text{(iv)} \quad \overline{AB} + \overline{ABC} + \overline{AB} + \overline{ABC} &= \overline{AB}(1 + C) + \overline{AB} + \overline{ABC} \\ &= \overline{AB} + \overline{AB} + \overline{ABC} \quad (\because 1 + C = 1) \\ &= \overline{B}(A + \overline{A}) + \overline{ABC} \\ &= \overline{B} + \overline{ABC} \quad (\because A + \overline{A} = 1) \end{aligned}$$

**Example 1.15:** Write the truth tables of the following Boolean expressions and draw their logic diagrams:

**(i)**  $Y = \overline{ABC} + \overline{ABC} + \overline{AB}$       **(ii)**  $Y = \overline{AB} + \overline{AC}$

**Solution:** **(i)**  $Y = \overline{ABC} + \overline{ABC} + \overline{AB}$

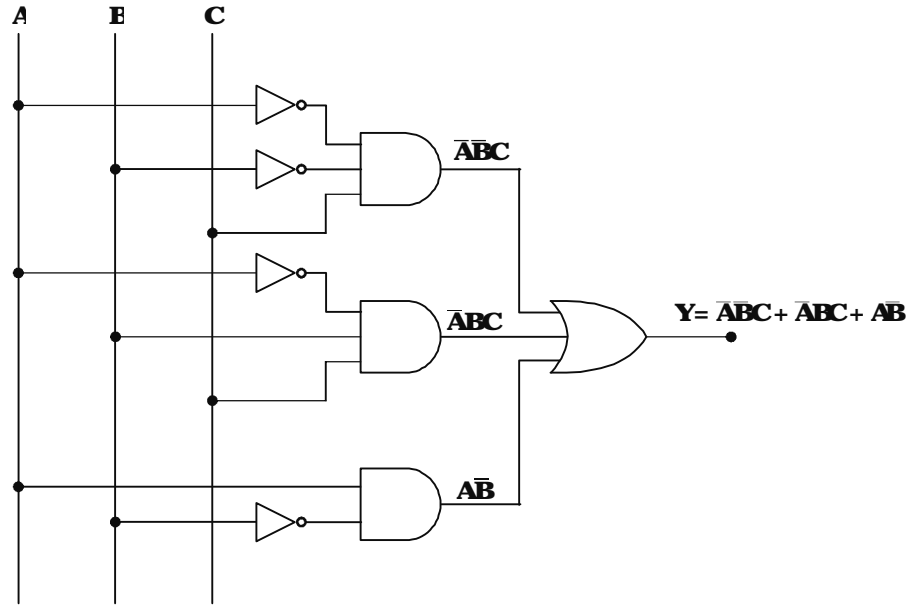
Truth Table

Inputs								Output
A	B	C	$\overline{A}$	$\overline{B}$	$\overline{ABC}$	$\overline{ABC}$	$\overline{AB}$	Y=
$\overline{ABC} + \overline{ABC} + \overline{AB}$								
0	0	0	1	1	0	0	0	0
0	0	1	1	1	1	0	0	1
0	1	0	1	0	0	0	0	0
0	1	1	1	0	0	1	0	1
1	0	0	0	1	0	0	1	1
1	0	1	0	1	0	0	1	1
1	1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0

NOTES

NOTES

Logic Diagram

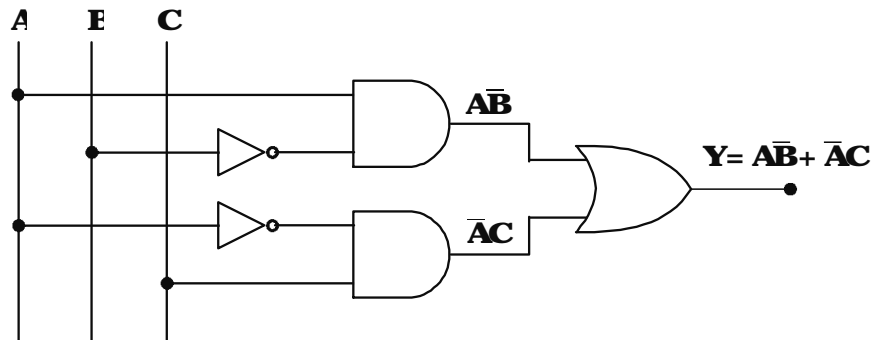


(ii)  $Y = \bar{A}\bar{B} + \bar{A}C$

Truth Table

Inputs							Output	
A	B	C	$\bar{A}$	$\bar{B}$	$\bar{A}\bar{B}$	$\bar{A}C$	$Y = \bar{A}\bar{B} + \bar{A}C$	
0	0	0	1	1	0	0	0	
0	0	1	1	1	0	1	1	
0	1	0	1	0	0	0	0	
0	1	1	1	0	0	1	1	
1	0	0	0	1	1	0	1	
1	0	1	0	1	1	0	1	
1	1	0	0	0	0	0	0	
1	1	1	0	0	0	0	0	

Logic Diagram

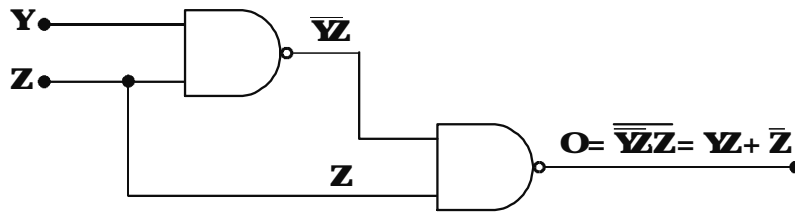


Example 1.16: Simplify and realize using only NAND gates the following Boolean expressions:

- (i)  $X\bar{Y}Z + X\bar{Y}\bar{Z} + YZ + \bar{Z}$       (ii)  $(A + \bar{B} + C)(\bar{A} + B + \bar{C})(A + \bar{B})$

Solution: (i) Output =  $XYZ + XYZ + YZ + \bar{Z}$  ( $\because A + \bar{A} = 1$ )  
 $= XYZ + YZ + \bar{Z}$   
 $= YZ(X+1) + \bar{Z}$   
 $= YZ + \bar{Z} = \overline{YZ} \cdot Z$  ( $\because 1 + A = 1$ )

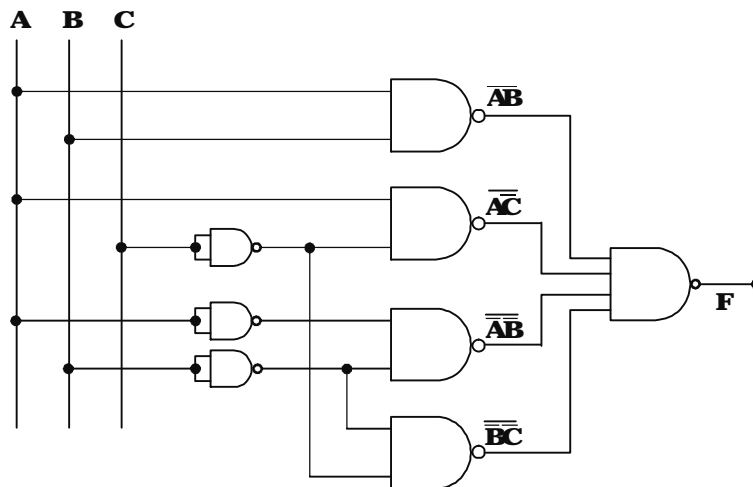
Logic Diagram



(ii)  $(A + \bar{B} + C)(\bar{A} + B + \bar{C})(A + \bar{B})$   
 $= (A\bar{A} + A\bar{B} + A\bar{C} + \bar{B}\bar{A} + \bar{B}B + \bar{B}\bar{C} + C\bar{A} + CB + C\bar{C})(A + \bar{B})$   
 $= (0 + A\bar{B} + A\bar{C} + \bar{B}\bar{A} + 0 + \bar{B}\bar{C} + C\bar{A} + CB + 0)(A + \bar{B})$  ( $\because A\bar{A} = 0$ )  
 $= A\bar{B} + A\bar{C} + \bar{B}\bar{A} + A\bar{B}\bar{C} + A\bar{C}\bar{A} + A\bar{C}B + \bar{B}\bar{A}B + \bar{B}\bar{A}\bar{C} + \bar{B}\bar{A}C + \bar{B}\bar{B}\bar{C} + \bar{B}\bar{A}C + \bar{B}\bar{C}A + \bar{B}CB$   
 $= A\bar{B} + A\bar{C} + 0 + A\bar{B}\bar{C} + 0 + A\bar{C} + 0 + \bar{B}\bar{A}\bar{C} + \bar{B}\bar{A} + \bar{B}\bar{C} + \bar{B}\bar{C}A + 0$   
 $= A\bar{B} + A\bar{C}(1 + \bar{B}) + \bar{B}\bar{A}(1 + C) + \bar{B}\bar{C}(A + 1) + A\bar{B}\bar{C}$   
 $= A\bar{B} + A\bar{C} + \bar{B}\bar{A} + \bar{B}\bar{C} + A\bar{B}\bar{C} = A\bar{B}(1 + C) + A\bar{C} + \bar{B}\bar{A} + \bar{B}\bar{C}$

$Y = AB + A\bar{C} + \bar{A}\bar{B} + \bar{B}\bar{C} = \overline{AB \cdot A\bar{C} \cdot \bar{A}\bar{B} \cdot \bar{B}\bar{C}}$

Logic Diagram



Example 1.17: Find the complement and simplify:

(i)  $AB + \bar{A}\bar{B}$  (ii)  $A + B + 1$  (iii)  $AB + 0$

Solution: (i)  $Y = AB + \bar{A}\bar{B}$   
 $\bar{Y} = \overline{AB + \bar{A}\bar{B}}$   
 $= (\bar{A}\bar{B})(\overline{AB})$

NOTES

$$\begin{aligned}
 &= (\bar{A} + \bar{B})(A + B) \\
 &= \bar{A}\bar{A} + \bar{A}B + \bar{B}A + \bar{B}\bar{B} = 0 + \bar{A}B + \bar{B}A + 0 \\
 &= \bar{A}B + \bar{B}A = A \oplus B
 \end{aligned}$$

(ii)

$$\begin{aligned}
 Y &= A + B + 1 \\
 \bar{Y} &= \overline{A + B + 1} \\
 &= \overline{A + 1} = \bar{1} \quad (\because B + 1 = 1) \\
 &= 0
 \end{aligned}$$

(iii)

$$\begin{aligned}
 Y &= \bar{A}B + 0 \\
 \bar{Y} &= \overline{\bar{A}B + 0} \\
 &= (\overline{\bar{A}B})(\bar{0}) = (\bar{A} + \bar{B})(1) = \bar{A} + \bar{B}
 \end{aligned}$$

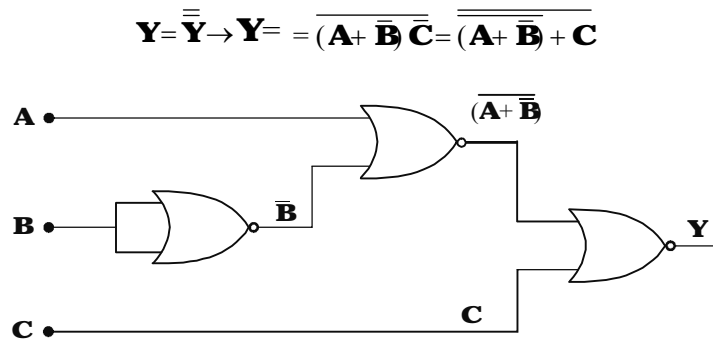
**Example 1.18:** Simplify and realize using only NOR gates:

$$Y = \bar{A}\bar{B}\bar{C} + \bar{A}B\bar{C} + \bar{B}\bar{C} + A\bar{C}$$

**Solution:**

$$\begin{aligned}
 Y &= \bar{A}\bar{B}\bar{C} + \bar{A}B\bar{C} + \bar{B}\bar{C} + A\bar{C} \\
 &= \bar{B}\bar{C}(A + \bar{A}) + \bar{B}\bar{C} + A\bar{C} \\
 &= \bar{B}\bar{C} + \bar{B}\bar{C} + A\bar{C} \quad (\because A + \bar{A} = 1) \\
 &= \bar{B}\bar{C} + A\bar{C} \quad (\because A + A = 1) \\
 &= \bar{C}(\bar{B} + A)
 \end{aligned}$$

Logic diagram using NOR gates.



**Example 1.19:** Simplify the following Boolean expression:  $f(Y) = \bar{A}B + \bar{A}C + BC$

**Solution:**

$$\begin{aligned}
 f(Y) &= \bar{A}B + \bar{A}C + BC \\
 &= \bar{A}B + \bar{A}C + BC.1 \\
 &= \bar{A}B + \bar{A}C + BC(A + \bar{A}) \quad (\because A + \bar{A} = 1) \\
 &= \bar{A}B + \bar{A}C + BCA + BC\bar{A} \\
 &= \bar{A}B(1 + C) + \bar{A}C(1 + B) \\
 &= \bar{A}B + \bar{A}C \quad (\because 1 + B = 1 + C)
 \end{aligned}$$

## 1.6.7 Combinational Logic

Combinational logic deals with the techniques of ‘Combining’ the basic gates into circuits that perform some desired functions. In combinational logic circuits, at any time, the logic level at the output depends on the combination of logic levels present at the inputs. A combinational circuit has no memory characteristic, so its output depends only on the current value of its inputs. A combinational logic digital function is completely specified by a truth table. Truth table shows output of the function corresponding to each and every possible combination of input variables. So, a truth table has number of columns equal to number of inputs plus one column for the output. If there are more than one outputs, then, only one output is considered at a time for designing the function. If there are  $n$  input variables, since each variable can have two values 0 or 1, so there are  $2^n$  possible combinations. Thus, the truth table has  $2^n$  rows. Hence, a three variable combination logic function truth table has 4 columns and 8 rows. Thus, the truth table specifies the complete requirement of the digital function.

An arbitrary logic function can be understood in the following terms:

- (i) Sum of Products (SOP)      (ii) Product of Sums (POS)

**Product Term:** The AND function is referred to as a **product**. In Boolean algebra, the word ‘Product’ loses its original meaning but serves to indicate an AND function. The logical product of several variables, on which a function depends, is considered to be a product term. The variables in a product term appear in a complemented or uncomplemented form. For example,  $\overline{A}BC$  is a product term.

**Sum Term:** An OR function is generally referred to as a **sum**. The logical sum of variables on which a function depends is considered to be a **sum term**. Variables in a sum term can appear either in complemented or uncomplemented form. For example,  $A + \overline{B} + C$  is a sum term.

### (a) Sum of Products

Sum of Products (SOP) is the logical sum of two or more logical product terms is called a sum of products expression. It is logical OR of multiple product terms. Each product term is the AND of binary literals. This is an ORing of ANDed variables such as:

$$(i) Y = AB + BC + AC \qquad (ii) Y = \overline{A}B + \overline{B}C + \overline{A}C$$

### (b) Product of Sums

A product of sums is the logical AND of multiple OR terms. Each sum term is the OR of binary literals. This is an ANDing of ORed variables such as:

$$(i) Y = (A + B)(B + C)(\overline{A} + C) \quad (ii) Y = (A + \overline{B})(\overline{B} + C)(\overline{A} + \overline{C})$$

**Minterm:** A minterm is a special case product (AND) term. A minterm is a product term that contains all of the input variables that make up a Boolean expression. A two variable function has four possible combinations, viz.,  $\overline{A}\overline{B}$ ,  $\overline{A}B$ ,  $A\overline{B}$  and  $AB$

**NOTES**

**Maxterm:** A maxterm is a special case sum (OR) term. A maxterm is a sum (OR) term that contains all of the input variables that make up a Boolean expression. A two variable function has four possible combinations, viz. **A+ B**, **A+  $\bar{B}$** ,  **$\bar{A}$ + B** and  **$\bar{A}$ +  $\bar{B}$** .

Table 1.11 shows the complement nature of minterms and maxterms. Note that an input variable is complemented when it has a value of 0, if you are writing minterms. The input variables are complemented when they have a value of 1, if you are writing maxterms. Minterms represent output variable 1s and maxterms represent output variable 0s. Lower case **m** is used to denote a minterm and upper case **M** is used to denote a maxterm. The number subscript indicates the decimal value of the term.

Output equations can be written directly from the truth table using either minterms or maxterms. When an output equation is written in minterms or maxterms, it is a canonical expression.

*Table 1.11 Minterm and Maxterm Designations for Three Variables*

Input variables			Minterm		Maxterm	
A	B	C	Term	Designation	Term	Designation
0	0	0	$\bar{A}\bar{B}\bar{C}$	$m_0$	<b>A+ B+ C</b>	<b>M<sub>0</sub></b>
0	0	1	$\bar{A}\bar{B}C$	$m_1$	<b>A+ B+ <math>\bar{C}</math></b>	<b>M<sub>1</sub></b>
0	1	0	$\bar{A}B\bar{C}$	$m_2$	<b>A+ <math>\bar{B}</math>+ C</b>	<b>M<sub>2</sub></b>
0	1	1	$\bar{A}BC$	$m_3$	<b>A+ <math>\bar{B}</math>+ <math>\bar{C}</math></b>	<b>M<sub>3</sub></b>
1	0	0	$A\bar{B}\bar{C}$	$m_4$	<b><math>\bar{A}</math>+ B+ C</b>	<b>M<sub>4</sub></b>
1	0	1	$A\bar{B}C$	$m_5$	<b><math>\bar{A}</math>+ B+ <math>\bar{C}</math></b>	<b>M<sub>5</sub></b>
1	1	0	$AB\bar{C}$	$m_6$	<b><math>\bar{A}</math>+ <math>\bar{B}</math>+ C</b>	<b>M<sub>6</sub></b>
1	1	1	$ABC$	$m_7$	<b><math>\bar{A}</math>+ <math>\bar{B}</math>+ <math>\bar{C}</math></b>	<b>M<sub>7</sub></b>

**Canonical Forms**

Canonical is a word used to describe a condition of a switching equation. In normal use, the word means ‘Conforming to a General Rule’. The ‘Rule’ for switching logic, is that each term is used in a switching equation which must contain all of the available input variables. Two formats generally exist for expressing switching equations in a canonical form : (i) Sums of Minterms, and (ii) Products of Maxterms.

- I. To place a SOP equation into canonical form using Boolean algebra, you can do the following:
  1. Identify the missing variable(s) in each AND term.
  2. AND the missing term(s) and its complement with the original AND term, **ABC+  $\bar{C}$** . Because **C+  $\bar{C}$  = 1**, the original AND term value is not changed.
  3. Expand the term by application of the property of the distribution, **ABC+  $ABC\bar{C}$** .
- II. To place a POS equation into canonical form using Boolean algebra, we do the following:



1. Identify the missing variable(s) in each OR term.
2. OR the missing term(s) and its complement with the original OR term,  $\mathbf{A + \bar{B} + C\bar{C}}$ . Because  $\mathbf{C\bar{C} = 0}$ , the original OR term value is not changed.
3. Expand the term by application of distributive property,  $\mathbf{(A + \bar{B} + C)(A + \bar{B} + \bar{C})}$ .

**Example 1.20:** Convert  $\mathbf{A + B}$  to minterms.

**Solution:**  $\mathbf{A + B = A \cdot 1 + B \cdot 1}$  ( $\because \mathbf{A \cdot 1 = A}$ )

$$= \mathbf{A(B + \bar{B}) + B(A + \bar{A})}$$

$$= \mathbf{AB + A\bar{B} + BA + B\bar{A}}$$
 (Commutative)

$\therefore \mathbf{Y = A + B = AB + A\bar{B} + \bar{A}B + B\bar{A}}$  ( $\mathbf{AB}$  is Redundant)

This general form obtained is called the **minterm canonical form**, sometimes also referred to as the standard sum.

**Example 1.21:** Convert  $\mathbf{(A + BC)}$  to minterms.

**Solution:**  $\mathbf{A + BC = A \cdot 1 + BC \cdot 1}$

$$= \mathbf{A(B + \bar{B}) + BC(A + \bar{A})}$$

$$= \mathbf{AB + A\bar{B} + ABC + BC\bar{A}}$$

$$= \mathbf{AB \cdot 1 + A\bar{B} \cdot 1 + ABC + BC\bar{A}}$$

$$= \mathbf{AB(C + \bar{C}) + A\bar{B}(C + \bar{C}) + ABC + BC\bar{A}}$$

$$= \mathbf{ABC + A\bar{B}C + A\bar{B}\bar{C} + ABC + BC\bar{A}}$$

$\mathbf{Y = A + BC = ABC + A\bar{B}C + A\bar{B}\bar{C} + ABC + BC\bar{A}}$

**Example 1.22:** Find the minterms for  $\mathbf{AB + ACD}$

**Solution:**  $\mathbf{ABXX}$  generates  $\mathbf{ABC\bar{D}, ABCD, AB\bar{C}D, ABCD}$

$\mathbf{AXCD}$  generates  $\mathbf{ABC\bar{D}}$  and  $\mathbf{A\bar{B}CD}$

$\therefore \mathbf{Y = AB + ACD}$

$$= \mathbf{ABC\bar{D} + ABCD + AB\bar{C}D + A\bar{B}CD + ABCD}$$

**Example 1.23:** Find the minterm designation of  $\mathbf{A\bar{B}C\bar{D}}$ .

**Solution:** Substituting 1's for nonbarred letters and 0's for barred letters we have

$$\mathbf{A\bar{B}C\bar{D} = 1000 = m_8}$$

**Example 1.24:** Express the function  $\mathbf{Y = A + \bar{B}C}$  in canonical SOP and POS forms.

**Solution:** (i) Sum of Products (SOP)

$$\mathbf{Y = A + \bar{B}C = A \cdot (B + \bar{B}) + \bar{B} \cdot C(A + \bar{A})}$$
 ( $\because \mathbf{A + \bar{A} = 1}$ )

$$= \mathbf{AB + A\bar{B} + \bar{A}BC + \bar{A}\bar{B}C}$$

$$= \mathbf{AB(C + \bar{C}) + \bar{A}B(C + \bar{C}) + \bar{A}\bar{B}C + \bar{A}\bar{B}C}$$

NOTES

$$\begin{aligned}
 &= \overline{A}BC + A\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\overline{C} + \overline{A}B\overline{C} + \overline{A}B\overline{C} \\
 &= \overline{A}BC + A\overline{B}C + \overline{A}B\overline{C} + \overline{A}B\overline{C} + \overline{A}B\overline{C} \\
 \mathbf{Y} &= \mathbf{m}_7 + \mathbf{m}_6 + \mathbf{m}_5 + \mathbf{m}_4 + \mathbf{m}_1
 \end{aligned}$$

The Σ sign is generally used to write the canonical SOP form of an expression,

$$\therefore \mathbf{Y} = \Sigma(1, 4, 5, 6, 7)$$

(ii) Product of Sums (POS)

$$\begin{aligned}
 \mathbf{Y} &= \mathbf{A} + \overline{\mathbf{B}}\mathbf{C} = (\mathbf{A} + \overline{\mathbf{B}})(\mathbf{A} + \mathbf{C}) && [\because \mathbf{A} + \overline{\mathbf{B}}\mathbf{C} = (\mathbf{A} + \overline{\mathbf{B}})(\mathbf{A} + \mathbf{C})] \\
 &= (\mathbf{A} + \overline{\mathbf{B}} + \mathbf{C})(\mathbf{A} + \overline{\mathbf{B}} + \overline{\mathbf{C}})(\mathbf{A} + \mathbf{B} + \mathbf{C})(\mathbf{A} + \overline{\mathbf{B}} + \mathbf{C}) \\
 &&& [\because (\mathbf{A} + \mathbf{B})(\mathbf{A} + \overline{\mathbf{B}}) = \mathbf{A}]
 \end{aligned}$$

$$\therefore \mathbf{Y} = (\mathbf{A} + \overline{\mathbf{B}} + \mathbf{C})(\mathbf{A} + \overline{\mathbf{B}} + \overline{\mathbf{C}})(\mathbf{A} + \mathbf{B} + \mathbf{C})$$

or  $\mathbf{Y} = \mathbf{M}_2 \cdot \mathbf{M}_3 \cdot \mathbf{M}_0 = \mathbf{M}_0 \mathbf{M}_2 \mathbf{M}_3$

In this case Π symbol is used to represent a product of maxterms. Then,

$$\mathbf{Y} = \Pi(0, 2, 3)$$

Deriving Sum of Product Expression from a Truth Table

The sum of product (SOP) expression for a Boolean function can be derived from its truth table by summing (OR operation) the product terms that correspond to the combinations containing a function value. In the product term, the input variable appears either in uncomplemented form if it possesses the value 1, or in complement form if it contains the value 0.

Truth Table 1.12

Inputs			Output	Product terms
A	B	C	Y	
0	0	0	0	-
0	0	1	0	-
0	1	0	1	$\overline{A}BC$
0	1	1	1	$\overline{A}BC$
1	0	0	0	-
1	0	1	0	-
1	1	0	1	$A\overline{B}C$
1	1	1	1	$A\overline{B}C$

Consider the Truth Table 1.12, for a three input circuit. Here, the output (Y) value is 1 for the input combinations 010, 011, 110 and 111 and their corresponding product terms are  $\overline{A}BC$ ,  $\overline{A}BC$ ,  $A\overline{B}C$  and  $A\overline{B}C$  respectively.

The final expression for the output Y is obtained by ORing the four AND terms. Thus,

$$\mathbf{Y} = \overline{A}BC + \overline{A}BC + A\overline{B}C + A\overline{B}C$$

The general expression for obtaining the output expression from a truth table in SOP can be summarized as follows:

1. Write a product term for each case in the table where the output is 1.
2. Each product term contains input variable in either complemented or uncomplemented form. If the input variable is 0, then it appears in complemented form, if the input variable is 1, it appears then in uncomplemented form.
3. All the product terms are then added (ORed) together to produce the final expression of the output.

### Deriving Product of Sum Expression from a Truth Table

The Product of Sum (POS) expression for a Boolean function can also be obtained from a truth table by multiplying (AND operation) of the sum terms corresponding to the combinations for which the function assumes the value 0. In the sum term, the input variable appears in an uncomplemented form if it has the value 0 in the corresponding combination, and in the complemented form if it has the value 1.

Consider the Truth Table 1.13, for a 3-input circuit. From the table it is observed that the **Y** value is 0 or the input combinations 001, 010, 100 and 110. The corresponding sum terms are  $(\mathbf{A} + \mathbf{B} + \bar{\mathbf{C}})$ ,  $(\mathbf{A} + \bar{\mathbf{B}} + \mathbf{C})$ ,  $(\bar{\mathbf{A}} + \mathbf{B} + \mathbf{C})$  and  $(\bar{\mathbf{A}} + \bar{\mathbf{B}} + \mathbf{C})$  respectively.

The final POS expression for the output **Y** is obtained by multiplying (AND operation) the four sum terms as follows:

$$\mathbf{Y} = (\mathbf{A} + \mathbf{B} + \bar{\mathbf{C}})(\mathbf{A} + \bar{\mathbf{B}} + \mathbf{C})(\bar{\mathbf{A}} + \mathbf{B} + \mathbf{C})(\bar{\mathbf{A}} + \bar{\mathbf{B}} + \mathbf{C})$$

*Truth Table 1.13*

Inputs			Output	Sum			Product
A	B	C	Y	terms	$\bar{\mathbf{Y}}$	$\bar{\mathbf{Y}}$	terms
0	0	0	1	—	$\bar{\mathbf{A}}\bar{\mathbf{B}}\bar{\mathbf{C}}$	0	—
0	0	1	0	$(\mathbf{A} + \mathbf{B} + \bar{\mathbf{C}})$	—	1	$\bar{\mathbf{A}}\bar{\mathbf{B}}\mathbf{C}$
0	1	0	0	$(\mathbf{A} + \bar{\mathbf{B}} + \mathbf{C})$	—	1	$\mathbf{A}\bar{\mathbf{B}}\bar{\mathbf{C}}$
0	1	1	1	—	$\bar{\mathbf{A}}\bar{\mathbf{B}}\mathbf{C}$	0	—
1	0	0	0	$(\bar{\mathbf{A}} + \mathbf{B} + \mathbf{C})$	—	1	$\bar{\mathbf{A}}\bar{\mathbf{B}}\bar{\mathbf{C}}$
1	0	1	1	—	$\bar{\mathbf{A}}\bar{\mathbf{B}}\mathbf{C}$	0	—
1	1	0	0	$(\bar{\mathbf{A}} + \bar{\mathbf{B}} + \mathbf{C})$	—	1	$\bar{\mathbf{A}}\bar{\mathbf{B}}\bar{\mathbf{C}}$
1	1	1	1	—	$\bar{\mathbf{A}}\bar{\mathbf{B}}\mathbf{C}$	0	—

The procedure for obtaining the output expression in POS form that from a truth table can be summarized as follows:

1. Write a sum term for each input combination in the table, which has an output value of 0.
2. Each sum term contains all its input variables in complemented or uncomplemented form. If the input variable is 0, then it appears in an uncomplemented form; if the input variable is 1, it appears in the complemented form.
3. All the sum terms are multiplied (ANDed) together to obtain the final POS expression of the output.

**NOTES**

The POS expression for a Boolean function can also be obtained from its SOP expression using the fact that POS expression is complement of SOP.

Consider a function,

$$Y = \overline{ABC} + \overline{ABC} + \overline{ABC} + ABC, \text{ in SOP.}$$

$\overline{Y}$  will be in POS form, which is:

$$\begin{aligned} & \overline{\overline{ABC} + \overline{ABC} + \overline{ABC} + ABC} \\ & = (A + B + C)(A + B + C)(A + \overline{B} + \overline{C})(\overline{A} + \overline{B} + \overline{C}) \end{aligned}$$

This expression is in the desired POS form.

**Check Your Progress**

6. What do you mean by NOT gate?
7. Define the term truth table?
8. State the rules of Boolean addition.
9. What are De Morgan's laws?

**1.7 BOOLEAN EXPRESSION SIMPLIFICATION**

Sometimes, you need to simplify the Boolean expression. The main advantage in doing so is that, it then uses less logic gates and less power to realize and thus, it is considered sometimes cheaper and faster.

There are basically two types of simplification techniques:

- Algebraic Simplification
- Karnaugh Maps (K-map)

**1.7.1 Algebraic Simplification**

This involves simplifications using Boolean theorems. Algebraic simplification aims to minimize the number of literals and terms.

For example, to reduce the Boolean expression  $F = (x+y).(x+y').(x'+z)$

(6 Literals)

$$\begin{aligned} F &= (x.x + x.y' + x.y + y.y').(x'+z) && \text{(Associative)} \\ &= (x+x.(y'+y)+0).(x'+z) && \text{(Idempotency, Associative, Complement)} \\ &= (x+x.(1)+0).(x'+z) && \text{(Complement)} \\ &= (x+x+0).(x'+z) && \text{(Identity 1)} \\ &= (x).(x'+z) && \text{(Idempotency, Identity 0)} \\ &= (x.x'+x.z) && \text{(Associative)} \\ &= (0+x.z) && \text{(Complement)} \\ &= x.z && \text{(Identity 0)} \end{aligned}$$

Number of literals reduced from 6 to 2.

For example,

1. Finding the minimal SOP and POS expressions of:

$$\begin{aligned}
 F(x,y,z) &= x'.y.(z + y'.x) + y'.z \\
 &= x'.y.z + x'.y.y'.x + y'.z && \text{(Distributive)} \\
 &= x'.y.z + 0 + y'.z && \text{(Complement, Null element 0)} \\
 &= x'.y.z + y'.z && \text{(Identity 0)} \\
 &= x'.z + y'.z && \text{(Absorption)} \\
 &= (x' + y').z && \text{(Distributive)}
 \end{aligned}$$

Minimal SOP of  $F = x'.z + y'.z$  (Two 2-input AND gates and one 2-input OR gate)

Minimal POS of  $F = (x' + y').z$  (One 2-input OR gate and one 2-input AND gate)

2. Finding the minimal SOP expression of:

$$\begin{aligned}
 F(a,b,c,d) &= a.b.c + a.b.d + a'.b.c' + c.d + b.d' \\
 &= a.b.c + \underline{a.b.d} + a'.b.c' + c.d + \underline{b.d'} && \text{(Absorption on underlined terms)} \\
 &= a.b.c + \underline{a.b} + \underline{a'.b.c'} + c.d + b.d' && \text{(Absorption on underlined terms)} \\
 &= \underline{a.b.c} + \underline{a.b} + b.c' + c.d + b.d' && \text{(Absorption on underlined terms)} \\
 &= a.b + \underline{b.c'} + c.d + \underline{b.d'} && \text{(Distributivity on underlined terms)} \\
 &= a.b + c.d + b.(\underline{c'} + \underline{d'}) && \text{(De Morgan on underlined terms)} \\
 &= a.b + \underline{c.d} + \underline{b.(c.d)'} && \text{(Absorption on underlined terms)} \\
 &= \underline{a.b} + c.d + \underline{b} && \text{(Absorption on underlined terms)} \\
 &= b + c.d
 \end{aligned}$$

Number of literals is reduced from 13 to 3.

However, the difficulty with this method is that it needs good algebraic manipulation skills.

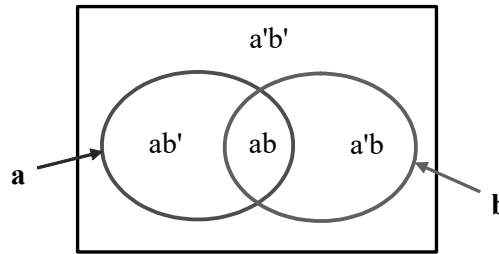
### 1.7.2 Karnaugh Map

It is a diagram-based simplification technique. It is easy for the circuit designer and involves pattern-matching skills. It gives simplified Boolean expressions in standard forms. However, this can be effectively utilized for reducing Boolean expressions with input variables less than 6.

It is a systematic method to obtain simplified Sum-Of-Products (SOP) Boolean expressions with the objective of fewest possible terms/literals. It is a diagrammatic technique based on a special form of Venn diagram. Here, Venn

**NOTES**

diagrams represent the space of Minterms. An example of a 2 variable (4 Minterms) Venn diagram is shown in Figure 1.9:



**Fig. 1.9 Venn Diagram (4 Minterms)**

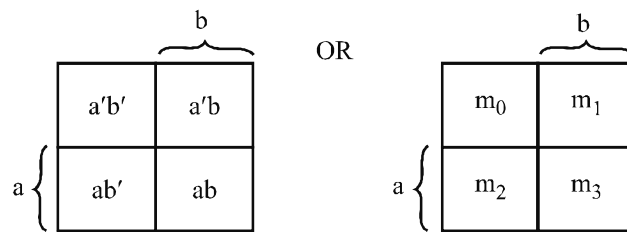
Each set of Minterms represents a Boolean function. For example,

$$\begin{aligned} \{a.b, a.b'\} &\rightarrow a.b + a.b' = a.(b+b') = a \\ \{a'.b, a.b\} &\rightarrow a'.b + a.b = (a'+a).b = b \\ \{a.b\} &\rightarrow a.b \\ \{a.b, a.b', a'.b\} &\rightarrow a.b + a.b' + a'.b = a + b \\ \{\} &\rightarrow 0 \\ \{a'.b', a.b, a.b', a'.b\} &\rightarrow 1 \end{aligned}$$

**2-Variable K-Map**

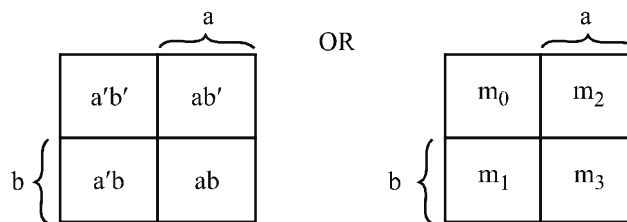
A Karnaugh map (K-map) is an abstract form of Venn diagram, organized as a matrix of squares, where each square represents a Minterm. Also, adjacent squares always differ by just one literal (so that the unifying theorem may apply:  $a + a' = 1$ ). For 2-variable case (e.g., variables a, b), assuming that **a** is the MSB and **b** is the LSB, the map can be drawn as follows:

**Alternative 1:**



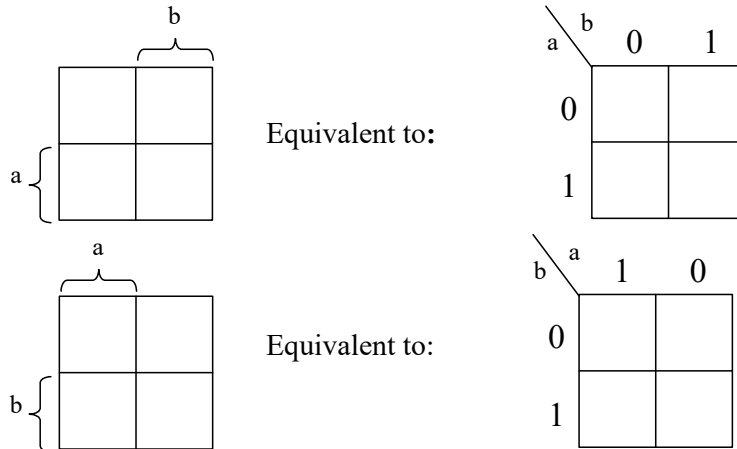
{-symbol implies that corresponding literal is in normal form.

**Alternative 2:**



{-symbol implies that corresponding literal is in normal form.

**Equivalent Labelling:**

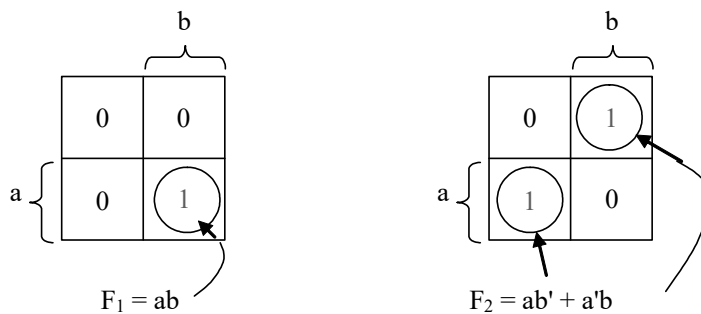


**NOTES**

The K-map for a function, which is the sum of Minterms, is specified by putting:

- '1' in the square corresponding to a Minterm
- '0' otherwise

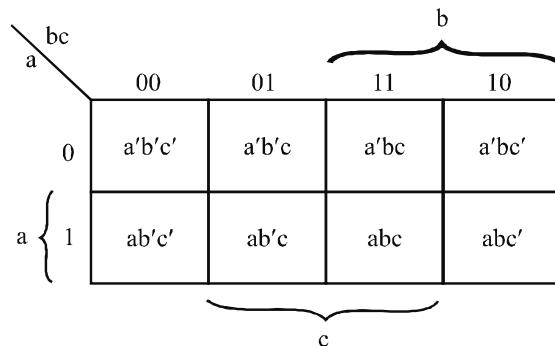
For example, if  $F_1 = a.b$  and  $F_2 = a'.b + a.b'$ , then the K-map entry for  $F_1$  and  $F_2$  will be as follows:



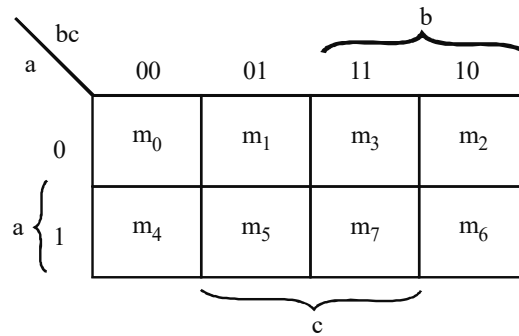
Here 1 is entered to the locations of Minterms of Boolean expression.

**3-Variable K-Map**

There are 8 Minterms for 3 variables (a, b, c). Therefore, there are 8 cells in a 3-variable K-map.



It is to be noted that the above arrangement ensures that Minterms of adjacent cells differ by only **one literal**



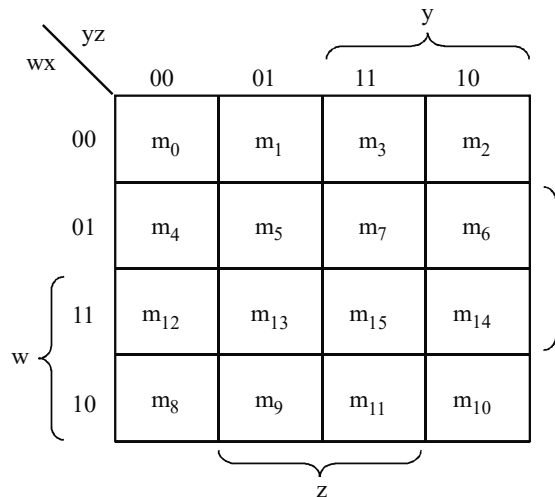
It is to be noted that there is wrap-around in the K-map:

- $a'.b'.c'$  ( $m_0$ ) is adjacent to  $a'.b.c'$  ( $m_2$ ) since only one literal **b** is different.
- $a.b'.c'$  ( $m_4$ ) is adjacent to  $a.b.c'$  ( $m_6$ ) since only one literal **b** is different.

Each cell in a 3-variable K-map has 3 adjacent neighbours. In general, each cell in an n-variable K-map has n adjacent neighbours. For example,  $m_0$  has 3 adjacent neighbours  $m_1$ ,  $m_2$  and  $m_4$ .

#### 4-Variable K-Map

There are 16 cells in a 4-variable (w, x, y, z) K-map. The K-map for the same is given as follows:



Every cell thus has 4 neighbours. For example, the cell corresponding to Minterm  $m_0$  has neighbours  $m_1$ ,  $m_2$ ,  $m_4$  and  $m_8$ .

### 1.7.3 Steps for Forming Karnaugh Map

The K-map of a function is easily drawn when the function is given in canonical Sum-of-Products form or Sum-of-Minterms form. When the function is not in the Sum-of-Minterms form, then first convert it to Sum-Of-Products (SOP) form. Expand the SOP expression into Sum-of-Minterms expression or fill in the K-map directly based on the SOP expression.

#### To Summarize:

- Find all the Minterms of the function using the method already discussed.
- Fill '1' for the Minterms in the appropriate location.
- Fill '0' Otherwise.



**Example 1.25:** Draw the K-map for the function F:

$$F(a, b, c) = a.b + b.c' + a'.b'.c$$

**Solution:** Find all the Minterms.

$$\begin{aligned} F(a, b, c) &= a.b(c + c') + b.c'(a + a') + a'.b'.c \\ &= a.b.c + a.b.c' + b.c'.a + b.c'.a' + a'.b'.c \end{aligned}$$

Rearranging the terms with the MSB first and then the next bit up to the LSB, and removing repeated Minterms, you get,

$$F(a, b, c) = a.b.c + a.b.c' + a'.b.c' + a'.b'.c = \Sigma m(1,2,6,7)$$

		b			
		00	01	11	10
a	0	0	1	0	1
	1	0	0	1	1
		c			

**Example 1.26:** The K-map of a 3-variable function F is as follows.

		b			
		00	01	11	10
a	0	1	0	0	1
	1	0	1	0	0
		c			

What is the sum-of-Minterms expression of F?

**Solution:** Assuming that **a** is the MSB and **c** is the LSB and function is of the form  $F(a, b, c)$ , then by seeing the entry of 1, you can say that Minterms are  $m_0$ ,  $m_2$  and  $m_5$ . So,

$$F = \Sigma m(0, 2, 5) = a'.b'.c' + a'.b.c' + a.b'.c$$

### 1.7.4 Simplification of Expressions using Karnaugh Map

Once the K-map for any Boolean expression is known, it can be used to find the minimized expression, which consists of less number of literals. The main advantage of reduction is that it needs less hardware in terms of logic gate. Less number of literals gives realization based on logic gate with less input pin.

The K-map based Boolean reduction is based on the following Unifying Theorem:

$$A + A' = 1$$

NOTES

**NOTES**

In a K-map, each cell containing a '1' corresponds to a Minterm of a given function F. Each group of adjacent cells containing '1' (a group must have size **in powers of two** 1, 2, 4, 8, ...) then corresponds to a simpler product term of F.

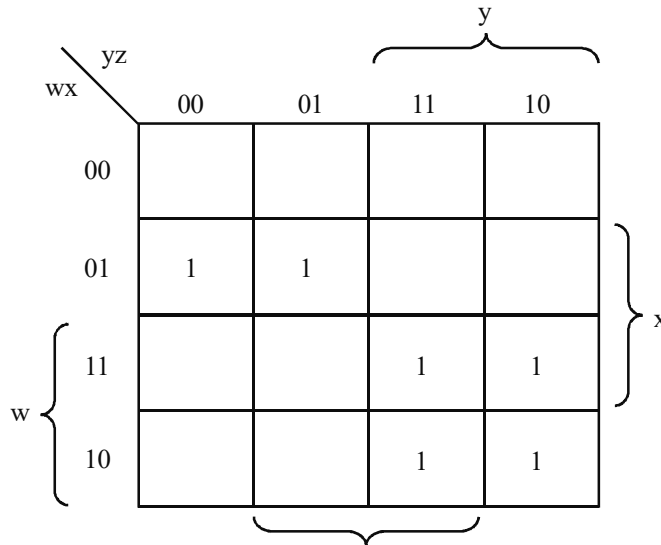
- Grouping 2 adjacent squares eliminates 1 variable, grouping 4 squares eliminates 2 variables, grouping 8 squares eliminates 3 variables, and so on. In general, grouping  $2^n$  squares eliminates n variables.
- Group as many squares as possible. The larger the group, the fewer the number of literals in the resulting product term.
- Select as few groups as possible to cover all the squares (Minterms) of the function. The fewer the groups, the fewer the number of product terms in the minimized function.

**Example 1.27** Find the reduced expression for the function given by:

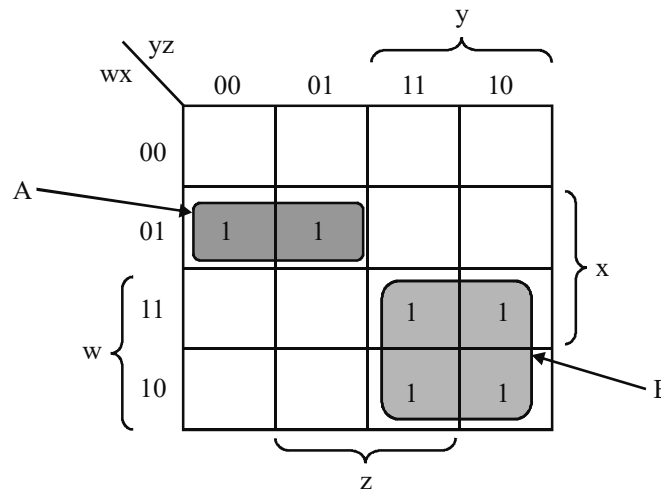
$$F(w,x,y,z) = w'.x.y'.z' + w'.x.y'.z + w.x'.y.z' + w.x'.y.z + w.x.y.z' + w.x.y.z$$

$$= \Sigma m(4, 5, 10, 11, 14, 15)$$

**Solution:** First draw the K-map. Cells with '0' are not shown for clarity.



Each group of adjacent Minterms (group size in powers of twos) corresponds to a possible product term of the given function.



NOTES

There are 2 groups of Minterms: A and B, where:

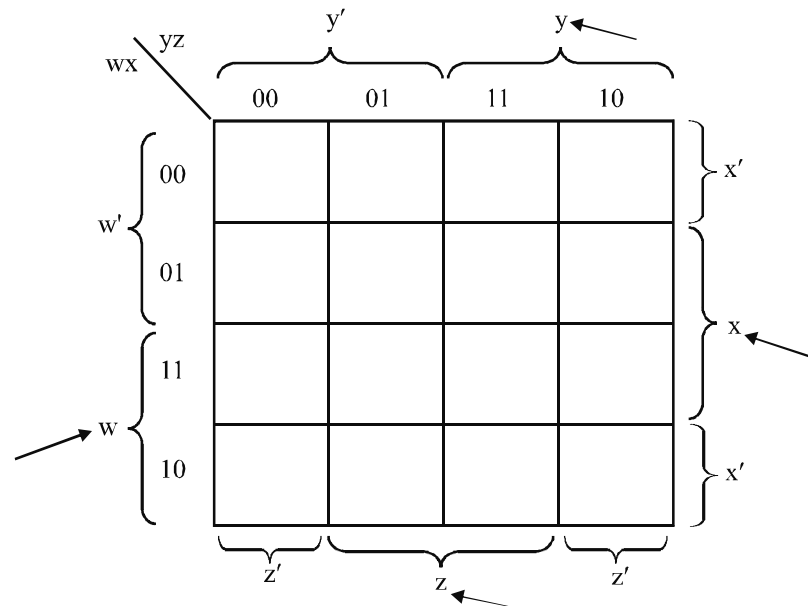
$$\begin{aligned} A &= w'.x.y'.z' + w'.x.y'.z \\ &= w'.x.y'.(z' + z) \\ &= w'.x.y' \end{aligned}$$

$$\begin{aligned} B &= w.x'.y.z' + w.x'.y.z + w.x.y.z' + w.x.y.z \\ &= w.x'.y.(z' + z) + w.x.y.(z' + z) \\ &= w.x'.y + w.x.y \\ &= w.(x' + x).y \\ &= w.y \end{aligned}$$

Each product term of a group,  $w'.x.y'$  and  $w.y$ , represents the **Sum of Minterms** in that group. The Boolean function is, therefore, the Sum-Of-Product (SOP) terms, which represents all groups of the Minterms of the function.

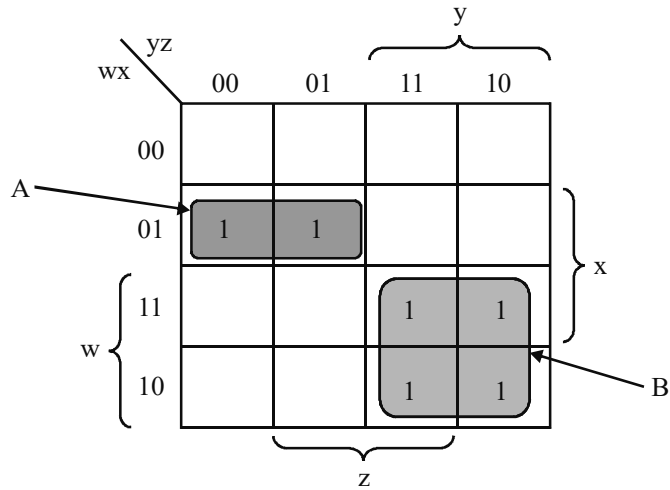
$$F(w,x,y,z) = A + B = w'.x.y' + w.y$$

Another way of getting the expression for the groups A and B is based on the intersection area concept. For example, take a look at four variables of K-map given as follows:



The notation  $w$  pointed to by an arrow shows that the complete region has Minterms in which  $w$  is 1. The region above  $w$  shows  $w'$  region. Similarly, the notation  $x$  pointed by the arrow shows that the complete region is having Minterms in which  $x$  is 1 and the region above and below  $x$  is termed as  $x'$ . The same is true for  $y$  and  $z$ . Using this technique, the K-map shown in the previous example can be solved directly.

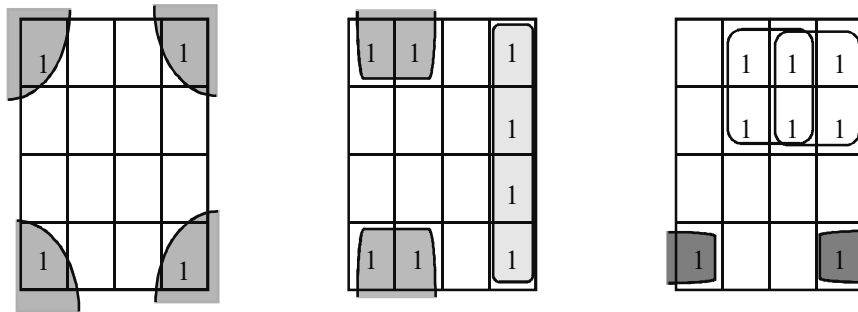
**NOTES**



The intersection area A shows intersection of  $w'$ ,  $x$  and  $y'$ . So, the Boolean expression for the region A can be written as  $w'.x.y'$ . Similarly, region B is the intersection of  $y$ ,  $w$  and the Boolean expression for  $B = w.y$ ; so the overall expression can be written as follows:

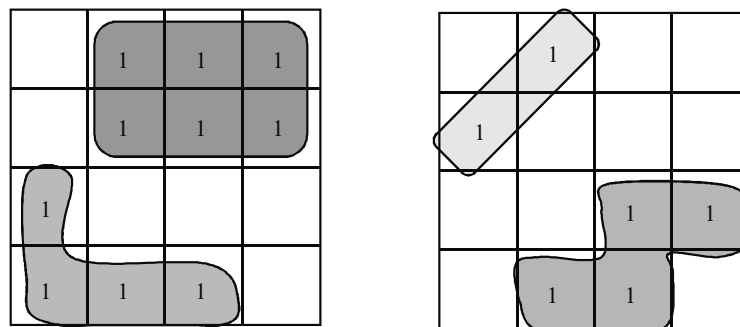
$$F(w,x,y,z) = A + B = w'.x.y' + w.y.$$

Larger groups correspond to product terms of fewer literals. In the case of a 4-variable K-map, if you have 1 cell, then you have 4 literals; if you have 2 cells, then 3 literals; if 4 cells, then 2 literals; if 8 cells, then 1 literal and at last, if 16 cells, then no literal. Also, some other possible valid groupings of a 4-variable K-map are shown as follows:



**1.7.5 Simplification using Karnaugh Map**

Groups of Minterms must be rectangular and must have size in powers of 2's. Otherwise, they are **invalid** groups. Some examples of **invalid** groups are as follows:



The K-map grouping shown above is invalid, since it is not satisfying the rectangular rule or the power of 2 rule.

**Example 1.28:** Find the grouping for the  $F(A,B,C, D) = A(C+D)'(B'+D') + C(B+C'+A'D)$

**Solution:**

**Step 1:** Find the SOP expression for the function.

$$\begin{aligned} F(A,B,C,D) &= A(C+D)'(B'+D') + C(B+C'+A'D) \\ &= A(C'D)'. (B'+D') + BC + CC' + A'D \\ &= AB'C'D' + AC'D' + BC + A'D \end{aligned}$$

**Step 2:** Find all the Minterms by inserting missing variable technique.

$$\begin{aligned} &AB'C'D' + AC'D' + BC + A'D \\ &= AB'C'D' + AC'D'(B+B') + BC + A'D \\ &= AB'C'D' + ABC'D' + AB'C'D' + BC. (A+A') + A'D \end{aligned}$$

After removing the repeated term  $AB'C'D'$ ,

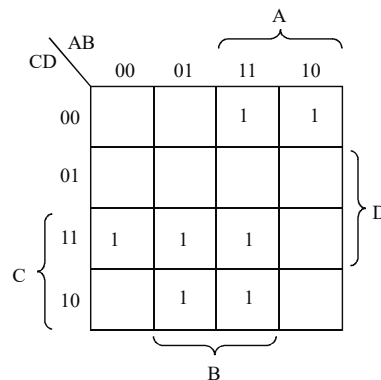
$$\begin{aligned} &= AB'C'D' + ABC'D' + ABC + A'BC + A'D \\ &= AB'C'D' + ABC'D' + ABC. (D+D') + A'BC. \\ &\quad (D+D') + A'D. (B+B') \end{aligned}$$

After removing the repeated term  $A'BCD$ ,

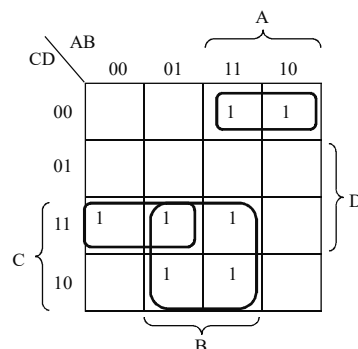
$$\begin{aligned} &= AB'C'D' + ABC'D' + ABCD + ABCD' + A'BCD \\ &\quad + A'BCD' + A'B'CD \\ &= \Sigma m (8, 12, 15, 14, 7, 6, 3) \end{aligned}$$

After rearranging =  $\Sigma m (3, 6, 7, 8, 12, 14, 15)$

**Step 3:** Draw K-map entries.



**Step 4:** Make grouping.



**NOTES**

Using the area intersection concept, the Boolean expression can be written as follows:

$$F(A, B, C, D) = A'.C.D + B.C + A.C'.D'$$

**1.7.6 Simplest SOP Expressions**

To find the simplest possible **SumOfProducts** (SOP) expression from a K-map, you need to obtain:

- Minimum number of literals per product term.
- Minimum number of product terms.

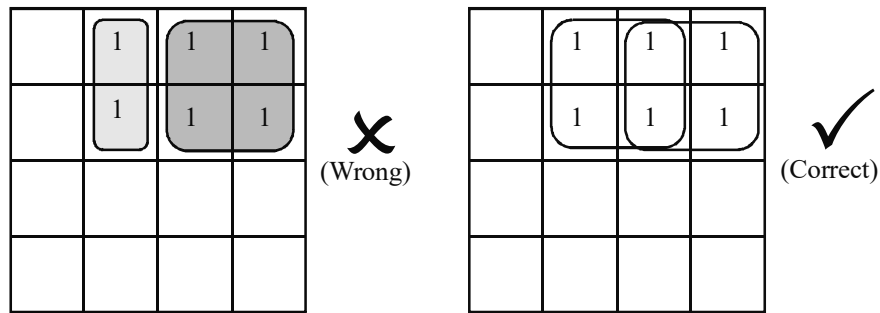
This is achieved in K-map using:

- Bigger groupings of Minterms (prime implicants) where possible.
- No redundant groupings (look for essential prime implicants).

Before learning the definition of prime implicants and essential prime implicants, you need to learn the definition implicant. An implicant is a product term that could be used to cover Minterms of the function.

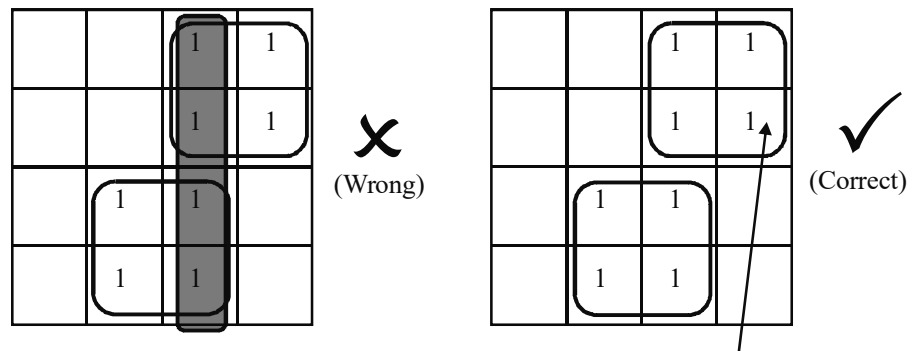
A prime implicant is a product term obtained by combining the maximum possible number of Minterms from adjacent squares in the map. You should use bigger groupings (prime implicants) wherever possible.

As an example, take the following K-maps and the groupings:



The first K-map is one group of four 1's and the other is a group of two, 1's. Since there are two more cells adjacent to that of group 2 cell, so the best covering for the implicant will be the second K-map covering.

An **essential prime implicant** is a prime implicant that includes at least one Minterm that is not covered by any other prime implicant.



Essential Prime Implicants

In the first K-map covering, there are three prime implicants shown by the covering of 4 cells out of which elements of one implicant are covered by either of the two other implicants. This is known as **redundant prime implicant**

The prime implicant, which has at least one element that is not present in any other implicant is known as **essential prime implicant**

**Example 1.29:** Identify the prime implicants and the essential prime implicants of the following K-map:

		bc			
		00	01	11 10	
a	0	1	1	0	1
	1	0	1	0	0
		c		b	

**Solution:** Prime implicants are the covering of Minterms  $\{m_0, m_1\}$ ,  $\{m_1, m_5\}$  and  $\{m_2, m_0\}$

The prime implicants formed by  $\{m_0, m_1\}$  are redundant since both the Minterms are present in other implicants. So, you have only two essential prime implicants, which are formed by covering  $\{m_1, m_5\}$  and  $\{m_2, m_0\}$ .

### How to find the Simplest SOP expression?

The following steps are used for obtaining a simplified SOP expression:

- (i) Circle all prime implicants on the K-map.
- (ii) Identify and select all essential prime implicants for the cover.
- (iii) Select a minimum subset of the remaining prime implicants to complete the cover, that is, to cover those Minterms not covered by the essential prime implicants.

**Example 1.30:** Reduce the following Boolean expression:

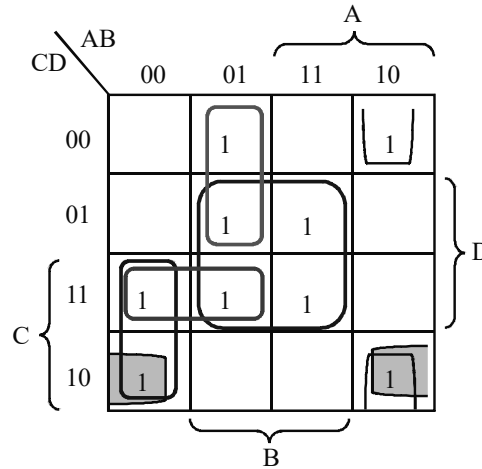
$$F(A,B,C,D) = \sum m(2, 3, 4, 5, 7, 8, 10, 13, 15)$$

**Solution:**

**Step 1:** First draw the K-map.

		A			
		00	01	11 10	
CD	00		1		1
	01		1	1	
C	11	1	1	1	
	10	1			1
		B		D	

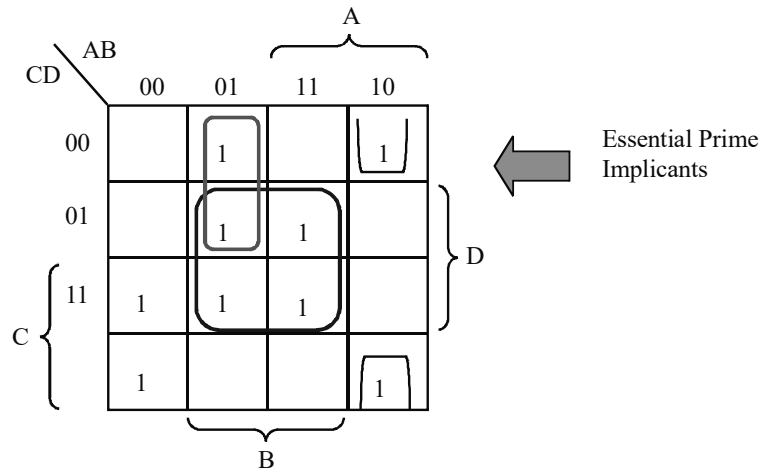
**Step 2: Find prime implicants.**



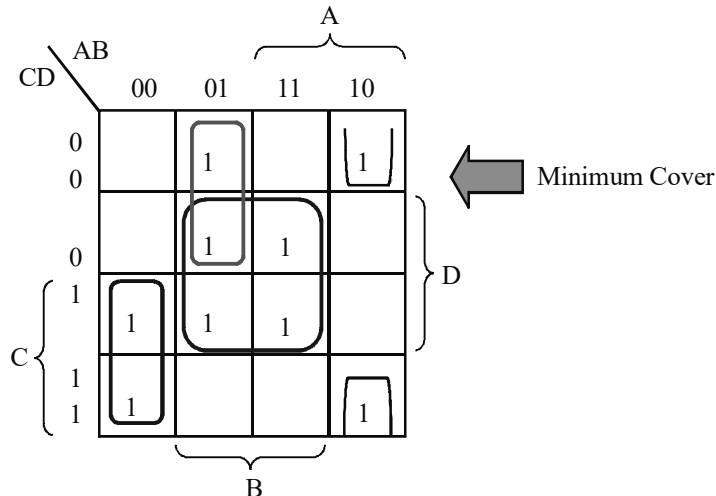
They are the coverings of  $\{m_2, m_3\}$ ,  $\{m_4, m_5\}$ ,  $\{m_3, m_7\}$ ,  $\{m_5, m_7, m_{13}, m_{15}\}$ ,  $\{m_8, m_{10}\}$ ,  $\{m_{10}, m_2\}$ .

Some of the prime implicants are redundant. They can be removed after finding the essential prime implicant.

**Step 3: Find essential prime implicants.**

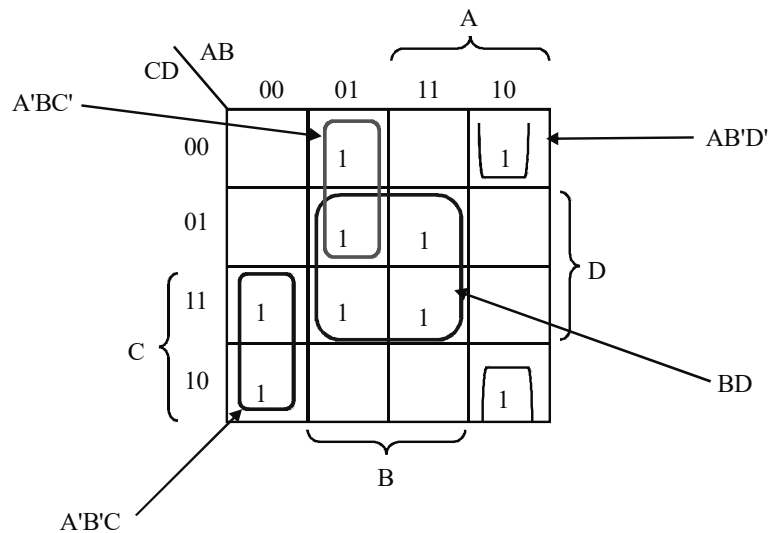


**Step 4: Find minimum cover.** After identifying the essential prime implicant, you have to find the prime implicant necessary for covering all the remaining 1's.





**Step 5:** Write the expression:



$$F(A,B,C,D) = B.D + A'.B'.C + A.B'.D' + A'.B.C'$$

**Example 1.31:** For the function  $F(A,B,C,D) = \sum m(2,3,4,5,7,8,10)$ , find the reduced expression using K-map.

**Solution:** It can have more than one solution. Two are as follows:

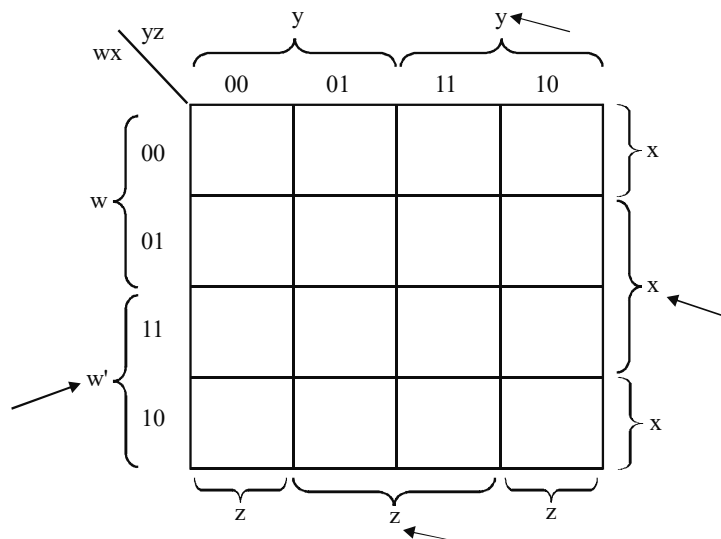
- (i)  $F = A'.B.C' + A'.B.D + A.B'.D' + A'.B'.C$
- (ii)  $F = A'.B.C' + A'.C.D + A.B'.D' + A'.B'.C$

**Example 1.32:**  $F(A,B,C,D) = A.B.C + B'.C.D' + A.D + B'.C'.D'$ , find the reduced expression using K-map.

**Solution:**  $F(A,B,C,D) = A.D + A.C + B'.D'$

### 1.7.7 Getting POS Expressions

Simplified Product of Sum (POS) expressions can be obtained by grouping the Maxterms (i.e., 0s) of the given function. As far as covering of 0's is concerned, it is done in the same fashion as was used for covering 1's. However, during writing the expression for the essential prime implicant or prime implicant, a different method is followed. Using the area intersection concept, the grouping for the 4-variable K-map for Maxterms will be as follows:

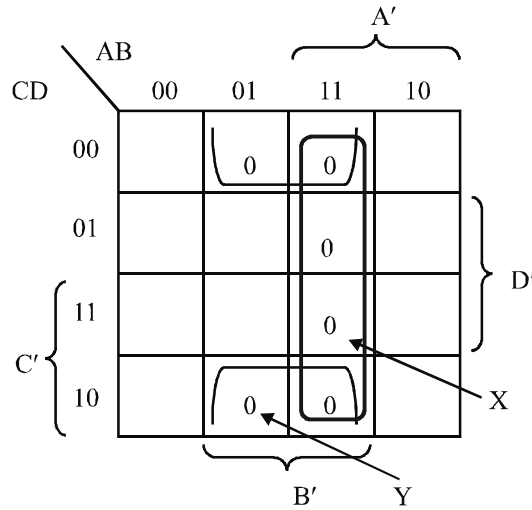


**NOTES**

The notation  $w'$  pointed to by an arrow shows that the complete region is having Maxterm in which  $w$  is 0. The region above  $w'$  shows  $w$  region. The same is true for  $x, y$  and  $z$  also. During writing expression for the covering of 0's, the intersection area is written as sum term as opposite to the product term that was used during grouping of 1's.

**Example 1.33:** Given  $F = \prod m(4, 6, 12, 13, 14, 15)$ , draw the K-map, and write reduced Maxterm expression. The grouping of 0's is as follows:

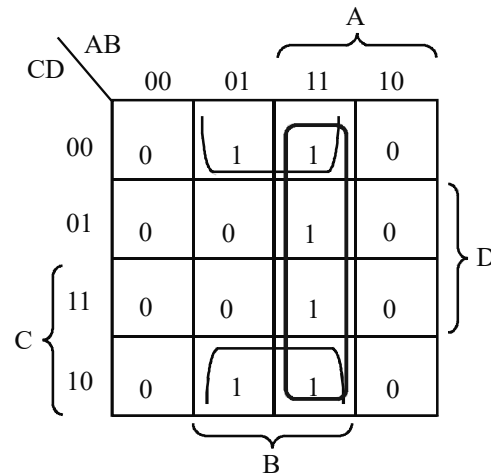
**Solution:**



The two terms, namely,  $X$  and  $Y$  can be written using the area intersection concept as  $X = A' + B'$  and  $Y = B' + D$ ; So, overall expression can be written as  $F = X \cdot Y = (A' + B') \cdot (B' + D)$ .

Another method for getting the minimized expression in the Maxterm form is to find the reduced Minterms expression first and then take the complement for getting the Maxterm expression.

Find the expression for the  $F'$  (complement of  $F$ ) by using K-map of  $F'$ .



This gives the SOP of  $F'$  to be:  $F' = B.D' + A.B$

To get POS of  $F$ , you have to take complement of  $F'$ :

$$\begin{aligned} F &= (F')' = (B.D' + A.B)' \\ &= (B.D')' . (A.B)' \\ &= (B'+D) . (A'+B') \end{aligned}$$

**Example 1.34:** Find the simplest POS expression for the following function:

$$F(A,B,C,D) = A.B.C + B'.C.D' + A.D + B'.C'.D'$$

**Solution:**  $F(A,B,C,D) = (A+B').(A+D').(B'+C+D)$

**Hint:** Draw the K-map of the complement of  $F$ ,  $F'$  and then using De Morgan theorem, find the POS.

### Check Your Progress

10. What is Boolean expression simplification techniques?
11. How the combinational circuits can be defined?
12. Define the term Binary addition.

## 1.8 COMBINATIONAL CIRCUITS

Combinational Circuits (CC) are those circuits where output depends on the present value of the inputs. If input values are changed, the information about the previous inputs is lost because combinational logic circuits have no memory. In such cases, sequential logic circuits are used to overcome this problem. In a **combinational logic** circuit the outputs depend on their current inputs. Combinational circuits are used to realize Boolean expressions.

### 1.8.1 Half-Adder

An electronic (combinational) circuit which performs the arithmetic addition of two binary digits is called a half-adder.

The half-adder has two inputs (augend and addend) and two outputs (sum and carry). The logic symbol for a half-adder is shown in the Figure 1.10(a). The logic diagram that consists of an XOR gate and an AND gate is shown in the Figure 1.10(b).

The half-adder functions is according to the Truth Table. You know that the AND gate produces a high output only when both inputs are high and the exclusive OR gate produces a high output if either input, but not both half-adder is high. From the Truth Table, the sum output corresponds to a logic EXOR function, while the carry output corresponds to AND function.

NOTES

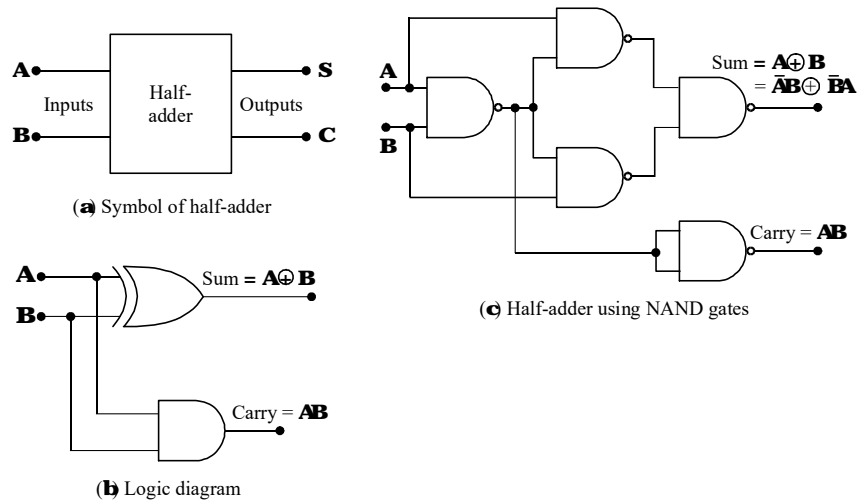


Fig. 1.10 Half-adder Logic Symbol and Diagram

Truth Table 1.14 Half-adder

Inputs		Outputs	
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Examine each entry in the Truth Table 1.14. Half-adder does electronically what you do mentally, when we add two bits.

- First Entry:** Input : **A= 0 and B= 0**  
 Human response : 0 plus 0 is 0 with a carry of 0  
 Half-adder response : SUM = 0 and CARRY = 0
- Second Entry:** Input : **A= 1 and B= 0**  
 Human response : 1 plus 0 is 1 with a carry of 0.  
 Half-adder response : SUM = 1 and CARRY = 0
- Third Entry:** Input : **A= 1 and B= 1**  
 Human response : 0 plus 1 is 1 with a carry of 0.  
 Half-adder response : SUM = 1 and CARRY = 0
- Fourth Entry:** Input : **A= 1 and B= 1**  
 Human response : 1 plus 1 is 0 with a carry of 1.  
 Half-adder response : SUM = 0 and CARRY = 1

The SUM output represents the Least Significant Bit (LSB) of the sum. The Boolean expression for the two outputs can be obtained directly from the Truth Table 1.14.

$$S_{(\text{sum})} = \overline{A}B + A\overline{B} = (A + B)(\overline{A} + \overline{B}) = A \oplus B$$

$$C_{(\text{carry})} = AB = (A + B)(A + \overline{B})(\overline{A} + B)$$

The implementation of the half-adder circuit using basic gates is shown in the Figure 1.11.

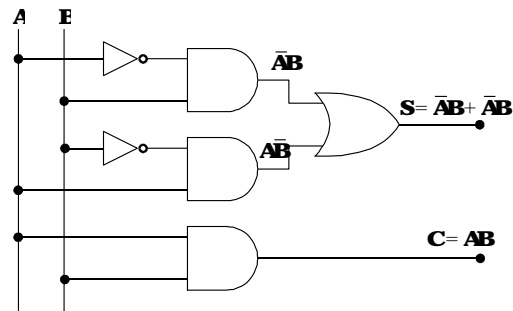


Fig. 1.11 Half Adder Circuit

### 1.8.2 Full-Adder

A half-adder has only two inputs and there is no provision to add a carry coming from the lower order bits when multibit addition is performed. For this purpose, a third input terminal is added and this circuit is used to add **A**, **B** and **C<sub>in</sub>**.

Truth Table 1.15 Full-Adder

Inputs			Outputs	
AugendBit <b>A</b>	AddendBit <b>B</b>	CarryBit <b>C<sub>in</sub></b>	SumBit <b>S</b>	CarryBit Output <b>C<sub>out</sub></b>
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

The logic expression of exclusive ORing of three variables **A**, **B** and **C<sub>in</sub>** is as follows:

$$\begin{aligned}
 \mathbf{A} \oplus \mathbf{B} \oplus \mathbf{C} &= (\overline{\mathbf{A}\mathbf{B}} + \mathbf{A}\overline{\mathbf{B}}) \oplus \mathbf{C}_{in} \\
 &= \overline{(\overline{\mathbf{A}\mathbf{B}} + \mathbf{A}\overline{\mathbf{B}}) \mathbf{C}_{in}} + \overline{\mathbf{C}_{in}} (\overline{\mathbf{A}\mathbf{B}} + \mathbf{A}\overline{\mathbf{B}}) \\
 &= (\overline{\mathbf{A}\mathbf{B}}) \cdot (\overline{\mathbf{A}\mathbf{B}}) \mathbf{C}_{in} + \overline{\mathbf{C}_{in}} (\overline{\mathbf{A}\mathbf{B}} + \mathbf{A}\overline{\mathbf{B}}) \\
 &= (\mathbf{A} + \overline{\mathbf{B}}) \cdot (\overline{\mathbf{A}} + \mathbf{B}) \mathbf{C}_{in} + \overline{\mathbf{C}_{in}} (\overline{\mathbf{A}\mathbf{B}} + \mathbf{A}\overline{\mathbf{B}})
 \end{aligned}$$

$$\text{SUM} = \mathbf{A} \oplus \mathbf{B} \oplus \mathbf{C}_{in} = \overline{\mathbf{A}\mathbf{B}\mathbf{C}_{in}} + \overline{\mathbf{A}\mathbf{B}}\overline{\mathbf{C}_{in}} + \overline{\mathbf{A}\mathbf{B}\mathbf{C}_{in}} + \overline{\mathbf{A}\mathbf{B}\mathbf{C}_{in}}$$

For **A**= 1, **B**= 0 and **C<sub>in</sub>** = 1,

$$\begin{aligned}
 \mathbf{S} &= \overline{1 \cdot 0 \cdot 1} + \overline{1 \cdot 0 \cdot 1} + \overline{1 \cdot 0 \cdot 1} + \overline{1 \cdot 0 \cdot 1} \\
 &= 0 \cdot 1 \cdot 1 + 0 \cdot 0 \cdot 0 + 1 \cdot 1 \cdot 0 + 1 \cdot 0 \cdot 1 = 0
 \end{aligned}$$

NOTES

The sum of products for,

$$\begin{aligned} C_{out} &= \bar{A}BC_{in} + A\bar{B}C_{in} + AB\bar{C}_{in} + ABC_{in} + \bar{A}BC_{in} + A\bar{B}C_{in} \\ &= \bar{A}BC_{in} + A\bar{B}C_{in} + \bar{A}BC_{in} + A\bar{B}C_{in} + \bar{A}BC_{in} + A\bar{B}C_{in} \\ &= BC_{in}[\bar{A} + A] + AC_{in}[\bar{B} + B] + AB[\bar{C}_{in} + C_{in}] \\ C_{out} &= BC_{in} + AC_{in} + AB = AB + BC_{in} + C_{in}A \end{aligned}$$

For  $A=1$ ,  $B=0$  and  $C_{in}=1$ ,  $C_{out} = 1.0 + 0.1 + 1.1 = 1$

### 1.8.3 Parallel Binary Adder

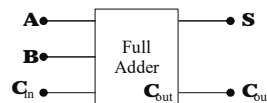
In most logic designs, more than one bit words are added. The addition of multibit numbers is accomplished by using several full-adders. Figure 1.12 shows the block diagram of a 5-bit parallel adder using 5 full-adder circuits connected in series, i.e., the carry output of each adder is connected to the carry input of the next higher order adder. Let the 5-bit words to be added be represented by  $A_4 A_3 A_2 A_1 A_0 = 11111$  and  $B_4 B_3 B_2 B_1 B_0 = 00011$ .

A full-adder is a combinational circuit that performs the arithmetic sum of three input bits and producing a sum and a carry.

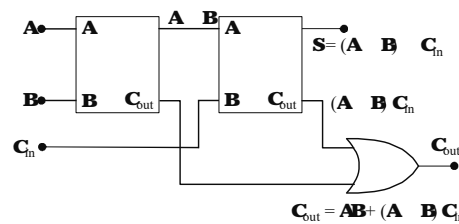
It consists of three inputs and two outputs. Two inputs variables denoted by  $A$  and  $B$  represent the carry from the previous lower significant position. Two outputs are necessary because the arithmetic sum of three binary digits ranges from 0 to 3 and binary 2 or 3 needs two digits. The outputs are designed by the symbol  $S$  (for sum) and  $C_{out}$  (for carry). The binary variable  $S$  gives the value of the LSB of the sum. The binary variable  $C_{out}$  gives the output carry.

The symbolic diagram for full-adder is shown in Figure 1.12 (a). A full-adder is formed by using two half-adder circuits and an OR gate as shown in Figure 1.12 (b). Note the symbol  $\Sigma$  (sigma) for the sum. The full-adder circuit which consists of three AND gates, an OR gate and 3-input exclusive OR gate is shown in Figure 1.12 (c).

Truth Table 1.15 shows the Truth Table of a full-adder. There are several possible cases for the three inputs and for each case the desired output values are listed. For example, consider the case  $A=1$ ,  $B=0$  and  $C_{in}=1$ . The full-adder must add these bits to produce a sum ( $S$ ) of 0 and carry ( $C_{out}$ ) of 1. The reader should check the other cases to understand them. The full-adder can do more than a million additions per second.

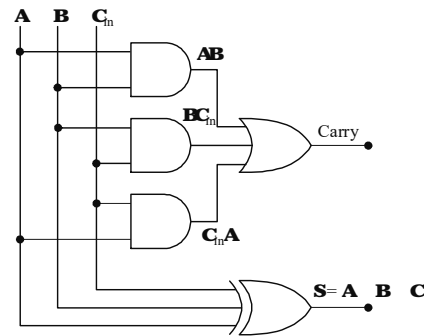


(a) Logic Symbol of Full-Adder

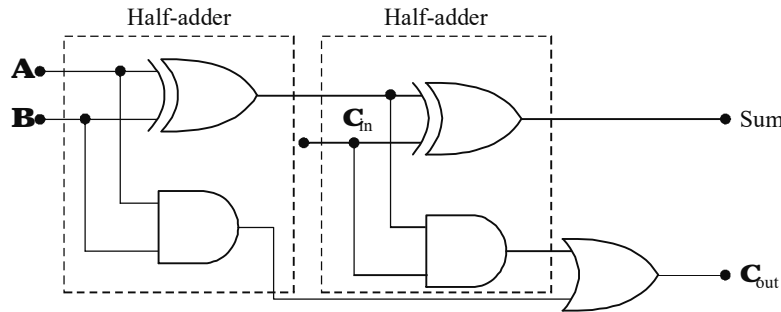


(b) Full-Adder using Two Half-Adders

NOTES



(c) Logic Circuit of Full-Adder



(d) Full-Adder using Two Half-Adders and an OR Gate

Fig. 1.12 Block Diagram of a 5 Bit Parallel Adder

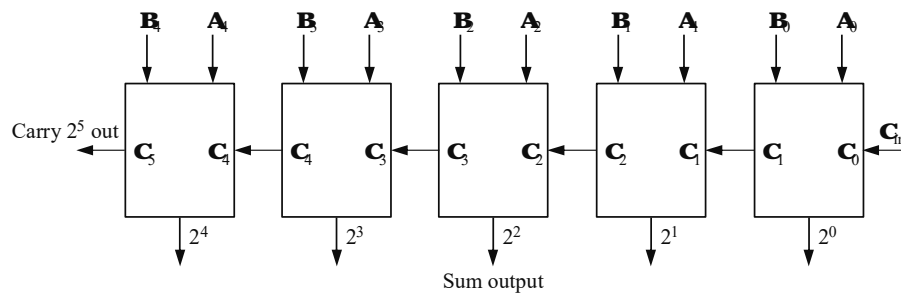


Fig. 1.13 A Parallel 5 Bit Binary Adder

Significant place	:	5	4	3	2	1	0
Input carry	:		1	1	1	1	0
Augend word <b>A</b>	:		1	1	1	1	1
Addend word <b>B</b>	:		0	0	0	1	1
Sum	:		0	0	0	1	0
Output carry	:	1	1	1	1	1	1

In this circuit, the output carry of the lower order is the input carry for the next higher order. Hence, this type of adder is called **ripple carry adder**. Since all the bits of the augend and addend are fed into the adder circuits simultaneously, this circuit is known as **parallel adder circuit**. This means that additions in each position are taking place at the same time.

The same basic configuration illustrated above may be extended to any number of binary bits.

NOTES

It should be noted that either a half-adder can be used for the least significant position or the carry input of a full-adder is made 0 because there is no carry into the least significant bit position.

4-Bit Parallel Binary Adder

IC manufacturers produce several adders one elementary arithmetic IC is the TTL 7483, a 4-bit binary full-adder. A logic symbol for the 7483 IC is shown in Figure 1.14. It has two 4-bit inputs **A<sub>3</sub>A<sub>2</sub>A<sub>1</sub>A<sub>0</sub>** and **B<sub>3</sub>B<sub>2</sub>B<sub>1</sub>B<sub>0</sub>** and a carry input **C<sub>in</sub>** in the LSB stage. The outputs are a 4-bit sum **S<sub>3</sub>S<sub>2</sub>S<sub>1</sub>S<sub>0</sub>** and a carry output **C<sub>out</sub>** from the most significant bit stage. For adding just two 4-bit numbers, the **C<sub>in</sub>** input is held at 0. The carry output **C<sub>out</sub>** is attached to the 16s output indicator. This binary adder can indicate a sum as high as 11110 (decimal 30) when adding binary 1111 to 1111.

The 7483 IC adder can be cascaded by connecting **C<sub>out</sub>** output of the first IC to the carry input **C<sub>in</sub>** of the next 7483 IC. With two 7483 IC adders cascaded, an 8-bit binary adder is produced. The 7483 IC can also be used as a 4-bit subtractor.

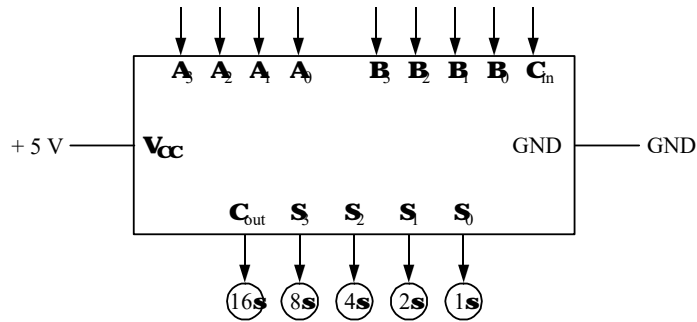


Fig. 1.14 The 7483 4-Bit Binary Adder IC

Serial Adder

Though the parallel adder performs the addition of two binary numbers at a relatively fast rate, the disadvantage of the parallel addition is that it requires a large amount of logic circuitry. This increases indirect proportion to the number of bits in the numbers being added. In serial addition, the addition process is carried out in a manner in which you perform addition on paper, that is one position at a time. This results in much simpler circuitry than parallel addition but results in a much smaller speed of operation.

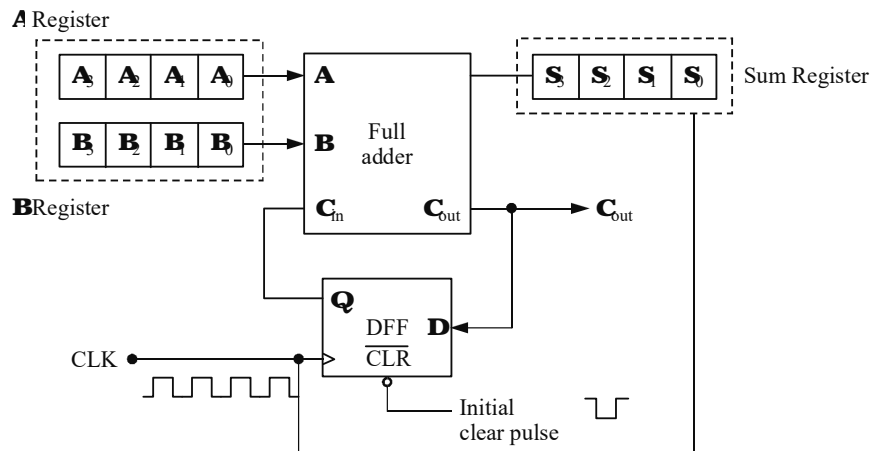


Fig. 1.15 4-Bit Serial Adder



## NOTES

The diagram of a 4-bit serial adder is shown in Figure 1.15. The registers **A** and **B** are used to store the numbers to be added. The **D** flip-flop is required for storage of a possible carry to the next cycle. However, in the serial adder these registers are shift registers whose binary values shift from left to right upon application of each clock pulse. The outputs (LSBs) **A**<sub>0</sub> and **B**<sub>0</sub> are fed into a single full-adder along with the output of the carry FF. The carry FF is a separate FF used to store the carry output of the FA so that it can be added to the next significant position of the numbers in the registers.

The SUM output, **A**<sub>3</sub>, of the FA is fed to the **D** input of the MSB of the **A** register. As soon as the clock pulse is occurred the SUM values is transferred into **A**<sub>3</sub>. The output **B**<sub>0</sub> is connected to the **D** input of **B**<sub>3</sub>, so **B**<sub>3</sub> takes on the value of **B**<sub>0</sub> when a clock pulse occurs. In this way the contents of **B** register will be unchanged after all the shifting operations are completed.

Understand the operation of serial adder by following through a complete cycle. Let the augend be 0111 and the addend be 0010 in registers **A** and **B** respectively. Also, you will assume that the carry FF has been initially cleared to the 0 state, so CARRY-IN. Refer to the first entry in Figure 1.16 which shows the various logic levels before the first clock pulse is applied.

### First Clock Pulse

Since **A**<sub>0</sub> = 1, **B**<sub>0</sub> = 0 and CARRY-IN = 0, the full-adder outputs will be SUM = 1 and CARRY-OUT = 0. These levels are present at the full-adder outputs before the first clock pulse occurs. When the first Clock Pulse occurs, the values in the **A** register shift from left to right one bit, as do the values in the **B** register. In addition, the SUM level is transferred into **A**<sub>3</sub>, the **B**<sub>0</sub> level is transferred into **B**<sub>3</sub>, and the CARRY-OUT level is transferred to the carry FF, whose output becomes CARRY-IN = 0.

### Second Clock Pulse

At this point, **A**<sub>0</sub> and **B**<sub>0</sub> contain the bits (1 and 1) that were in the second position of the original augend and added. These are fed to the full-adder along with CARRY-IN (0) from CARRY-OUT = 1. When the second Clock Pulse occurs, the **A** and **B** registers again shift right, SUM = 0 is transferred to **A**<sub>3</sub>, and CARRY-OUT = 1 is transferred to the CARRY FF.

### Third Clock Pulse

The values of **A**<sub>0</sub> and **B**<sub>0</sub> are now 1 and 0 respectively and CARRY-IN is 1. These values produce SUM = 0 and CARRY-OUT = 1 at the full-adder outputs. On the occurrence of the third Clock Pulse, the **A** and **B** registers again shift right, SUM = 0 goes to **A**<sub>3</sub>, and CARRY-OUT = 1 goes to the carry FF.

### Fourth Clock Pulse

**A**<sub>0</sub> and **B**<sub>0</sub> are now both 0 and CARRY-IN = 1, the FA produces SUM = 1 and CARRY-OUT = 0. The fourth Clock Pulse transfers SUM = 1 into **A**<sub>3</sub> and initiates all the other transfers. At the completion of this fourth Clock Pulse the **A** register holds the number 1001, which of course, is the SUM of the original augend and addend. In addition, the **B** register holds the original addend 0010. The addition process is now complete.

NOTES

There are presently six complex function adder packages in the series SN54/74 logic for 1, 2 and 4 bits. The SN54/7480 is a single bit gate full-adder with gated complementary inputs and complementary sum ( $\Sigma$  and  $\bar{\Sigma}$ ) outputs. The inverted carry output is designed for medium and high speed multiple bit parallel adder for serial carry applications. It is designed specifically for multibit addition or subtraction operations without external gates or inverters.

The SN54/77482 and SN7483 are 2- and 4-bit binary adders respectively that perform parallel addition with internally connected ripple through (serial) carry. The basic logic configuration for the sum and carry is the same as for the SN7480, but there is no gating for inputs on the SN7482 and SN7483 adders. The complement of the carry out is used for the carry input of high order bits. The consequence is that for even numbered bits, it is required to invert the inputs and for even numbered sum outputs, an inverter is not required. Table 1.16 summarizes the differences between parallel adder and serial adder.

Table 1.16 Comparison of Parallel Adder and Serial Adder

Serial Adder	Parallel Adder
1. Serial adder is less faster.	1. Parallel adder is generally faster.
2. It requires more components.	2. It requires less components compared to serial adder.
3. Only the right most FFs of these registers can be added at one time.	3. All the outputs of the register are available for addition at the same time.
4. The bit positions are processed one at a time.	4. All the bit positions are processed simultaneously.
5. It would require 36 clock pulses to add two 36-bit numbers. It takes nearly $36 \times 50 = 1800$ ns to perform addition.	5. It produces the desired output only after a short delay. A parallel adder can add two 56-bit numbers in about 100 ns.
6. It requires only one full-adder for any number of bits.	6. It requires 36 full-adders to add two 36-bit numbers.

Look Ahead Carry Generator

In the case of parallel adder, the speed with which an addition can be performed is limited by the time required for the carries to propagate or ripple through all of the stages of the adder. One method of speeding up this process by eliminating the ripple carry delay is called **look ahead carry addition**. This method is based on two functions of the full-adder called the carry generate and the carry propagate functions.

The carry generate ( $G_i$ ) function indicates when an output carry is generated by the full-adder. We know that a carry is generated only when both inputs are 1s. This condition is expressed as the AND function of the two inputs, **A** and **B** i.e.,

$$G_i = AB$$

A carry input may be propagated by the full adder when either or both  $r_0$  the input bits are 1s. This condition is expressed as the OR function of the input bits:

$$P_i = A + B$$

Consider the circuit of full-adder shown in Figure 1.16. The carry generate and carry propagate are given by,

$$P_i = A_i \oplus B_i$$

and

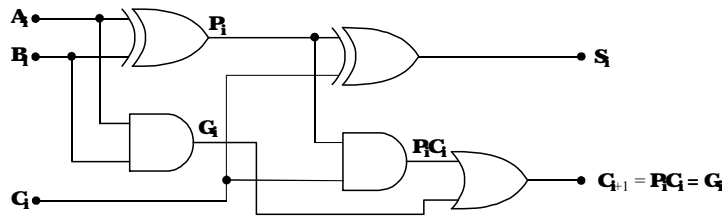
$$G_i = AB_i$$

The output sum and carry can be expressed as,

$$S_i = P_i \oplus C_i = A_i + B_i \oplus C_i$$

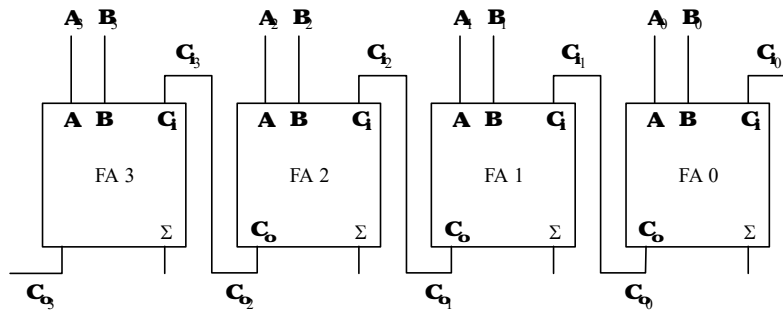
and

$$C_{i+1} = G_i + P_i C_i$$



**Fig. 1.16 Full-Adder Circuit**

Consider the addition of two 4-bit binary numbers  $A_3A_2A_1A_0$  and  $B_3B_2B_1B_0$ . For each full-adder, the output carry is dependent on its carry generate ( $G_i$ ), its carry propagate ( $P_i$ ) and its carry input ( $C_i$ ). The  $G_i$  and  $P_i$  functions for each stage are immediately available as soon as the input bits  $A_i$  and  $B_i$  and the input carry to the LSB adder are applied, because they are dependent only on these bits. The carry input to each stage is the carry output of the previous stage.



**Fig. 1.17 Carry Generate and Carry Propagate Functions in Terms of the Input Bits to a 4 Bit Adder**

**For Full-Adder 0:**  $C_0 = G_0 + P_0 C_{i0}$  where  $G_0 = A_0 B_0$ ,  $P_0 = A_0 \oplus B_0$  and  $C_{i0} = 0$  ... (1.13)

**For Full-Adder 1:**  $C_{i1} = C_{\infty}$   
 $C_{i1} = G_1 + P_1 C_{i1} + G_1 + P_1 C_{00}$   
 $= G_1 + P_1 (G_0 + P_0 C_{00})$   
 $C_{01} = G_1 + P_1 G_0 + P_1 P_0 C_{00}$  ... (1.14)

where,  $G_1 = A_1 B_1$  and  $P_1 = A_1 \oplus B_1$

**For Full-Adder 2:**  $C_{i2} = C_{01}$   
 $C_{02} = G_2 + P_2 C_{01}$   
 $= G_2 + P_2 (G_1 + P_1 G_0 + P_1 P_0 C_{00})$   
 $C_{02} = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{00}$  ... (1.15)

where,  $G_2 = A_2 B_2$  and  $P_2 = A_2 \oplus B_2$

**For Full-Adder 3:**  $C_{i3} = C_{02}$   
 $C_{03} = G_3 + P_3 C_{02}$   
 $= G_3 + P_3 (G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{00})$   
 $C_{03} = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_{00}$  ... (1.16)

NOTES

NOTES

where,

$$G_3 = A_3 B_3 \text{ and } P_3 = A_3 \oplus B_3$$

The sum of **A** and **B** is given by,

$$S = C_0 S_3 S_2 S_1 S_0 \quad \dots(1.17)$$

where,

$$S_i = A_i \oplus B_i \oplus C_{i-1} \text{ for } i = 0, 1, 2, 3$$

i.e.,

$$S_0 = A_0 \oplus B_0 \oplus C_0$$

$$S_1 = A_1 \oplus B_1 \oplus C_{01}$$

$$S_2 = A_2 \oplus B_2 \oplus C_{02}$$

$$S_3 = A_3 \oplus B_3 \oplus C_{03}$$

From the above equations, it is seen that the carry output for each full-adder stage is dependent only on the initial input carry, its generate and propagate functions and the generate and propagate functions of the preceding stage.

Equations (1.13) through (1.17) can be implemented with logic gates as shown in Figure 1.18. From the diagram one can easily understand that the addition of two 4-bit numbers can be done by a carry look ahead adder in a four gate propagation time. Note that the addition of **n** bit binary numbers also takes the same four gate propagation delay.

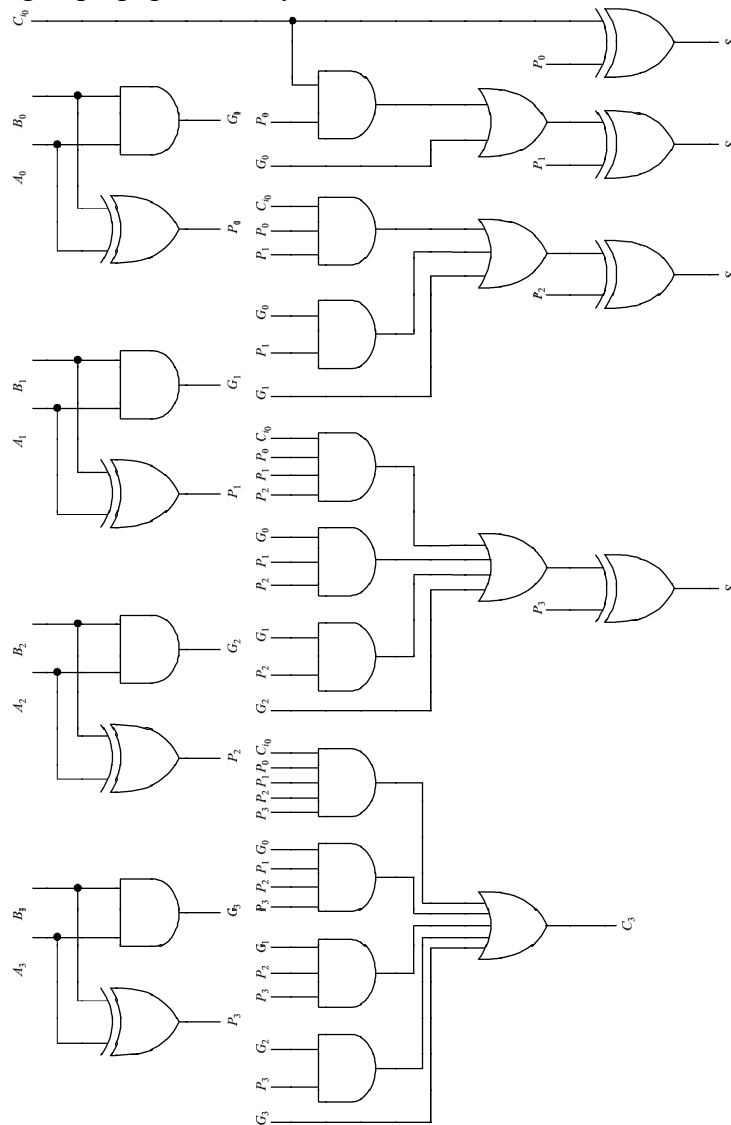


Fig. 1.18 Logic Diagram of 4 Bit Carry Look Ahead Adder

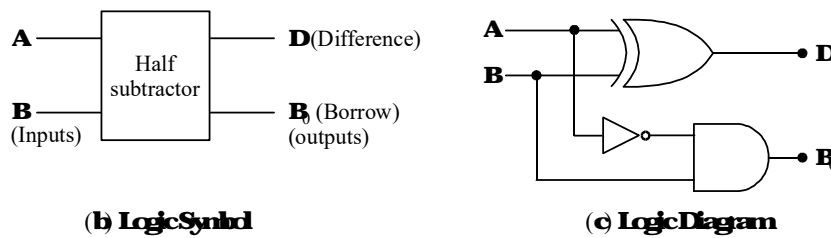
## 1.8.4 Subtractors

### Half-Subtractor

Just like adders there can be single bit half-subtractors, full-subtractors and parallel subtractors. Binary subtraction tables are shown in Figure 1.19(a). Converting these rules to Truth Table form gives the Truth Table 1.17. **B** is subtracted from **A** to give output **D** difference. If **B** is larger than **A**, such as in line 2 they need a borrow, which is shown in the column labeled **B<sub>0</sub>** (Borrow out).

0	0	1	1	1
-0	-1	borrow 1	-0	-1
0	1		1	0

(a) Binary Subtraction Table



(b) Logic Symbol

(c) Logic Diagram

Fig. 1.19 Half Subtractor

A half-subtractor is a combinational circuit that subtracts two binary bits and produces their difference.

A logic symbol of a half-subtractor is shown in Figure. 1.19(b). It has two inputs (minuend **A** and subtrahend **B**) and two outputs (difference) **D** and **B<sub>0</sub>** (borrow). The subtraction of two binary numbers may be accomplished by taking the complement of the subtrahend and adding to the minuend. By this procedure the subtraction becomes an addition operation. By looking at the Truth table, you can determine the Boolean expressions for the half-subtractor. The expression for the **D** (difference column) is  $A \oplus B = D$  Boolean expression for the **B** (borrow column) is  $\bar{A}B = B_0$ . Combining these two expressions, the half-subtractor can be implemented using an XOR gate, a NOT gate and an AND gate as shown in Figure 1.19(c).

Truth Table 1.17 Half Subtractor

Inputs		Outputs	
A	B	D	B <sub>0</sub>
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

NOTES

Full-Subtractor

A full-subtractor is a combinational circuit that performs a subtraction involving three bits namely minuend bit, subtrahend bit and borrow from the previous stage.

The logic symbol for full-subtractor is shown in Figure 1.20(a). It has three inputs and two outputs. The three inputs are **A**(minuend), **B**(subtrahend) and **B<sub>in</sub>** (borrow from previous stage). The two outputs are **D**(difference) and **B<sub>out</sub>** (borrow out). A Truth Table that considers all the possible combinations in binary subtraction is shown in the next Truth Table. Like the full-adder, the full-subtractor can be wired using two half-subtractors and an OR gate, shown in Figure 1.20(b). A logic diagram for a full-subtractor using XOR, AND and NOT gates is shown in Figure 1.20(c). This circuit performs as a full-subtractor as specified in the Truth Table 1.18.

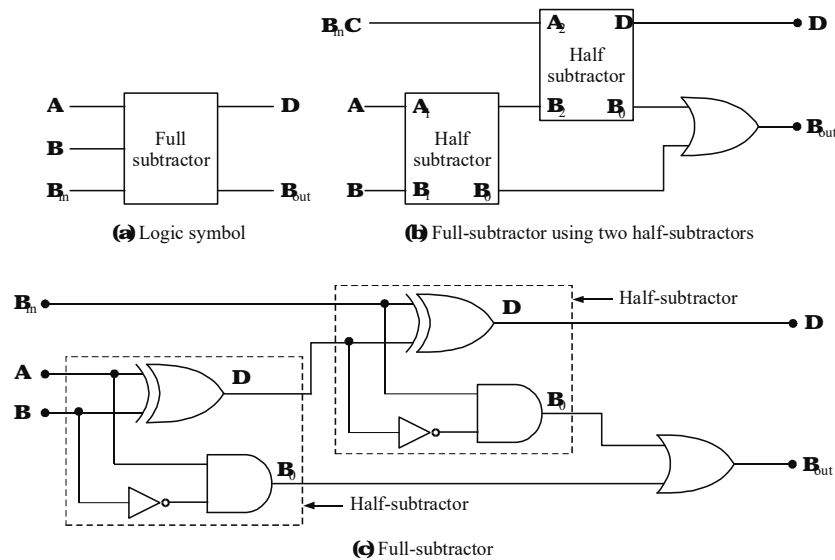


Fig. 1.20 Full-Subtractor

From Truth Table 1.18, the sum of product expression for the difference (**D**) output can be written as,

$$D = \overline{A}B\overline{B}_m + A\overline{B}\overline{B}_m + \overline{A}B\overline{B}_m + AB\overline{B}_m$$

Simplifying, we have

$$\begin{aligned} D &= (\overline{A}B + AB)\overline{B}_m + (\overline{A}B + AB)B_m \\ &= (A \oplus B)\overline{B}_m + (A \oplus B)B_m \\ D &= A \oplus B \oplus B_m \end{aligned}$$

Similarly, the sum of product expression for **B<sub>out</sub>** can be written as,

$$B_{out} = \overline{A}B\overline{B}_m + \overline{A}B\overline{B}_m + \overline{A}B\overline{B}_m + AB\overline{B}_m$$

The above equation can be simplified using K-map as shown in Figure 1.21.

Truth Table 1.18 Full-Subtractor

Inputs			Outputs	
A	B	B <sub>in</sub>	D	B <sub>out</sub>
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

NOTES

With the simplified Boolean expressions, the full-subtractor can be implemented as shown in Figure 1.22.

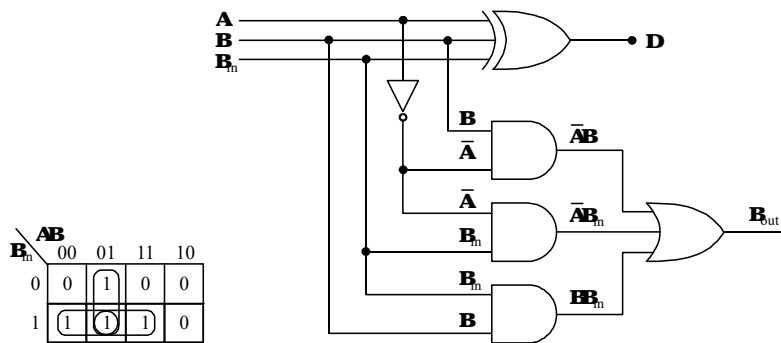


Fig. 1.21 KMap for B<sub>out</sub>

Fig. 1.22 Implementation of Full-Subtractor

### 1.8.5 Decoders

Many digital systems require the decoding of data. Decoding is necessary in such applications as data multiplexing, rate multiplying, digital display, digital-to-analog converters and memory addressing. It is accomplished by matrix systems that can be constructed from such devices as magnetic cores, diodes, resistors, transistors and FETs.

A decoder is a combinational logic circuit, which converts binary information from **n** input lines to a maximum of 2<sup>n</sup> unique output lines such that each output line will be activated for only one of the possible combinations of inputs. If the **n** bit decoded information has unused or don't care combinations, the decoder output will have fewer than 2<sup>n</sup> outputs.

A decoder is similar to demultiplexer, with one exception there is no data input.

A single binary word **n** digits in length can represent 2<sup>n</sup> different elements of information.

An AND gate can be used as the basic decoding element because its output is HIGH only when all of its inputs are HIGH. For example, the input binary is 1011. In order to make sure that all of the inputs to the AND gate are HIGH when binary number 1011 occurs, then the third bit (0) must be inverted.

**NOTES**

If a NAND gate is used in place of the AND gate, a LOW output will indicate the presence of the proper binary code.

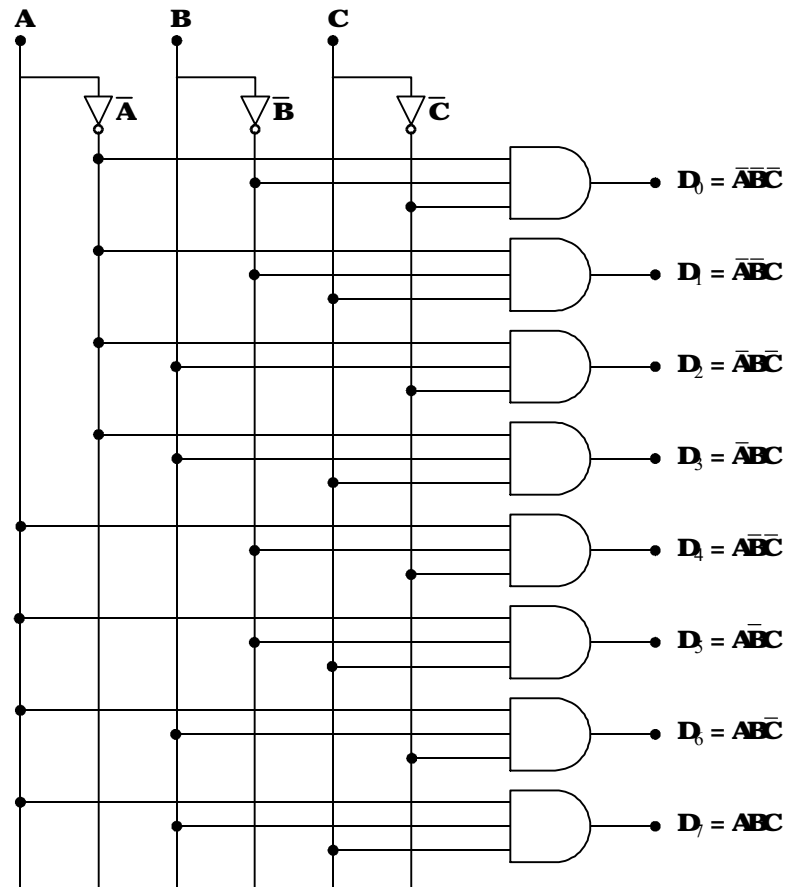
**3-Line-to-8-Line Decoder**

Figure shows the reference matrix for decoding a binary word of 3 bits. In this case, 3-inputs are decoded into eight outputs. Each output represents one of the minterms of the 3-input variables. A 3-bit binary decoder whose control equations are implemented in Figure 1.23. The operation of this circuit is listed in Table 1.19.

*Table 1.19 Truth Table for 3-to-8-Line Decoder*

Inputs			Outputs							
A	B	C	D <sub>0</sub>	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	D <sub>4</sub>	D <sub>5</sub>	D <sub>6</sub>	D <sub>7</sub>
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Figure 1.23 shows the diagram of 3-line-to-8-line decoder.



*Fig. 1.23 A 3-Line-to-8-Line Decoder*



## 1.8.6 Multiplexer

Multiplexer means ‘Many Into One’. Multiplexing is the process of transmitting a large number of information units over a small number of channels or lines.

A digital multiplexer or a data selector (MUX) is a combinational circuit that accepts several digital data inputs and selects one of them and transmits information on a single output line.

Control lines are used to make the selection. The basic multiplexer has several data input lines and a single output line. The selection of a particular line is controlled by a set of selection lines. The block diagram of a multiplexer with  $n$  input lines,  $m$  control signals and one output line is shown in Figure 1.24. A multiplexer is also called a data selector since it selects one of many inputs and steers the data to the output line.

The multiplexer acts like a digitally controlled multiplexer switch where the digital code applied to the SELECT input controls which data inputs will be switched to the output. A digital multiplexer has  $N$  inputs and only one output.

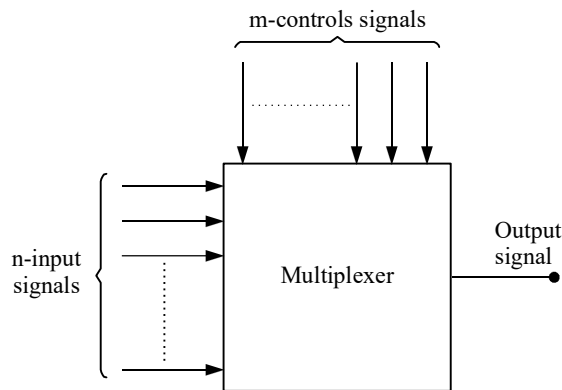


Fig. 1.24 Block Diagram of Multiplexer

### Basic Two-Input Multiplexer

Figure 1.25 shows the basic  $2 \times 1$  MUX. This MUX has two input lines **A** and **B** and one output line **Y**. There is one select input line **S**. When the select input **S** = 0, data from **A** is selected to the output line **Y**. If **S** = 1, data from **B** will be selected to the output **Y**. The logic circuitry for a two-input MUX with data inputs **A** and **B** and select input **S** is shown in Figure 1.25(b). It consists of two AND gates **G**<sub>1</sub> and **G**<sub>2</sub>, a NOT gate **G**<sub>3</sub> and an OR gate **G**<sub>4</sub>. The Boolean expression for the output is given by:

$$Y = A\bar{S} + BS$$

When the select line input **S** = 0, the expression becomes

$$Y = A.1 + B.0 \quad (\text{Gate } G_1 \text{ is enabled})$$

which indicates that output **Y** will be identical to input signal **A**.

Similarly, when **S** = 1, the expression becomes

$$Y = A.0 + B.1 = B \quad (\text{Gate } G_2 \text{ is enabled})$$

showing that output **Y** will be identical to input signal **B**.

## NOTES

NOTES

In many situations a strobe or enable input **E** is added to the select line **S** as shown in Figure 1.25. The multiplexer becomes operative only when the strobe line **E**= 0.

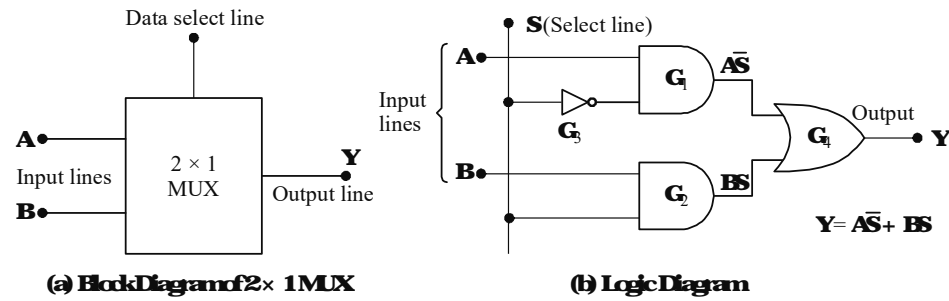


Fig. 1.25 Basic 2 Input Multiplexer

Figure 1.26 shows the logic diagram of 2-input multiplexer with strobe input.

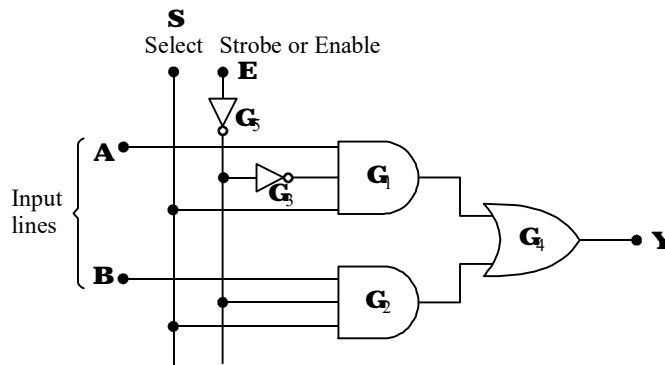


Fig. 1.26 Logic Diagram of 2 Input Multiplexer with Strobe Input

When the strobe input **E** is at logic 0, the NOT gate **G<sub>5</sub>** is 1 and all AND gates **G<sub>1</sub>** and **G<sub>2</sub>** are enabled. Accordingly, when **S**= 0 and 1, inputs **A** and **B** are selected as before. When the strobe input **E**= 1, all lines are disabled and the circuit will not function.

Four-Input Multiplexer

A logic symbol and diagram of a 4-input multiplexer are shown in Figure 1.27. It has two data select lines **S<sub>0</sub>** and **S<sub>1</sub>** and four data input lines. Each of the four data input lines is applied to one input of an AND gate.

Depending on **S<sub>0</sub>** and **S<sub>1</sub>** being 00, 01, 10 or 11, data from input lines **A** to **D** are selected in that order. The Boolean expression for the output is given by the Table 1.20.

Table 1.20 Truth Table for Function Table

Select	Lines	Output
<b>S<sub>1</sub></b>	<b>S<sub>0</sub></b>	<b>Y</b>
0	0	A
0	1	B
1	0	C
1	1	D

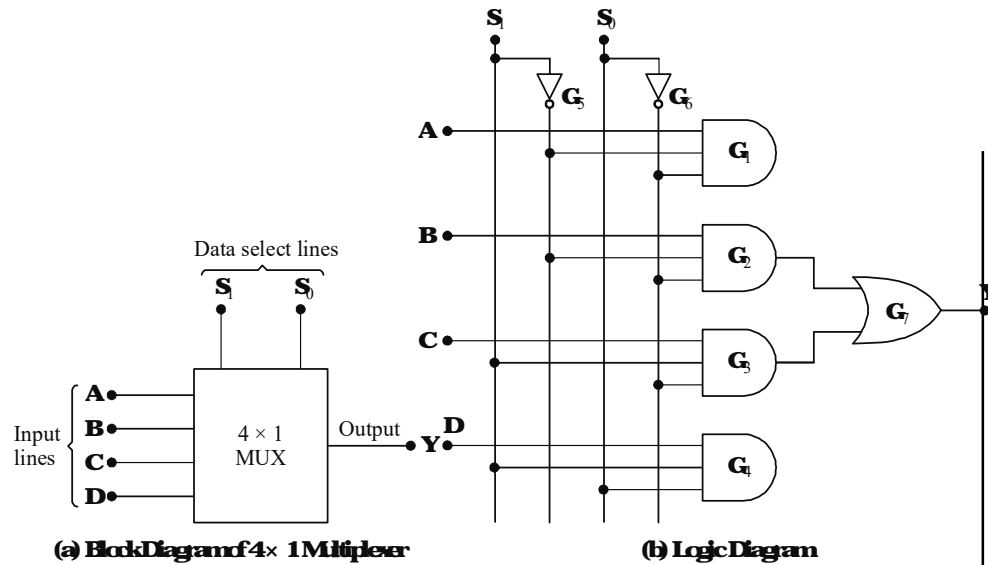


Fig. 1.27 Four-Input Multiplexer

$$Y = A\bar{S}_0\bar{S}_1 + BS_0\bar{S}_1 + C\bar{S}_0S_1 + DS_0S_1$$

If  $S_0S_1 = 00$  (binary 0) is applied to data select lines, the data on input A appears on the data output line.

$$\begin{aligned} Y &= A \cdot 1 \cdot 1 + B \cdot 0 \cdot 1 + C \cdot 1 \cdot 0 + D \cdot 0 \cdot 0 \\ &= A \text{ (Gate } G_1 \text{ is enabled)} \end{aligned}$$

Similarly,  $Y = BS_0\bar{S}_1 = B \cdot 1 \cdot 1 = B$  when  $S_1S_0 = 01$  (Gate  $G_2$  is enabled)

$$Y = C\bar{S}_0S_1 = C \cdot 1 \cdot 1 = C \text{ when } S_1S_0 = 10 \text{ (Gate } G_3 \text{ is enabled)}$$

$$Y = DS_0S_1 = D \cdot 1 \cdot 1 = D \text{ when } S_1S_0 = 11 \text{ (Gate } G_4 \text{ is enabled)}$$

In a similar style, we can construct  $8 \times 1$  MUXes,  $16 \times 1$  MUXes, etc. Nowadays two-, four-, eight- and 16-input multiplexes are readily available in the TTL and CMOS logic families. These basic ICs can be combined for multiplexing a larger number of inputs.

**Multiplexer Applications:** Multiplexer circuits find numerous applications in digital systems. These applications include data selection, data routing, operation sequencing, parallel to serial conversion, waveform generation and logic function generation.

### 1.8.7 Code Converters

A code converter is a logic circuit that changes data presented in one type of binary code to another type of binary code.

The BCD to 7-segment decode/driver is a code converter. The following are the some of the most commonly used code converters:

- (i) BCD to 7-segment
- (ii) BCD to binary
- (iii) Binary to BCD

NOTES

- (iv) Binary to Gray code
- (v) Gray code to binary
- (vi) ASCII to EBCDIC
- (vii) EBCDIC to ASCII

**BCD-to-Binary Converters**

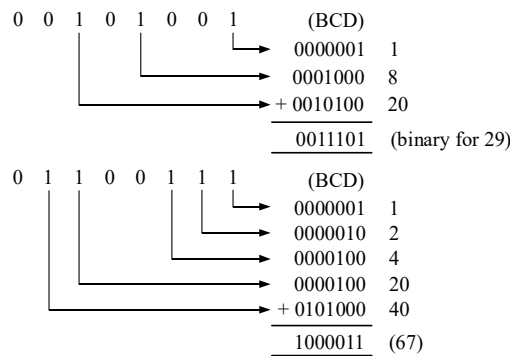
The basic conversion process is as follows:

1. The value of each bit in the BCD number is represented by a binary number.
2. All the binary representation of the bits those are 1's in the BCD are added.
3. The result of this addition is the binary equivalent of the BCD number.

Two-digit decimal values ranging from 00 to 99 can be represented in BCD by two 4-bit code groups. For example,  $19_{10}$  is represented as:

$$\begin{array}{cc} 1 & 9 \\ \hline 0001 & 1001 \end{array}$$

The left most four-bit groups represents 10 and right most four-bit groups represents 9. That is, the left most group has a weight of 10 and the right most group has a weight of  $10^0 = 1$ . The straight binary representation for decimal 19 is  $19_{10} = 10011_2$ .



**Circuit Implementation**

Two 74LS83 four-bit parallel binary adders are wired to perform the conversion process is shown in Figure 1.27. From Figure 1.28, it is seen that  $A_0$  is the only BCD bit that contributes to the LSB,  $b_0$ , of the binary equivalent. Since there is no carry into this bit position,  $A_0$  is connected directly as output  $b_0$ . BCD bits  $B_0$  and  $A_1$  contribute to bit  $b_1$  of the binary output. These two bits are combined in the upper adder to produce output  $b_1$ . Similarly, BCD bits  $D_0$ ,  $A_1$  and  $C_1$  contribute to bit  $b_2$ . The upper adder combines  $D_0$  and  $A_1$  to generate  $\Sigma_2$ , which is connected to the lower adder, where  $C_1$  is added to it to produce  $b_2$ .

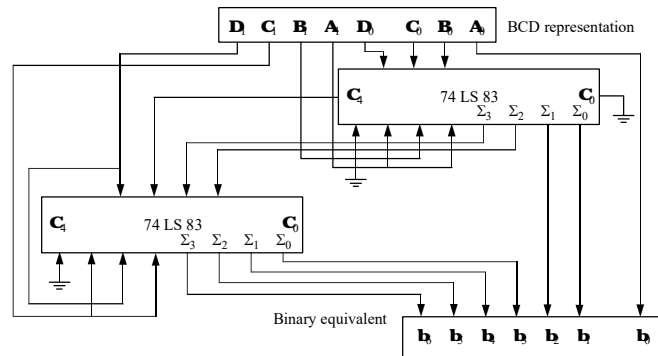


Fig. 1.28 BCD to Binary Converter Implemented with 74LS83 Four Bit Parallel Adders

**Example 1.35:** The BCD representation for decimal 56 is applied to the converter of Figure 1.29. Determine the  $\Sigma$  outputs from each adder and the final binary output.

**Solution:** Let 01010110 be the BCD representation on the circuit diagram. Since  $A_0 = 0$ , the  $b_0$  bit of the output is 0. Let 0011 and 0101 be the inputs of the upper and lower adders. Adding these two we have,

$$\begin{array}{r} 0011 \\ + 0101 \\ \hline 1000 = \Sigma_3 \Sigma_2 \Sigma_1 \Sigma_0 \text{ outputs of the upper adder.} \end{array}$$

The top inputs of the lower adder are 0010 because  $\Sigma_1$  and  $\Sigma_0$  bits become binary outputs  $b_2$  and  $b_1$ . The bottom inputs are 0101. The adder adds these two, we get

$$\begin{array}{r} 0010 \\ + 0101 \\ \hline \therefore b_3 b_2 b_1 b_0 = 0111 = \Sigma_3 \Sigma_2 \Sigma_1 \Sigma_0 \text{ outputs of the lower adder.} \end{array}$$

Hence,  $b_3 b_2 b_1 b_0 = 0111000$  as the correct binary equivalent for decimal 56.

### Binary-to-Gray Code Converters

The Gray code is often used in digital systems because it has the advantage that only one bit in the numerical representation changes between successive numbers. The block diagram of a 4-bit binary-to-gray code converter is shown in Figure 1.28. It has four inputs ( $B_3 B_2 B_1 B_0$ ) representing 4-bit binary numbers and four outputs ( $G_3 G_2 G_1 G_0$ ) representing 4-bit gray code. Truth Table 1.21 shows decimal to binary codes and corresponding gray code.

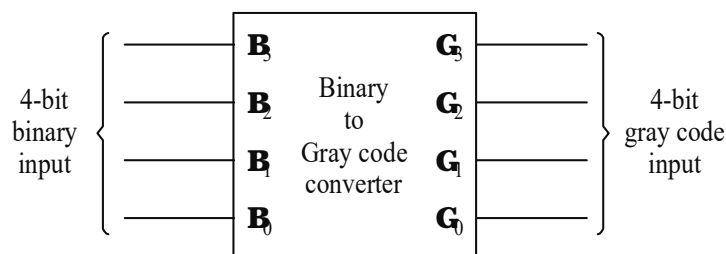


Fig. 1.29 Logic Symbol of 4 Bit Binary to Gray Code

NOTES

Truth Table 1.21 Binary to Gray Code Converter

Decimal	Binary Code Inputs				Gray Code Outputs			
	B <sub>3</sub>	B <sub>2</sub>	B <sub>1</sub>	B <sub>0</sub>	G <sub>3</sub>	G <sub>2</sub>	G <sub>1</sub>	G <sub>0</sub>
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	0	0	0	1	1
3	0	0	1	1	0	0	1	0
4	0	1	0	0	0	1	1	0
5	0	1	0	1	0	1	1	1
6	0	1	1	0	0	1	0	1
7	0	1	1	1	0	1	0	0
8	1	0	0	0	1	1	0	0
9	1	0	0	1	1	1	0	1
10	1	0	1	0	1	1	1	1
11	1	0	1	1	1	1	1	0
12	1	1	0	0	1	0	1	0
13	1	1	0	1	1	0	1	1
14	1	1	1	0	1	0	0	1
15	1	1	1	1	1	0	0	0

From the Truth Table 1.21, the logic expressions for the gray code outputs can be written as:

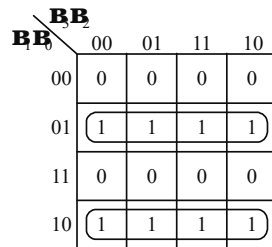
$$G_3 = \sum m(8, 9, 10, 11, 12, 13, 14, 15)$$

$$G_2 = \sum m(4, 5, 6, 7, 8, 9, 10, 11)$$

$$G_1 = \sum m(2, 3, 4, 5, 10, 11, 12, 13)$$

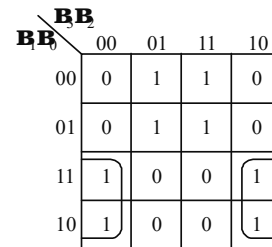
$$G_0 = \sum m(1, 2, 5, 6, 9, 10, 13, 14)$$

The above expression can be simplified using K-map method as shown in Figure 1.29.



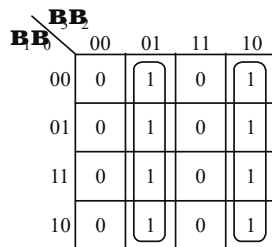
(a) K-map for G<sub>0</sub>

From the K-map,  $G_0 = \overline{B_3}B_2 + B_3B_2 = B_2 \oplus B_3$



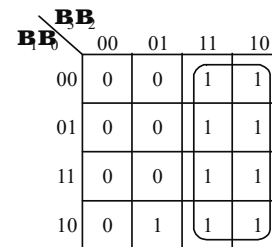
(b) K-map for G<sub>1</sub>

From the K-map,  $G_1 = \overline{B_3}B_1 + B_3\overline{B_1} + G_1 = B_2 \oplus B_1$



(c) K-map for G<sub>2</sub>

From the K-map,  $G_2 = \overline{B_3}B_1 + B_3B_1 = B_1 \oplus B_3$

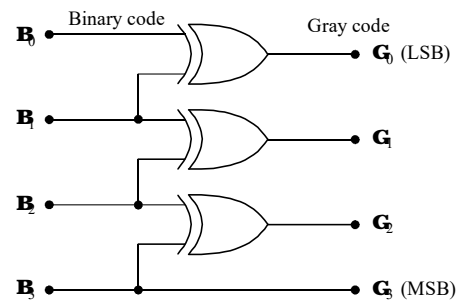


(d) K-map for G<sub>3</sub>

From the K-map,  $G_3 = B_3$

Fig. 1.30 KMap Simplification for Binary to Gray Code Converter

The logic diagram of 4-bit binary-to-gray code converter using XOR gates is shown in Figure 1.30.



**Fig. 1.31 Logic Diagram of 4 Bit Binary to Gray Code Converter**

**NOTES**

**Gray Code-to-Binary Converter**

The block diagram of a 4-bit gray-code-to-binary converter is shown in Figure 1.32. It has four inputs  $G_3, G_2, G_1, G_0$  representing 4-bit gray code and four outputs  $B_3, B_2, B_1, B_0$  representing 4-bit binary numbers. Table 1.22 shows the Truth Table for gray code-to-binary code converter.

**Table 1.22 Truth Table of Gray Code to Binary Code Converter**

Gray Code Inputs				Binary Code Outputs			
$G_3$	$G_2$	$G_1$	$G_0$	$B_3$	$B_2$	$B_1$	$B_0$
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	1
1	0	0	0	1	1	1	1
1	0	0	1	1	1	1	0
1	0	1	0	1	1	0	0
1	0	1	1	1	1	0	1
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	1
1	1	1	0	1	0	1	1
1	1	1	1	1	0	1	0

From the Truth Table 1.22, the logic expressions for the binary output can be written as:

$$B_3 = \Sigma_m(8, 9, 10, 11, 12, 13, 14, 15)$$

$$B_2 = \Sigma_m(4, 5, 6, 7, 8, 9, 10, 11)$$

$$B_1 = \Sigma_m(2, 3, 4, 5, 8, 9, 14, 15)$$

$$B_0 = \Sigma_m(1, 2, 4, 7, 8, 11, 13, 14)$$

These expressions can be simplified using Karnaugh map as shown in Figure 1.32.

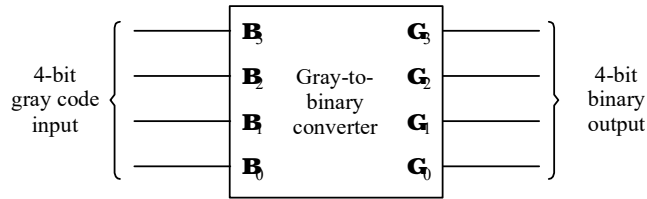


Fig. 1.32 Block Diagram of 4 Bit Gray Code to Binary Code Converter

From the K-map shown in Figure 1.33(a), we have

$$\begin{aligned}
 B_0 &= \bar{G}_3 \bar{G}_2 \bar{G}_1 G_0 + \bar{G}_3 \bar{G}_2 G_1 \bar{G}_0 + G_2 G_2 \bar{G}_1 G_0 + G_3 G_2 G_1 \bar{G}_0 + \bar{G}_3 G_2 \bar{G}_1 \bar{G}_0 \\
 &\quad + G_3 \bar{G}_2 \bar{G}_1 \bar{G}_0 + \bar{G}_3 G_2 G_1 G_0 + G_3 \bar{G}_2 G_1 G_0 \\
 &= \bar{G}_3 \bar{G}_2 (\bar{G}_1 G_0 + G_1 \bar{G}_0) + G_3 G_2 (\bar{G}_1 G_0 + G_1 \bar{G}_0) + \bar{G}_1 \bar{G}_0 (\bar{G}_3 G_2 + G_3 \bar{G}_2) \\
 &\quad + G_1 G_0 (\bar{G}_3 G_2 + G_3 \bar{G}_2) \\
 &= \bar{G}_3 \bar{G}_2 (G_0 \oplus G_1) + G_3 G_2 (G_0 \oplus G_1) + \bar{G}_1 \bar{G}_0 (G_2 \oplus G_3) + G_1 G_0 (G_2 \oplus G_3) \\
 &= G_0 \oplus G_1 (\bar{G}_3 \bar{G}_2 + G_3 G_2) + (G_2 \oplus G_3) (\bar{G}_1 \bar{G}_0 + G_1 G_0) \\
 B_1 &= G_0 \oplus G_1 \oplus G_2 \oplus G_3 = G_0 \oplus B_1
 \end{aligned}$$

		$G_3 G_2$			
$G_1 G_0$		00	01	11	10
00		0	1	0	1
01		1	0	1	0
11		0	1	0	1
10		1	0	1	0

(a) K-map for  $B_0$

		$G_3 G_2$			
$G_1 G_0$		00	01	11	10
00		0	1	0	1
01		0	1	0	1
11		1	0	1	0
10		1	0	1	0

(b) K-map for  $B_1$

		$G_3 G_2$			
$G_1 G_0$		00	01	11	10
00		0	1	0	1
01		0	1	0	1
11		0	1	0	1
10		0	1	0	1

(c) K-map for  $B_2$

		$G_3 G_2$			
$G_1 G_0$		00	01	11	10
00		0	0	1	1
01		0	0	1	1
11		0	0	1	1
10		0	1	1	1

(d) K-map for  $B_3$

Fig. 1.33 Kmap Simplification for Gray Code to Binary Code Converter

From the K-map shown in Figure 1.33(b), we have

$$\begin{aligned}
 B_1 &= \bar{G}_3 \bar{G}_2 G_1 + \bar{G}_3 G_2 \bar{G}_1 + G_3 G_2 G_1 + G_3 \bar{G}_2 \bar{G}_1 \\
 &= \bar{G}_3 (\bar{G}_2 G_1 + G_2 \bar{G}_1) + G_3 (G_2 G_1 + \bar{G}_2 \bar{G}_1) \\
 &= \bar{G}_3 (G_2 \oplus G_1) + G_3 (G_2 \oplus G_1) \\
 B_1 &= G_3 \oplus G_2 \oplus G_1 = B_2 \oplus G_1
 \end{aligned}$$

From the K-map shown in Figure 1.33(c), we have

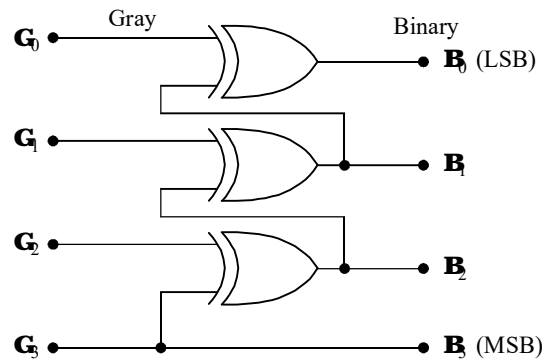
$$B_2 = \bar{G}_3 G_2 + G_3 \bar{G}_2 = G_3 \oplus G_2 = B_3 \oplus G_2$$



From the K-map shown in Figure 1.33(d), we have

$$B_3 = G_3$$

The simplified expressions can be implemented using XOR gates as shown in Figure 1.34.



**Fig. 1.34 Logic Diagram of 4 Bit Gray Code to Binary Code Converter**

**NOTES**

---

## 1.9 ARITHMETIC CIRCUITS

---

Arithmetic operations are performed in computers not by using decimal numbers, as we do normally, but by using binary numbers. Arithmetic circuits in computers and calculators perform arithmetic and logic operations. All arithmetic operations take place in the arithmetic unit of a computer. The electronic circuit is capable of doing addition of two or three binary digits at a time and the binary addition alone is sufficient to do subtraction. Thus, a single circuit of a binary adder with suitable shift register can perform all the arithmetic operations.

Arithmetic operations, such as addition, subtraction, multiplication and division can be performed on binary numbers.

### 1.91 Binary Addition

Binary addition is performed in the same manner as decimal addition. Binary addition is the key to binary subtraction, multiplication and division. There are only four cases that occur in adding the two binary digits in any position. This is shown in Table 1.23.

- (i)  $1 + 1 + 1 = 11$  (i.e., 1 carry of 1 into next position)
- (ii)  $1 + 1 + 1 + 1 = 100$
- (iii)  $10 + 1 = 11$

The rules of (1), (2) and (3) in Table 1.23 are just decimal addition. The rule (4) states that adding 1 and 1 gives one 0 (meaning decimal 2 and not decimal 10).

There is a carry from the previous position. ‘Carry overs’ are performed in the same manner as in decimal arithmetic. Since, 1 is the larger digit in the binary system, any sum greater than 1 requires that a digit be carried out.

NOTES

Table 1.23 Binary Addition

Sl. No	Addend (A)	+	Addend (B)	Carry (C)	Sum (S)	Result
1	0	+	0	0	0	0
2	0	+	1	0	1	1
3	1	+	0	0	1	1
4	1	+	1	1	0	10

**Example 1.36:** Add the binary numbers (i) 011 and 101, (ii) 1011 and 1110, (iii) 10.001 and 11.110, (iv) 1111 and 10010, and (v) 11.01 and 101.0111.

**Solution:**

(i) Binary number		Equivalent Decimal Number																					
	<pre> 11 ← Carry  011 + 101 ----- Sum = 1000           </pre>		<pre> 3 5 8           </pre>																				
(ii)	<table border="0"> <tr> <td>Binary</td> <td>Decimal</td> </tr> <tr> <td>11 ← Carry</td> <td></td> </tr> <tr> <td>1011</td> <td>11</td> </tr> <tr> <td>+ 1110</td> <td>+ 14</td> </tr> <tr> <td>Sum = 11001</td> <td>25</td> </tr> </table>	Binary	Decimal	11 ← Carry		1011	11	+ 1110	+ 14	Sum = 11001	25	(iii)	<table border="0"> <tr> <td>Binary</td> <td>Decimal</td> </tr> <tr> <td>1 ← Carry</td> <td></td> </tr> <tr> <td>10.001</td> <td>2.125</td> </tr> <tr> <td>+ 11.110</td> <td>+ 3.750</td> </tr> <tr> <td>Sum = 101.111</td> <td>5.875</td> </tr> </table>	Binary	Decimal	1 ← Carry		10.001	2.125	+ 11.110	+ 3.750	Sum = 101.111	5.875
Binary	Decimal																						
11 ← Carry																							
1011	11																						
+ 1110	+ 14																						
Sum = 11001	25																						
Binary	Decimal																						
1 ← Carry																							
10.001	2.125																						
+ 11.110	+ 3.750																						
Sum = 101.111	5.875																						
(iv)	<table border="0"> <tr> <td>Binary</td> <td>Decimal</td> </tr> <tr> <td>11 ← Carry</td> <td></td> </tr> <tr> <td>1111</td> <td>15</td> </tr> <tr> <td>+ 10010</td> <td>+ 18</td> </tr> <tr> <td>Sum = 100001</td> <td>33</td> </tr> </table>	Binary	Decimal	11 ← Carry		1111	15	+ 10010	+ 18	Sum = 100001	33	(v)	<table border="0"> <tr> <td>Binary</td> <td>Decimal</td> </tr> <tr> <td>11 1 ← Carry</td> <td></td> </tr> <tr> <td>11.01</td> <td>3.25</td> </tr> <tr> <td>101.0111</td> <td>+ 5.4375</td> </tr> <tr> <td>Sum = 1000.1011</td> <td>8.6875</td> </tr> </table>	Binary	Decimal	11 1 ← Carry		11.01	3.25	101.0111	+ 5.4375	Sum = 1000.1011	8.6875
Binary	Decimal																						
11 ← Carry																							
1111	15																						
+ 10010	+ 18																						
Sum = 100001	33																						
Binary	Decimal																						
11 1 ← Carry																							
11.01	3.25																						
101.0111	+ 5.4375																						
Sum = 1000.1011	8.6875																						

Since, the circuit in all digital systems actually performs addition that can handle only two numbers at a time, it is not necessary to consider the addition of more than two binary numbers. When more than two numbers are to be added, the first two are added together and then their sum is added to the third number, and so on. Almost all modern digital machines can perform addition operation in less than 1 μs.

Logic equations representing the sum is also known as the exclusive OR function and can be represented also in Boolean ring algebra as  $S = \bar{A}B + \bar{B}A = A \oplus B$ .

### 1.9.2 Binary Subtraction

Subtraction is the inverse operation of addition. To subtract, it is necessary to establish a procedure for subtracting a large digit from a small digit. The only case in which this occurs with binary numbers is when 1 is subtracted from 0. The remainder is 1, but it is necessary to borrow 1 from the next column to the left. The rules of binary subtraction are shown in Table 1.24.

1. 0 - 0 = 0
2. 1 - 0 = 1

3.  $1 - 1 = 0$
4.  $0 - 1 = 0$  with a borrow of 1
5.  $10 - 1 = 01$

**Table 1.24 Binary Subtraction**

Sl. No	Minuend A	–	Subtrahend B	Result
1	0	–	0	0
2	0	–	1	0
3	1	–	0	1
4	1	–	1	0

with a borrow of 1

<b>Example 1.37:</b>	(i) Binary	Decimal	(ii) Binary	Decimal
<b>Solution:</b>	1001	9	10000	16
	– 101	– 5	– 011	– 3
	Difference = 100	4	1101	13
	(iii) Binary	Decimal	(iv) Binary	Decimal
	110.01	6.25	1101	13
	– 100.1φ	– 4.5φ	– 1010	– 10
	1.11	1.75	0011	3

**Example 1.38:** Show the binary subtraction of  $128_{10}$  from  $210_{10}$ .

**Solution:** Converting the given decimal numbers into the corresponding hexadecimal number we have,

$$\begin{array}{r}
 210 \rightarrow D2H \rightarrow 1101\ 0010 \\
 128 \rightarrow 80H \rightarrow 1000\ 0000 \\
 \begin{array}{r}
 1101\ 0010 \\
 - 1000\ 0000 \\
 \hline
 0101\ 0010
 \end{array}
 \end{array}$$

$$\begin{array}{r}
 D2H \\
 - 80H \\
 \hline
 52H
 \end{array}$$

### 1.9.3 Binary Multiplication

Multiplication of binary numbers is performed in the same manner as the multiplication of decimal numbers. The following are the four basic rules for multiplying binary digits:

1.  $0 \times 0 = 0$
2.  $0 \times 1 = 0$
3.  $1 \times 0 = 0$
4.  $1 \times 1 = 1$

In a computer, the multiplication operation is performed by repeated additions, in much the same manner as the addition of all partial products to obtain the full product. Since the multiplier digits are either 0 or 1, we always multiply by 0 or 1 and no other digit.

### NOTES

**Example 1.39:** Multiply the binary numbers 1011 and 1101.

$$\begin{array}{r} \text{Solution: } 1011 \leftarrow \text{Multiplicand} = 11_{10} \\ \times 1101 \leftarrow \text{Multiplier} = \times 13_{10} \\ \hline \end{array}$$

$$\begin{array}{r} 1011 \\ 0000 \quad \text{Partial product} = 143_{10} \\ 1011 \\ 1011 \\ \hline 10001111 \quad \leftarrow \text{Final product} = 143_{10} \end{array}$$

### 1.9.4 Binary Division

The processes of dividing one binary number (the dividend) by another (the divisor) is the same as that which is followed for decimal numbers which we usually refer to as the method of 'Long Division'. The rules for binary division are as follows:

1.  $0 \div 1 = 0$
2.  $1 \div 1 = 1$
3.  $0 \div 0 = \text{No meaning as in decimal system}$
4.  $1 \div 0 = \text{No meaning}$

In considering division, we will assume that the dividend is larger than the divisor. The following are the steps for binary division:

1. Start from the left or the dividend.
2. Perform a series of subtractions in which the divisor is subtracted from the dividend.
3. If subtraction is possible, put a 1 in the quotient and subtract the divisor from the corresponding digits of the dividend.
4. If subtraction is not possible (divisor greater than remainder), record a zero in the quotient. Bring down the next digit to add to the remainder digits. Proceed as before in a manner similar to long division.

---

## 1.10 COMBINATIONAL CIRCUITS AND SEQUENTIAL CIRCUITS

---

In digital electronics, we have two broad categories of logic circuits. They are as follows:

- Combinational Circuit
- Sequential Circuit

**Combinational Circuit:** In a combinational circuit, each output depends entirely on the immediate (present) inputs to the circuit. The block diagram of a combinational circuit is shown in Figure 1.35.

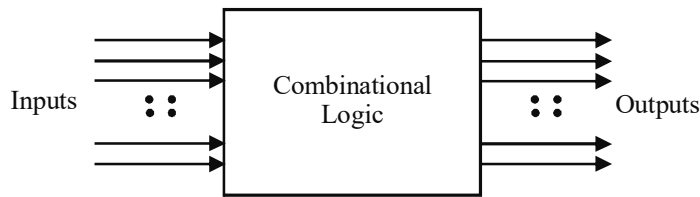


Fig. 1.35 Block Diagram of a Combinational Circuit

**Sequential Circuit:** In this type of circuit, the output depends on both the **present** and the **past** inputs. It means that this type of circuit involves the memory elements for storing **past** input conditions. The block diagram of a sequential circuit is shown in Figure 1.36.

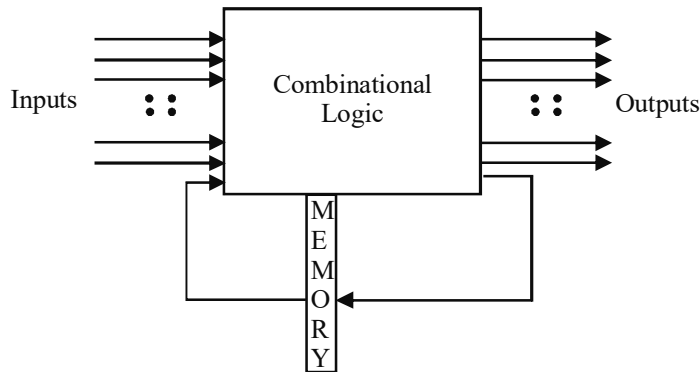


Fig. 1.36 Block Diagram of a Sequential Circuit

### 1.10.1 Analysis of a Combinational Circuit

An analysis of the function of a combinational circuit is shown in Figure 1.37.

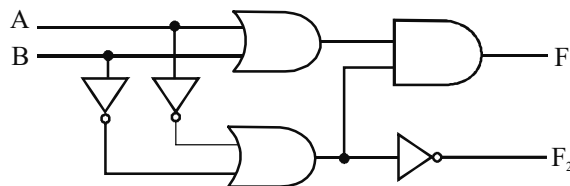


Fig. 1.36 A Simple Combinational Circuit

The steps for analysis are as follows:

- Label the inputs and outputs
- Obtain the functions of intermediate points and the outputs
- Draw the truth table
- Deduce the functionality of the circuit

In the circuit shown in Figure 1.38, we have the following:

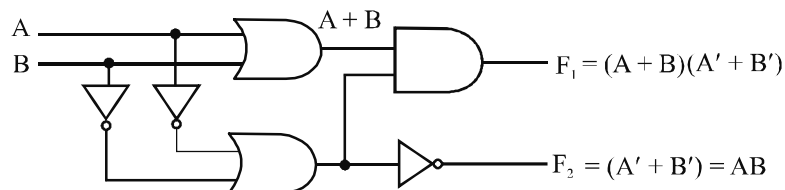


Fig. 1.38 A Combinational Circuit with Label

Then we form the truth table for the system.

Table 1.25 Truth Table for the Circuit shown in Figure 1.38

A	B	(A + B)	(A' + B')	F <sub>1</sub>	F <sub>2</sub>
0	0	0	1	0	0
0	1	1	1	1	0
1	0	1	1	1	0
1	1	1	0	0	1

## 1.11 REGISTERS AND COUNTERS

A register is a group of flip-flops used to store or manipulate data or both. Each flip-flop is capable of storing one bit of information. An **n** bit register has **n** flip-flop and is capable of storing any binary information containing **n** bits.

The register is a type of sequential circuit and an important building block that is used in digital system like multipliers, dividers, memories, microprocessors, etc.

A register stores a sequence of 0's and 1's. The registers that are used to store information are known as **memory registers**. When used to process information, they are called **shift registers**.

A shift register is a group of FFs arranged so that the binary numbers stored in the FFs are shifted from one FF to the next for every clock pulse.

Shift registers often are used to store data momentarily. Figure 1.39 shows a typical example of where shift registers might be used in a digital system (calculator). Shift registers are used to hold information from the encoder for the processing unit. A shift register is also being used for temporary storage between the processing unit and the decoder. Shift registers are also used at other locations within a digital system.

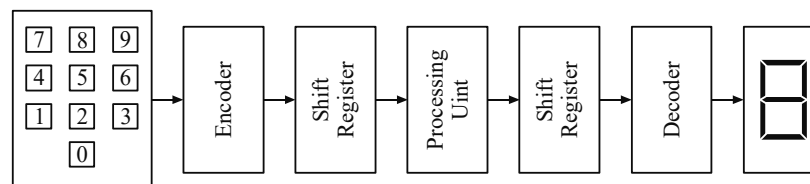


Fig. 1.39 Block Diagram of a Digital System using Shift Registers

There are two modes of operation for registers. The first operation is **series or serial operation**. The second type of operation is **parallel shifting**. Input and output functions associated with registers include (1) Serial input/serial output (2) Serial input/parallel output (3) Parallel input/parallel output (4) Parallel input/serial output.

Hence, input data is presented to registers in either a parallel or a serial format.

All the flip-flops to be affected (set or reset) to input parallel data to a register, requires that at the same time. To output parallel data, it requires that the flip-flop Q outputs be accessible. Serial input data loading requires that one data bit at a time is presented to either the most or least significant flip-flop. Data are shifted from the flip-flop initially are loaded to the next one in series. Serial output data are taken from a single flip-flop, one bit at a time.

Serial data input or output operations require multiple clock pulses. Parallel data operations only take one clock pulse. Data can be loaded in one format and removed in another. Two functional parts are required by all shift registers: (1) Data storage flip-flops and (2) Logic to load, unload and shift the stored information.

The block diagrams of four basic register types is shown in Figure 1.40. Registers can be designed using discrete flip-flops (S-R J-K and D-type). Registers are also available as MSI (Medium-Scale Integration).

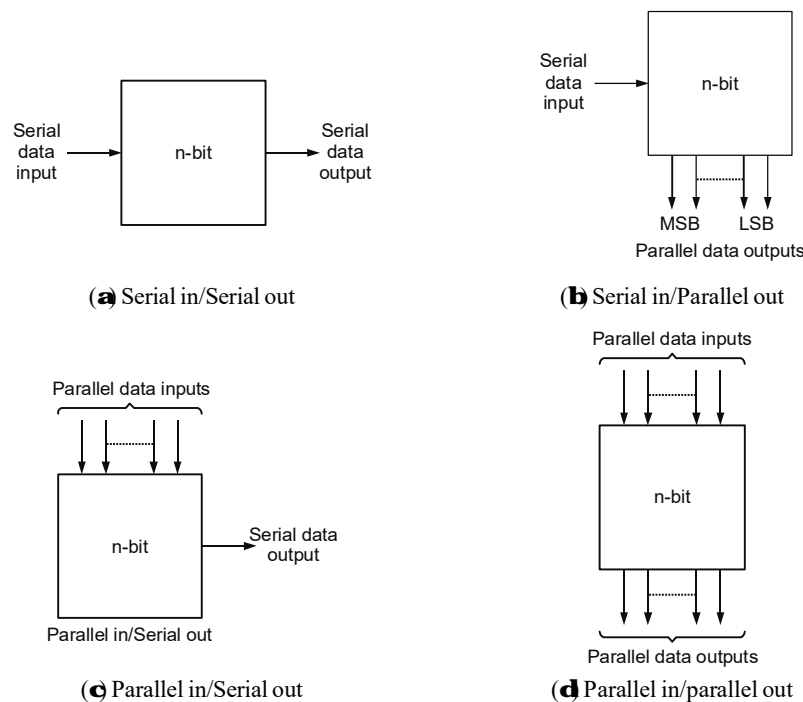


Fig. 1.40 Register Types

### 1.11.1 Serial-in-Serial-out Shift Registers

This type of shift register accepts data serially in a numerical order bit at a time on a single line. It produces the stored information on its output also in serial form. Data may be shifted left (from low-to high order bits) using **shift left register** or shifted right (from high to low order bits) using a **shift right register**.

#### Shift-Left Register

A shift left register can be built using **DFFs** or **JKFFs** as shown in Figure 1.41. A J-K FF register requires connection of both **J** and **K** inputs, input data are connected to the right most (lowest order) stage with data being shifted bit-by-bit to the left.

NOTES

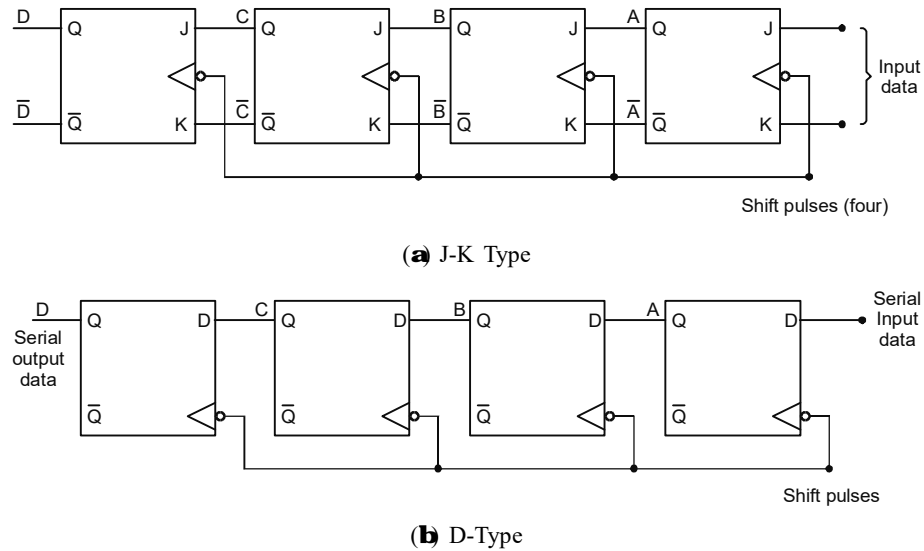


Fig. 1.41 Shift Left Registers (a) JK (b) D-Type

For register of Figure 1.41 (b) using DFFs, a single data line is connected between states, again, 4 shift pulse are required to shift a 4-bit word into the 4-stage register.

The shift pulse is applied to each stage, operating each simultaneously. When the shift pulse occurs, the data input is shifted into that stage. Each stage is set or reset corresponding to the input data at the time of shift pulse occurs. Thus, the input data bit is shifted into stage **A** by the first shift pulse. At the same time the data of stage **A** is shifted into stage **B** and so on for the following stages. For each shift pulse, data stored in the register stages shift left by one stage. New data are shifted into stage **A** whereas the data present in stage **D** are shifted out (to the left) for use by some other shift register or computer unit.

Consider starting with all stages reset and applying a steady logical 1 input a data input to stage **A**. The data in each stage after each of four shift pulses is shown in Truth Table 1.26. The logical 1 input shifts into stage **A** and the shifts left to stage **D** after four shift pulses.

Another example, that could be considered is shifting of alternate 0 and 1 data into stage **A**, starting from all logical 1. Truth Table 1.26 shows the data in each stage after each of four shift pulses.

Truth Table 1.26 Operation of Shift Left Register

Shift Pulse	D	C	B	A
0	0	0	0	0
1	0	0	0	1
2	0	0	1	1
3	0	1	1	1
4	1	1	1	1

As a third example of shift register operation, consider starting with the count in step 4 of Truth Table 1.27 and applying four more shift pulses while placing a steady logical-0 input as data input to stage **A**. This is shown in Truth Table 1.28.



NOTES

Truth Table 1.27

Shift pulse	D	C	B	A
0	1	1	1	1
1	1	1	1	0
2	1	1	0	1
3	1	0	1	0
4	0	1	0	1

Truth Table 1.28

Shift pulse	D	C	B	A
0	0	1	0	1
1	1	0	1	0
2	0	1	0	0
3	1	0	0	0
4	0	0	0	0

**Shift Right Register**

A shift right register can also be built using **D**FFs or **JK**FFs as shown in Figure 1.42. Let us illustrate the entry of the 4-bit binary number 1101 into the register, beginning with the right most bit. The 1 is put into the data input line, making **D**= 1 for stage **D**. When the first clock pulse is applied, FF **A** is SET, thus storing the 1. Next the 0 is applied to the data input, making **D**= 0 for FF **B** because **D** (input) of FF **B** is connected to the **Q<sub>A</sub>** output.

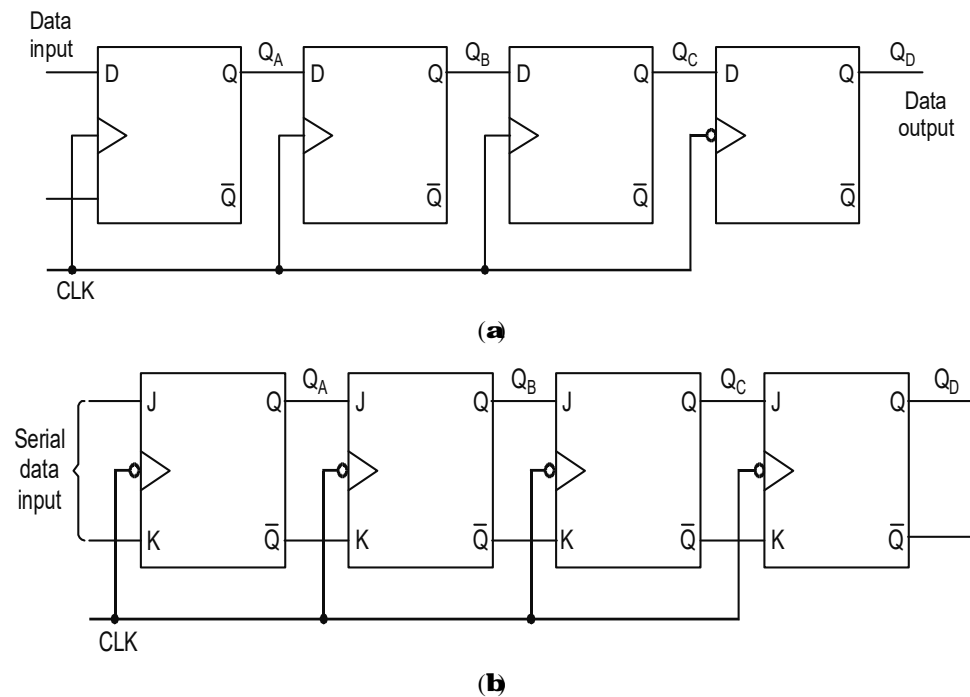


Fig. 1.42 Shift Right Registers (a) JK Type (b) D Type

When the second clock pulse occurs, the 0 on the data input is ‘Shifted’ into the FF **A** because FF **A** RESETs, and the 1 that was in FF **A** is ‘Shifted’ into FF **B**. The next 1 in the binary number is now put onto the data-input line, and a clock pulse is applied. The 1 is entered into FF **A**, the 0 stored in FF **A** is shifted into FF **B**, and the 1 stored in FF **B** is shifted into FF **C**. The last bit in the binary number, a 1, is now applied to the data input, and a clock pulse is applied. This time the 1 is entered into FF **A**, the 1 stored in FF **A** is shifted into FF **B**, the 0 stored in FF **B** is shifted into FF **C**, and the 1 stored in FF **C** is shifted into FF **D**. This completes the serial entry of the 4-bit binary number into the shift register, where it can be stored for any amount of time. Truth Table 1.29 shows the action of shifting all logical-1 inputs into an initially reset shift register. Truth Table 1.30 shows the register operation for the entry of 1101.

NOTES

Truth Table 1.29

Shiftpulse	Q <sub>A</sub>	Q <sub>B</sub>	Q <sub>C</sub>	Q <sub>D</sub>
0	0	0	0	0
1	1	0	0	0
2	0	1	0	0
3	1	0	1	0
4	1	1	0	1

Truth Table 1.30

Shiftpulse	Q <sub>A</sub>	Q <sub>B</sub>	Q <sub>C</sub>	Q <sub>D</sub>
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0
4	1	1	1	1

The waveforms shown in Figure 1.43 illustrate the entry of 4-bit number 0100. For a **JK**FF, the data bit to be shifted into the FF must be present at the **J** and **K** inputs when the clock transitions (low or high). Since, the data bit is either a 1 or a 0, there are two cases:

1. To shift a 0 into the FF, **J**= 0 and **K**= 1,
2. To shift a 1 into the FF, **J**= 1 and **K**= 0,

**At time A :** All the FFs are reset. The FF output just after time **A** are QRST = 0000.

**At time B :** The FFs all contain 0s, the FF outputs are QRST = 0000.

**At time C :** The FFs still all contain 0s. The FF output after time **C** are QRST = 1000.

**At time D :** The FF output are QRST = 0100.

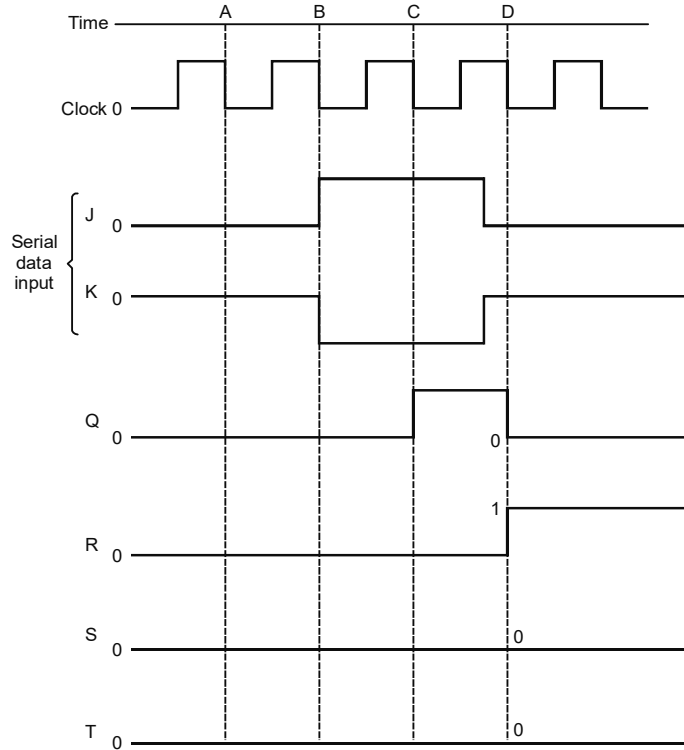
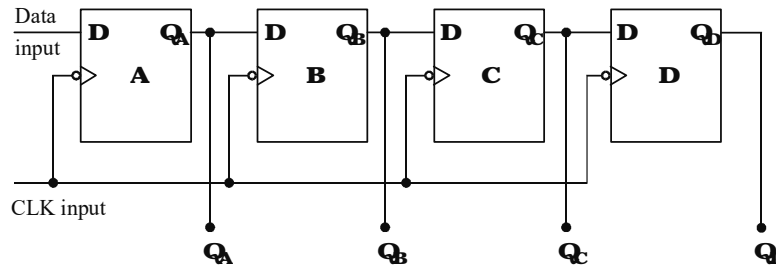


Fig. 1.43 Waveforms of 4 Bit Serial Input Shift Register

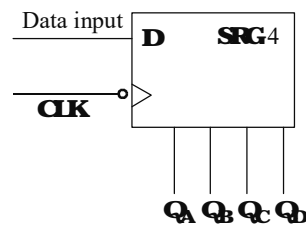
NOTES

### 1.11.2 Serial-in-Parallel-out Shift Registers

The logic diagram of a 4-bit serial-in-parallel-out shift register is shown in Figure 1.44. It has one input and the number of output pins is equivalent to the order of FFs in the register. In this register, data is entered serially but shifted out in parallel. In order to shift the data out in parallel, it is necessary to have all the data available at the outputs at the same time. On storing the data, every bit surfaces on its relevant output and all bits are available simultaneously, rather than on a bit-by-bit basis as with the serial output.



(a) Logic Diagram



(b) Logic Symbol

Fig. 1.44 ~~A Serial In Parallel Out Shift Register~~

### 1.11.3 Parallel-in-Serial-out Shift Registers

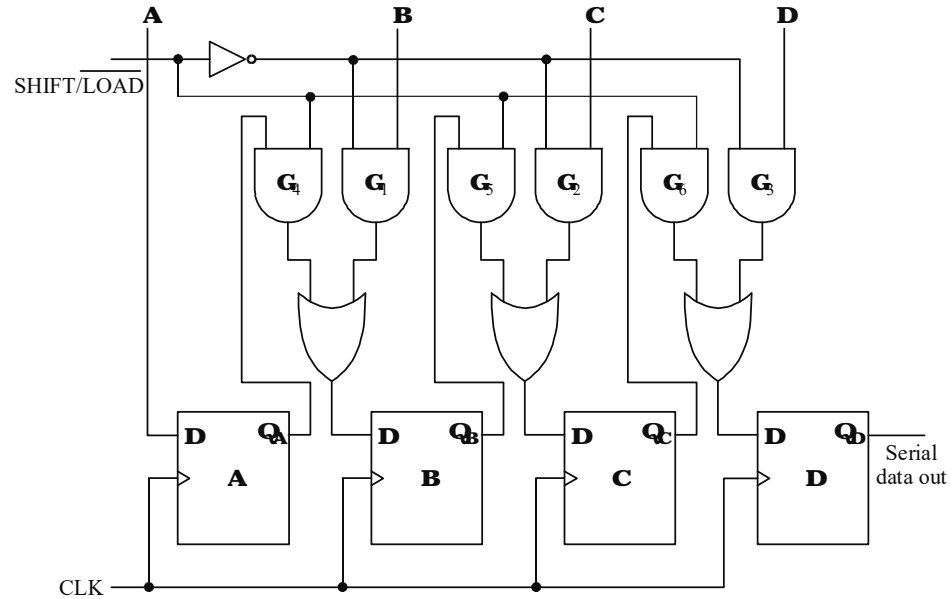
For a register with parallel data inputs, the bits are entered simultaneously into their respective stages on parallel lines rather than on a bit-by-bit basis on one line.

A 4-bit parallel-in-serial-out shift register is illustrated in Figure 1.45. It has four data-input lines **A**, **B**, **C** and **D** and a  $\overline{\text{SHIFT/LOAD}}$  input.  $\overline{\text{SHIFT/LOAD}}$  is a control input that allows four bits of data to enter the register in parallel or shift the data in serial.

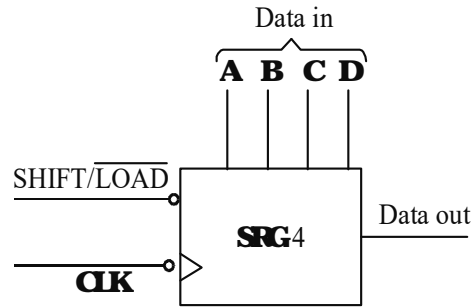
When  $\overline{\text{SHIFT/LOAD}}$  is LOW, AND gates **G**<sub>1</sub> through **G**<sub>3</sub> are enabled, allowing each data bit to be applied to the **D** input of its respective FF. When a clock pulse is applied, the FFs with **D** = 1 will SET and those with **D** = 0 will RESET, thereby storing all four bits simultaneously.

When  $\overline{\text{SHIFT/LOAD}}$  is HIGH, AND gates through **G**<sub>1</sub> through **G**<sub>3</sub> are disabled and AND gates **G**<sub>4</sub> through **G**<sub>7</sub> are enabled, allowing the data bits to shift right from one stage to the next. The OR gates allow either the normal shifting operation or the parallel data entry operation, depending on which AND gates are enabled by the level on the  $\overline{\text{SHIFT/LOAD}}$  input.

NOTES



(a) Logic Diagram



(b) Logic Symbol

Fig. 1.45 A 4 Bit Parallel-in-Serial-Out Shift Register

### 1.11.4 Parallel-in-Parallel-out Registers

In this type of register, data inputs can be shifted either in or out of the register in parallel. It has four inputs and four outputs. In this register, there is no interconnection between successive FFs since no serial shifting is required. Therefore, the moment the parallel entry of the input data is accomplished, the respective bits will appear at the parallel outputs.

The logic diagram of a 4-bit parallel-in-parallel-out shift register is shown in Figure 1.46. Let **A**, **B**, **C** and **D** be the inputs applied directly to delay (**D**) inputs of respective FFs. Now, on applying a clock pulse, these inputs are entered into the register and are immediately available at the outputs **QA**, **QB**, **QC** and **QD**.

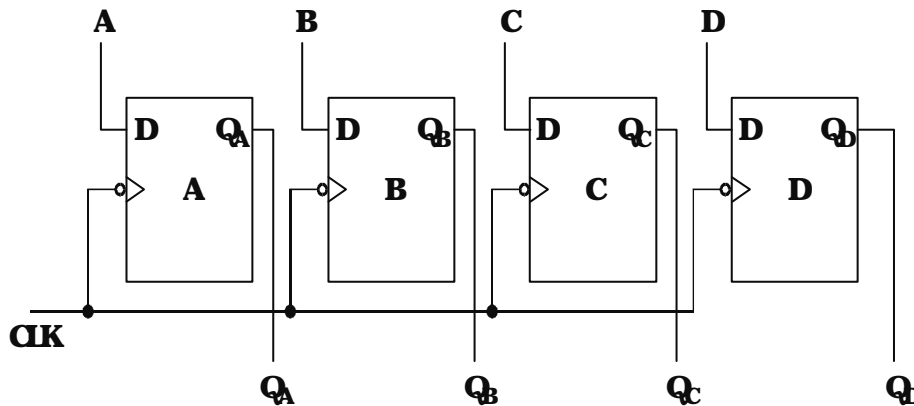


Fig. 1.46 Logic Diagram of a 4-Bit Parallel-In-Parallel-Out Shift Register

### 1.11.5 Bidirectional Shift Registers

A register which is capable of shifting data on either side is known as a bidirectional shift register. A register that can shift in only one direction is called a **unidirectional shift register**. If the register has shift and parallel load capabilities, then it is called a **shift register with parallel load** or **universal shift register**.

Shift registers can be used for transforming serial data to parallel data, and vice versa. If a parallel load capability is added to a shift register, then data entered in parallel can be taken out in a serial manner by shifting the data stored in the register. The most general shift register has the following capabilities:

1. A **clear control** to clear the register to 0.
2. A **clock input** for clock pulses to synchronize all operations.
3. A **shift right** control to enable the shift-right operation and serial input and output lines associated with the shift right.
4. A **shift left** control to enable the shift-left operation and the serial input and output lines associated with the shift left.
5. A **parallel load** control to enable a parallel transfer and the **n** input lines associated with the parallel transfer.
6. **n** parallel output lines.
7. A control line that leaves the information in the register unchanged even though clock pulses are continuously applied.

The logic diagram of a 4-bit bidirectional shift register is shown in Figure 1.47(b). A HIGH on the RIGHT /  $\overline{\text{LEFT}}$  control input allows data to be shifted to the right, and a LOW enables a left-shift of data. When the RIGHT /  $\overline{\text{LEFT}}$  control is HIGH, AND gates through  $G_1$  through  $G_4$  are enabled, and the state of the  $Q$  output of each FF is passed through to the **D** input of the following FF. When the RIGHT /  $\overline{\text{LEFT}}$  control is LOW, AND gates  $G_5$  through  $G_8$  are enabled, and the  $Q$  output of each FF is passed through to the **D** input of the **preceding** FF. When a clock pulse occurs, the data are then effectively shifted one place to the left.

### 1.11.6 Applications of Shift Registers

Shift registers find an endless array of applications.

1. **Time Delay:** The serial-in-serial-out shift register can be used to provide a **time delay** from input to output that is a function of both the number of stages (**n**) in the register and the clock frequency.

When a data pulse is applied to the serial input, it enters the first stage on the triggering edge of the clock pulse. It is then shifted from one stage to another on each successive clock pulse till it appears on the serial output **n** clock periods later. The time can be adjusted up or down by changing the clock frequency. The time delay can also be increased by cascading shift registers and decreased by taking the output from successively lower stages in the register.

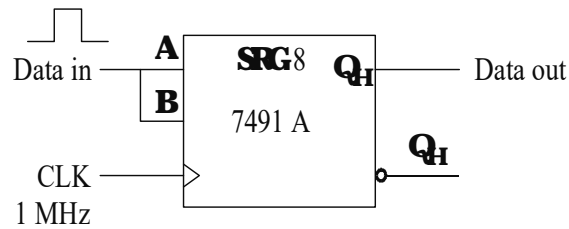


Fig. 1.47(a) **The Shift Register as a Time Delay Device**

2. **Ring Counter:** If the output is connected back to the serial input, a shift register can be used as a ring counter. This application is illustrated in Figure 1.48 using a 74195 IC 4-bit shift-register.

Initially, For example, a  $1000_2$  bit pattern can be synchronously preset into the counter by applying the bit pattern to the parallel data inputs and taking the SH / LD input LOW. After initialization, the 1 continues to circulate through the counter as shown in the diagram in Figure 1.49.

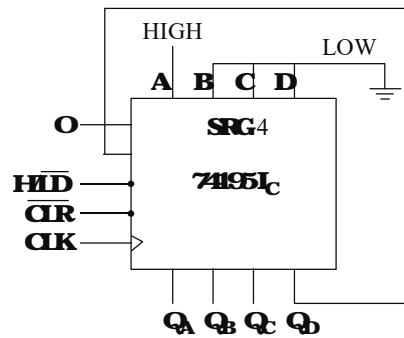


Fig. 1.48 **74195 Connected as a Ring Counter**

NOTES

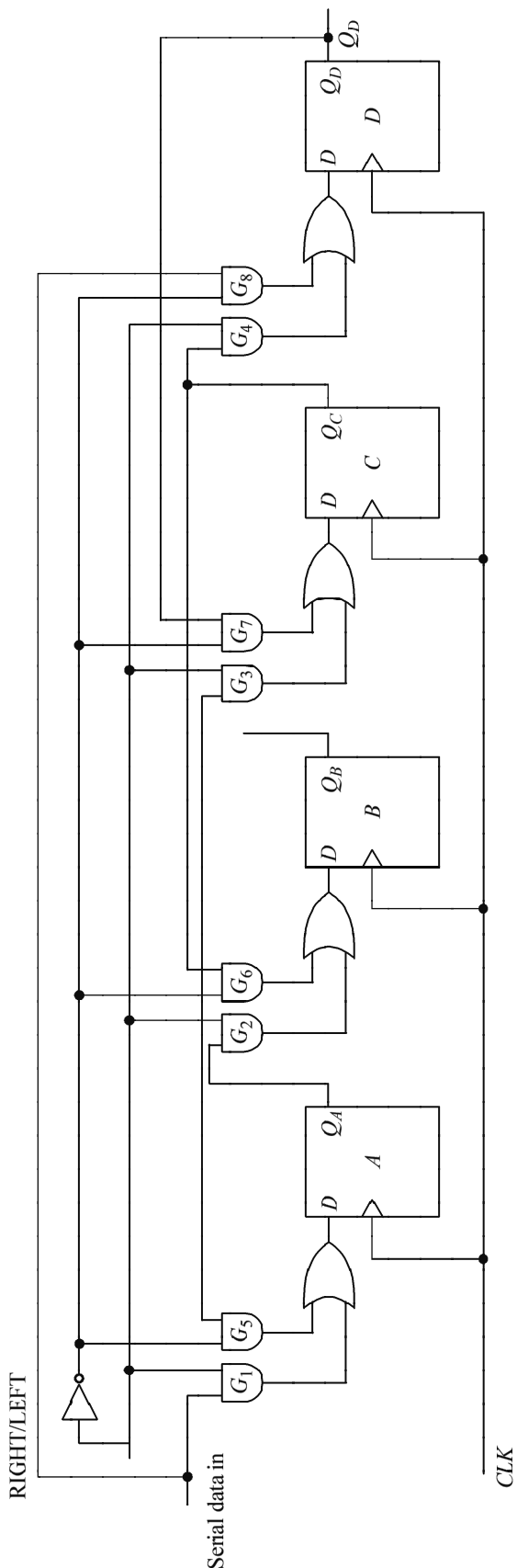


Fig. 1.47(b) Logic Diagram of a 4-Bit Bidirectional Shift Register

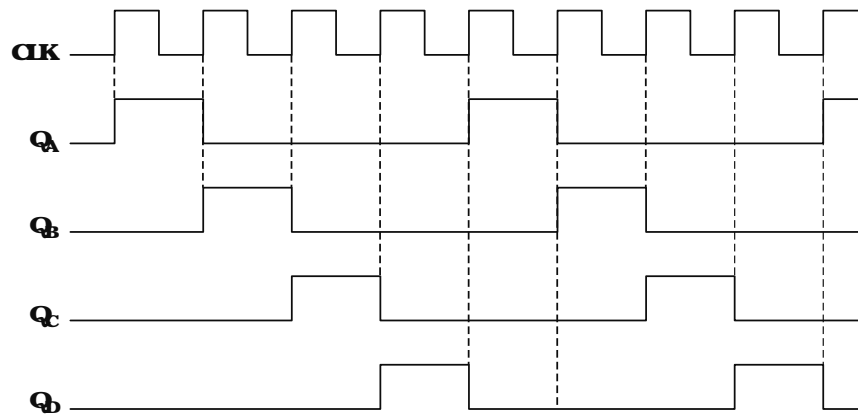


Fig. 1.49 **Timing Diagram Showing Two Complete Cycles of the Ring Counter when it is Initially Preset to  $100_2$**

3. **Serial-to-Parallel Data Converter:** Serial data transmission from one digital system to another is commonly used to reduce the number of wires in the transmission line. For example, eight bits can be sent serially over one wire, but it takes eight wires to send the same data in parallel.

A parallel format is a requisite for a computer or microprocessor based system because of incoming data to be in parallel format which is the requirement for serial-to-parallel conversion.

#### Check Your Progress

13. What are the broad categories of logic circuits?
14. Define registers.
15. What are two modes of operation for registers?

## 1.12 ANSWERS TO 'CHECK YOUR PROGRESS'

1. A number system that uses only two digits, 0 and 1 is called the Binary number system.
2. Mixed numbers are those numbers which contain both integer and fractional parts.
3. When you add or subtract numbers with decimal points where you have to first line up the decimals. Likewise, to add or subtract floating-point numbers, you will have to first modify the F of one of the two numbers by shifting left or right to make it equal to the E of the other number.
4. The following are the three possible techniques for representing signed integers:
  - Signed Magnitude Representation
  - Diminished Radix-Complement Representation
  - Radix-Complement Representation



5. Most of the processing in computers and other digital circuits are done in the binary formats. Various binary codes are used to represent data, which may be numerals, alphabets or special characters. A user must be very careful about the code being used while interpreting information available in the binary format. For example,

1000001 represents  $(65)_{10}$  in straight binary.

1000001 represents  $(41)_{10}$  in BCD.

1000001 represents A in ASCII code.

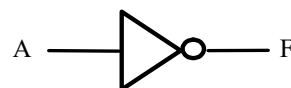
Some commonly used codes are as follows:

- Straight Binary Codes
  - Natural BCD Codes
  - Excess-3 Codes
  - Gray Codes
  - Alphanumeric Codes
  - Error Codes
6. The basic NOT gate has only one input and one output. The output is always the opposite or negation of the input. The following is the truth table for NOT gate:

A	F
0	1
1	0

Symbol:  $F = A'$

The following is the figure of NOT gate representation:



7. A truth table is a mathematical table that gives output of all combinations of inputs. These are used to compute the functional values of logical expressions. Every logical operation can be represented by its truth value. A truth table has number of columns equal to number of inputs plus one column for the output. If there are n-input variables, then there are  $2^n$  possible combinations since each variable can have two values 0 or 1.

8. Boolean addition: The basic rules of Boolean addition are given as follows:

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 1$$

Boolean addition is same as the logical OR addition.

## NOTES

9. De Morgan's two theorems are:

Theorem I:  $(A + B)' = A' \cdot B'$ .

Theorem II:  $(A \cdot B)' = A' + B'$

10. The main advantage in doing so is that, it then uses less logic gates and less power to realize and thus, it is considered sometimes cheaper and faster.

There are basically two types of simplification techniques:

- Algebraic Simplification
- Karnaugh Maps (K-map)

11. Combinational Circuits (CC) are those circuits where output depends on the present value of the inputs. If input values are changed, the information about the previous inputs is lost because combinational logic circuits have no memory. In such cases, sequential logic circuits are used to overcome this problem. In a combinational logic circuit the outputs depend on their current inputs. Combinational circuits are used to realize Boolean expressions.

12. Binary addition is performed in the same manner as decimal addition. Binary addition is the key to binary subtraction, multiplication and division.

13. In digital electronics, we have two broad categories of logic circuits. They are as follows:

- Combinational Circuit
- Sequential Circuit

**Combinational Circuit:** In a combinational circuit, each output depends entirely on the immediate (present) inputs to the circuit.

**Sequential Circuit:** In this type of circuit, the output depends on both the present and the past inputs. It means that this type of circuit involves the memory elements for storing past input conditions.

14. A register is a group of flip-flops used to store or manipulate data or both. Each flip-flop is capable of storing one bit of information. An **n**-bit register has **n** flip-flop and is capable of storing any binary information containing **n** bits.

The register is a type of sequential circuit and an important building block that is used in digital system like multipliers, dividers, memories, microprocessors, etc.

15. The first operation is series or serial operation. The second type of operation is parallel shifting. Input and output functions associated with registers include (1) Serial input/serial output (2) Serial input/parallel output (3) Parallel input/parallel output (4) Parallel input/serial output.

---

## 1.13 SUMMARY

---

- A number is an idea that is used to refer amounts of things. People use number words, number gestures and number symbols. Number words are

**NOTES**

said out loud. Number gestures are made with some part of the body, usually the hands. Number symbols are marked or written down. A number symbol is called a numeral.

- On hearing the word number, we immediately think of the familiar decimal number system with its 10 digits; 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. These numerals are called Arabic numerals.
- The number system which utilizes ten distinct digits, i.e., 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 is known as decimal number system. It represents numbers in terms of groups of tens
- A number system that uses only two digits, 0 and 1 is called the binary number system. The binary number system is also called a base two system.
- A binary number with 4 bits, is called a nibble and binary number with 8 bits is known as a byte.
- A binary fraction can be represented by a series of 1 and 0 to the right of a binary point. The weights of digit positions to the right of the binary point are given by  $2^{-1}$ ,  $2^{-2}$ ,  $2^{-3}$  and so on.
- Mixed Numbers: Mixed numbers contain both integer and fractional parts. The weights of mixed numbers are

$$2^3 \quad 2^2 \quad 2^1 \quad . \quad 2^{-1} \quad 2^{-2} \quad 2^{-3} \quad \text{etc.}$$

↑

Binary Point

- When you add or subtract numbers with decimal points where you have to first line up the decimals. Likewise, to add or subtract floating-point numbers, you will have to first modify the F of one of the two numbers by shifting left or right to make it equal to the E of the other number.
- For positive numbers, this presentation is the same as the unsigned binary representation for any number.
- Most of the processing in computers and other digital circuits are done in the binary formats. Various binary codes are used to represent data, which may be numerals, alphabets or special characters. A user must be very careful about the code being used while interpreting information available in the binary format.
- Gray code is a very useful code in which a decimal number is represented in the binary form in such a way that each Gray code differs from the preceding and the succeeding numbers by a single bit. It is not a weighted code. It is also known as reflected code.
- The MSB of binary code is the same as the MSB of the corresponding Gray code.
- A basic AND gate consists of two inputs and an output. In the AND gate, the output is 'High' or gate is 'On' only if both the inputs are 'High'. The relationship between the input signals and the output signals is often represented in the form of a truth table.

**NOTES**

- A basic OR gate is a two input, single output gate. Unlike the AND gate, the output is 1 when any one of the input signals is 1. The OR gate output is 0 only when both the inputs are 0.
- Basic logic functions and operations are the AND function (logical multiplication), the OR function (logical addition) and the NOT operation (logical complementation).
- The logical inverse operation changes logical 1 to logical 0 and vice versa. It is also called the NOT operation.
- A Boolean function is an algebraic expression formed using binary constants, binary variables and basic logic operation symbols.
- A great mathematician De Morgan has contributed with two of the most important theorems of Boolean algebra. De Morgan's theorems are extremely useful in simplifying expression in which product or sum of variables are complemented.
- Combinational logic deals with the techniques of 'Combining' the basic gates into circuits that perform some desired functions. In combinational logic circuits, at any time, the logic level at the output depends on the combination of logic levels present at the inputs. A combinational circuit has no memory characteristic, so its output depends only on the current value of its inputs.
- The AND function is referred to as a product. In Boolean algebra, the word 'Product' loses its original meaning but serves to indicate an AND function. The logical product of several variables, on which a function depends, is considered to be a product term. The variables in a product term appear in a complemented or uncomplemented form. For example,  $\overline{A}BC$  is a product term.
- An OR function is generally referred to as a sum. The logical sum of variables on which a function depends is considered to be a sum term. Variables in a sum term can appear either in complemented or uncomplemented form. For example,  $A + \overline{B} + C$  is a sum term.
- A minterm is a special case product (AND) term. A minterm is a product term that contains all of the input variables that make up a Boolean expression. A two variable function has four possible combinations, viz.,  $\overline{A}\overline{B}$ ,  $\overline{A}B$ ,  $A\overline{B}$  and  $AB$ .
- A maxterm is a special case sum (OR) term. A maxterm is a sum (OR) term that contains all of the input variables that make up a Boolean expression. A two variable function has four possible combinations, viz.  $A + B$ ,  $A + \overline{B}$ ,  $\overline{A} + B$  and  $\overline{A} + \overline{B}$ .
- There are basically two types of simplification techniques:
  - o Algebraic Simplification
  - o Karnaugh Maps (K-map)

- Combinational Circuits (CC) are those circuits where output depends on the present value of the inputs. If input values are changed, the information about the previous inputs is lost because combinational logic circuits have no memory. In such cases, sequential logic circuits are used to overcome this problem. In a combinational logic circuit the outputs depend on their current inputs. Combinational circuits are used to realize Boolean expressions.
- An electronic (combinational) circuit which performs the arithmetic addition of two binary digits is called a half-adder.
- Binary addition is performed in the same manner as decimal addition. Binary addition is the key to binary subtraction, multiplication and division.
- In digital electronics, we have two broad categories of logic circuits. They are as follows:
  - o Combinational Circuit
  - o Sequential Circuit
- Combinational Circuit: In a combinational circuit, each output depends entirely on the immediate (present) inputs to the circuit.
- Sequential Circuit: In this type of circuit, the output depends on both the present and the past inputs. It means that this type of circuit involves the memory elements for storing past input conditions.
- Time Delay: The serial-in-serial-out shift register can be used to provide a time delay from input to output that is a function of both the number of stages ( $n$ ) in the register and the clock frequency.

---

## 1.17 KEY TERMS

---

- **Arabic Numerals:** On hearing the word number, we immediately think of the familiar decimal number system with its 10 digits; 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. These numerals are called Arabic numerals.
- **Binary System:** A number system that uses only two digits, 0 and 1 is called the binary number system. The binary number system is also called a base two system.
- **Binary Fraction:** A binary fraction can be represented by a series of 1 and 0 to the right of a binary point. The weights of digit positions to the right of the binary point are given by  $2^{-1}$ ,  $2^{-2}$ ,  $2^{-3}$  and so on.
- **Mixed Numbers:** Mixed numbers contain both integer and fractional parts. The weights of mixed numbers are
 

$2^3$	$2^2$	$2^1$	.	$2^{-1}$	$2^{-2}$	$2^{-3}$	etc.
	↑						
	Binary Point						
- **Gray Codes:** It is a very useful code in which a decimal number is represented in the binary form in such a way that each Gray code differs

## NOTES

from the preceding and the succeeding numbers by a single bit. It is not a weighted code. It is also known as reflected code.

- **Truth Table:** A basic AND gate consists of two inputs and an output. In the AND gate, the output is 'High' or gate is 'On' only if both the inputs are 'High'. The relationship between the input signals and the output signals is often represented in the form of a truth table.
- **Product Term:** The AND function is referred to as a product. In Boolean algebra, the word 'Product' loses its original meaning but serves to indicate an AND function. The logical product of several variables, on which a function depends, is considered to be a product term. The variables in a product term appear in a complemented or uncomplemented form. For example,  $\overline{A}BC$  is a product term.
- **Minterm:** A minterm is a special case product (AND) term. A minterm is a product term that contains all of the input variables that make up a Boolean expression. A two variable function has four possible combinations, viz.,  $\overline{A}\overline{B}$ ,  $\overline{A}B$ ,  $A\overline{B}$  and  $AB$
- **Maxterm:** A maxterm is a special case sum (OR) term. A maxterm is a sum (OR) term that contains all of the input variables that make up a Boolean expression. A two variable function has four possible combinations, viz.  $A + B$ ,  $A + \overline{B}$ ,  $\overline{A} + B$  and  $\overline{A} + \overline{B}$ .
- **Half-Adder:** An electronic (combinational) circuit which performs the arithmetic addition of two binary digits is called a half-adder.
- **Combinational Circuit:** In a combinational circuit, each output depends entirely on the immediate (present) inputs to the circuit.
- **Sequential Circuit:** In this type of circuit, the output depends on both the **present** and the **past** inputs. It means that this type of circuit involves the memory elements for storing **past** input conditions.
- **Time Delay:** The serial-in-serial-out shift register can be used to provide a **time delay** from input to output that is a function of both the number of stages (**n**) in the register and the clock frequency.

---

## 1.15 SELF-ASSESSMENT QUESTIONS AND EXERCISES

---

### Short-Answer Questions

1. What is a number system?
2. Write basic features of floating point representation.
3. Define the integer representation.
4. What are the different types of character codes?

5. What is a logic gate?
6. Define OR/AND operation.
7. Write the Boolean expression for a five-input OR gate.
8. What do you mean by combinational circuit?
9. What is a registers?
10. Give the features of counters.

## NOTES

### Long-Answer Questions

1. Write explanatory notes on the four systems of arithmetic.
2. Explain floating-point arithmetic operations.
3. Briefly discuss the concept of decimal arithmetic operations giving examples.
4. Write explanatory notes on:
  - (a) Signed magnitude representation
  - (b) Radix-complement representation
5. Write a note on character codes. Explain with the help of examples.
6. What is a logic gate? Explain the basic logic operations of Boolean algebra.
7. Explain half-adder using XOR and AND gates with the help of logic diagram. Write its truth tables.
8. Discuss the applications of boolean algebra.
9. Describe De Morgan's theorems.
10. Explain the two types of simplification techniques of Boolean expression.
11. What is an arithmetic circuit? Explain in detail with the help of examples.
12. Differentiate between the combinational circuits and sequential circuits.
13. Discuss the significance and application of registers giving details of each types.
14. Explain the two modes of operation for registers.

---

## 1.16 FURTHER READING

---

- Tanenbaum, Andrew S. 1999. **Structured Computer Organization**, 4th Edition. New Jersey: Prentice-Hall Inc.
- Mano, M. Morris. 1993. **Computer System Architecture**, 3rd Edition. New Jersey: Prentice-Hall Inc.
- Bartee, Thomas C. 1985. **Digital Computer Fundamentals** New York: McGraw-Hill.
- Mano, M. Morris. 1979. **Digital Logic and Computer Design** New Delhi: Prentice-Hall of India.

**NOTES**

Leach, Donald P. and Albert Paul Malvino. 1994. **Digital Principles and Applications** New York: McGraw-Hill.

Mano, M. Morris. 2002. **Digital Design** New Delhi: Prentice-Hall of India.

Kumar, A. Anand. 2003. **Fundamentals of Digital Circuits** New Delhi: Prentice-Hall of India.

Stallings, William. 2007. **Computer Organization and Architecture** New Delhi: Prentice-Hall of India.



---

# UNIT 2 REGISTER, MICRO- OPERATIONS AND DESIGN CONCEPTS

---

*Register, Micro-  
Operations and  
Design Concepts*

## NOTES

### Structure

- 2.0 Introduction
- 2.1 Objectives
- 2.2 Register Transfer
- 2.3 Bus System
  - 2.3.1 Bus Organization
  - 2.3.2 Multiple Bus Organization
- 2.4 Micro-Operations
  - 2.4.1 Arithmetic Micro-Operations
  - 2.4.2 Logic Micro-Operations
  - 2.4.3 Shift Micro-Operations
- 2.5 Instruction and Instruction Code
  - 2.5.1 Instruction Execution
  - 2.5.2 Binary Coded Decimal (BCD) Code
  - 2.5.3 Excess-3 Code
  - 2.5.4 Gray Code
  - 2.5.5 Alphanumeric Codes
  - 2.5.6 Error-Detecting Codes
  - 2.5.7 Error-Correcting Codes
  - 2.5.8 Hamming Codes
- 2.6 Computer Instruction
  - 2.6.1 Instruction Representation
- 2.7 Timing and Controls
  - 2.7.1 Functions of Control Unit
  - 2.7.2 Instruction Cycle
- 2.8 Memory Reference Instructions
  - 2.8.1 Memory Reference Format
- 2.9 Input/Output and Interrupts
- 2.10 Complete Computer Description
  - 2.10.1 Basic Anatomy of the Computer
  - 2.10.2 Data Representation within the Computer
  - 2.10.3 Design of a Basic Computer
  - 2.10.4 Components of a Computer System
  - 2.10.5 Machine Language
- 2.11 Answers to 'Check Your Progress'
- 2.12 Summary
- 2.13 Key Terms
- 2.14 Self-Assessment Questions and Exercises
- 2.15 Further Reading

---

## 2.0 INTRODUCTION

---

In hardwired control which includes gates, flip-flops, decoders, multiplexers and other digital circuits. A complete processor includes global and local descriptor tables which are found in the memory systems. Microprogrammed unit includes

## NOTES

the fundamental components to perform the microinstructions. Basically, microinstructions are the inputs to the hardwired control unit.

The fundamental concept of bus structure, a shared communication path consisting of one or more connection lines is known as a bus and the transfer of data through this bus is known as bus transfer. The CPU communicates with the other components via a bus. Data bus, address bus and control bus are the types of bus. A set of instructions is called a program in a computer. These instructions are decoded and executed in Arithmetic Logic Unit (ALU) with the help of registers. A computer is defined on the set of registers used and operations that are performed on the data stored in them. The operations executed on the data stored in registers are called micro-operations. Micro-operations are considered fundamental or primitive (usually atomic) operations carried out in the computer. An instruction is a command given to a computer to perform a specified operation on some given data. A code on the other hand is a symbol or a group of symbols that stands for something. The Central Processing Unit (CPU), acts as the brain of computer, and controls other peripherals and interfaces.

In this unit, you will study about the register transfer, bus system, micro-operations, instruction and instruction code, computer instruction, timing and controls, instruction cycle, memory reference instruction, input/output and interrupts, complete computer description, machine language and design of basic computer.

---

## 2.1 OBJECTIVES

---

After going through this unit, you will be able to:

- Explain the organization of registers in different computers and register transfers
- Elaborate on the bus structure
- Know about micro-operations
- Understand instruction and instruction code
- Know computer instructions
- Describe instruction cycle
- Explain memory reference instructions
- Learn about the input/output and interrupts
- Define the term machine language
- Understand about the design of a Computer System

---

## 2.2 REGISTER TRANSFER

---

A digital system is a sequential logic system in which flip-flops and gates are constructed. The register transfer logic methods focus on how adders, decoders and registers use expressions and statements which resembles the statements used in programming language. High level language C supports register transfer technique

for executing applications. It encompasses all types of registers such as shift registers, counters and memory units. Here, a counter is incremented by one and the memory unit is considered as a collection of storage registers.

In register transfer operations, the straight forward register transfers the data from register to another register temporarily. For example, the data is transferred from register R3 to register R1. It is shown symbolically as follows:

$$R1 \leftarrow R3$$

The left arrow ( $\leftarrow$ ) is used to show that data from the right is going to move to the left side register. In the digital system, registers are attached to each other which makes it possible that more than one register can be transferred simultaneously. More registers are separated by comma but they are kept on the same line. It is done in the following way:

$$R1 \leftarrow R3, R2 \leftarrow R5$$

The above statement shows that the contents of register R3 are transferred to register R1 and the contents of register R5 are transferred to register R2 at the same time.

Sometimes, register transfer operation depends on certain conditions. For example, register  $R1 \leftarrow R3$  takes place only if Boolean variable  $k = 1$  is satisfied. In a programming language, it is coded as follows:

$$\text{if } (k=1) \text{ then } R1 \leftarrow R3$$

Inter-register microoperations do not change the information content if the binary data and information moves from one register to another register. The characteristics of microoperations are as follows:

- Arithmetic microoperations perform arithmetic or number operations; logic performs AND, OR, XOR operation; and shift microoperations perform shift register.
- The register is designated by capital letters and sometimes followed by numerals, such as R1, R2, IR, etc. The flip-flops of an n-bit register are numbered from 1 to n (or from 0 to n-1) starting either from the left or from the right.

Simple digital system contains the combinational and sequential circuits. They are characterized as follows:

- The type of registers they contain.
- The operations they perform.

Typically, register transfer focuses on the operations of data which are passed into different registers. These operations are called microoperations. The main functions which take place in the register are as follows: Shift, Load, Clear and Increment.

During one clock pulse, the information which is operated and stored in different registers is performed under elementary operation. Table 2.1 shows the transfer functions that are used in transferring registers:

## NOTES

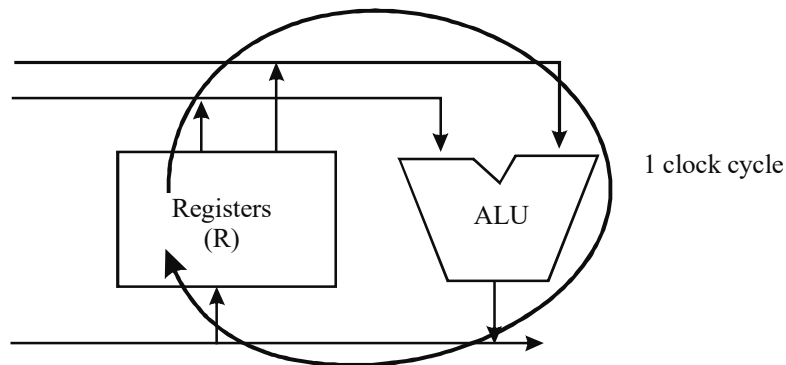
**NOTES**

**Table 2.1** Transfer Functions that are used in Transferring Registers

Function Name	Purpose	Prototype
<code>__gpr_to_d64</code>	Transfer from general purpose register to floating point register	<code>_Decimal64 __gpr_to_d64 (long long);</code>
<code>__gprs_to_d128</code>	Transfer from general purpose register to floating point register. Transfers a value from a pair of general purpose registers (64-bit mode) or four general purpose registers (32-bit mode).	<code>_Decimal128 __gprs_to_d128 (unsigned long long*upper, unsigned long long*lower);</code>
<code>__d64_to_gpr</code>	Transfer from floating point register to general purpose register. Transfers a value from a floating point register to a general purpose register (64-bit mode) or a general purpose register pair (32-bit mode).	<code>long long __d64_to_gpr (_Decimal64);</code>
<code>__d128_to_gprs</code>	Transfer from floating point register to general purpose register. Transfers a value from a pair of floating point registers to a pair of general purpose registers (64-bit mode) or four general purpose registers (32-bit mode).	<code>void __d128_to_gprs (_Decimal128, unsigned long long*upper, unsigned long long*lower);</code>

In Figure 2.1,  $f(R,R)$  function has two parameters which denotes different functions as follows:

**f:** shift, load, clear, increment, add, subtract, complement, AND, OR, XOR.



**Fig. 2.1** One Clock Cycle of  $R \leftarrow f(R, R)$

**Organization of a Digital System**

Digital systems contain the set of registers and their functions in the internal organization of the computer. The main function is that they control signals to initiate the sequence of microoperations to perform the functions. It maintains the way of register transfer on any digital system and therefore it is called register transfer level. The characteristics are as follows:

- It depends on system registers.
- Information/data is transferred on different registers.

## Designation of Resistors

Designation of resistors is interrelated with the register transfer facility. It enhances the transferring rate of data which is stored in registers. The characteristics of designation of resistors are as follows:

- Registers are represented by capital letters, such as A, R13, IR.
- The variable names indicate the functions which are used as follows:
  - o MAR indicates Memory Address Register
  - o PC indicates Program Counter
  - o IR indicates Instruction Register

Contents of registers are viewed and designated in various ways. Basically, a register is viewed as a single entity and processes the bits of data it contains.

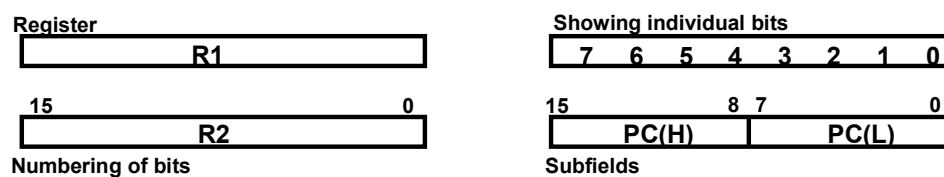


Fig. 2.2 Block Diagram of a Register

In Figure 2.2,  $R2 \leftarrow R1$  depicts that the contents of register R2 are copied and loaded into register R1. It transfers simultaneously from the source register R1 to the destination register R2 during one clock pulse. This is called non-destructive method. In this method, the contents of register R1 are not altered during the operation of copying or loading to register R2. It performs step by step. It first copies the contents of one register and then transfers to another register. Let us take another example. A statement is written as follows:

$$R3 \leftarrow R5$$

It implies that the data lines move from the source register R5 to the destination register R3. It loads parallel in the destination register R3. The control lines are used to perform the operation.

## Control Functions of Register Transfer

The control functions of register transfer are as follows:

- If a certain condition is true, microoperation is activated as per requirement.
- In register transfer, control function is similar as 'if' statement in a programming language.
- Control functions use control signal to perform microoperations. If the control signal comes as 1, the operation takes place.

The statement is represented as follows:

$$P: R2 \leftarrow R1$$

The above statement tells that if  $P=1$ , then load the contents of register R1 into register R2. This statement is written in programming language as follows:

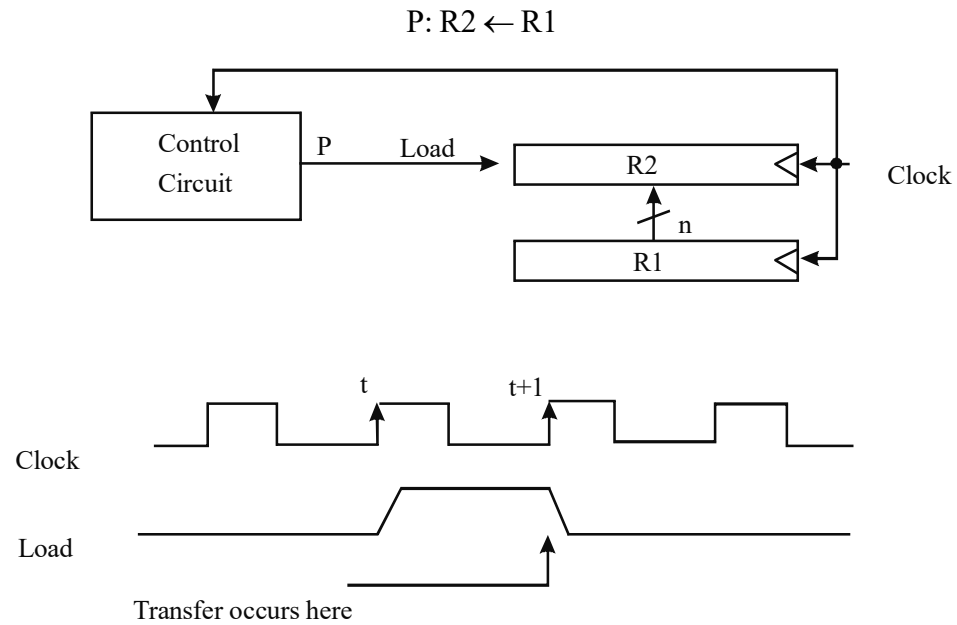
$$\text{if}(P=1) \text{ then } (R2 \leftarrow R1)$$

## NOTES

### Implementation of Controlled Transfer

The control transfer passes the functions via control circuit. It is represented as follows:

**NOTES**



**Fig. 2.3** Block and Timing Diagram

Figure 2.3 shows that the same clock controls the circuits that generate the control function. It is incremented by one such as  $t$ ,  $t+1$  and so on and then reaches to the destination register. The transfer occurs in Load process. Registers are assumed to use positive edge triggered flip flops.

There are four types of connections or links between components, such as bus, control, Boolean and miscellaneous. The bus is a general purpose data and control connection passing synchronized information between components. Control connections are used to link the control components and also to invoke activity in other components. Boolean connections are special purpose signals aiding the execution flow by connecting conditions with decision making control components. Finally, there exist miscellaneous connections to connect these register transfer components with external devices.

**Table 2.2** Working Registers

Name	Function	Bits	States
W	<u>W</u> -Register	32	–
X	<u>X</u> -Register	32	–
Y	<u>Y</u> -Register	32	–
A	<u>A</u> ddress Register	12	4096
C	<u>C</u> ommand Instruction Register	6	64
PSN	<u>P</u> rogram <u>S</u> tep <u>N</u> umber	8	256
ID	<u>I</u> nformation <u>D</u> ecoder	10	1024
OSP	<u>O</u> utput <u>S</u> tart <u>P</u> oint (Index Register)	5	32

Table 2.2 shows the working registers and their functions.

Following are the properties of register transfer:

- Information transfer from one register to another (transfer of the content of register R1 into register R2). The content of the source register R1 does not change after the transfer.
- The transfer occurs only under a predetermined control condition. The transfer operation is executed by the hardware only if  $P=1$ .
- A comma is used to separate two or more operations (executed at the same time)  $R1, R2$ .

## NOTES

## 2.3 BUS SYSTEM

A shared communication path consisting of one or more connection lines is known as a bus and the transfer of data through this bus is known as bus transfer. When data is read from or stored in memory, it is referred to as memory transfer.

The functional components of a computer must be connected in order to make a system operational. The CPU (Control Processing Unit) communicates with the other components via a bus. A bus is a set of wires that acts as a shared but common data path to connect multiple subsystems within the computer system. It consists of multiple lines, allowing the parallel movement of bits. Buses are low cost but very versatile and help connect devices with each other as well as the system. At any given point in time, only one device (be it a register, the ALU, memory or some other component) may use the bus. However, this sharing often results in a communications bottleneck. The speed of the bus is affected by its length as well as by the number of devices sharing it. Following are the types of bus:

- **Data Bus:** It is used for the transmission of data. Data lines and the number of bits in a word are similar.
- **Address Bus:** It carries the address of the main memory location from where data can be accessed.
- **Control Bus:** It is used to indicate the direction of data transfer and to coordinate the timing of events during the transfer.

A digital computer consists of many processor registers and the transfer of information from one register to another is often required. Hence, paths must be provided so that such transfer operations can take place. Figure 2.4 shows the transfer among three registers R1, R2 and R3 through six data paths.

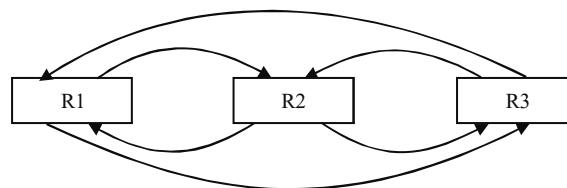
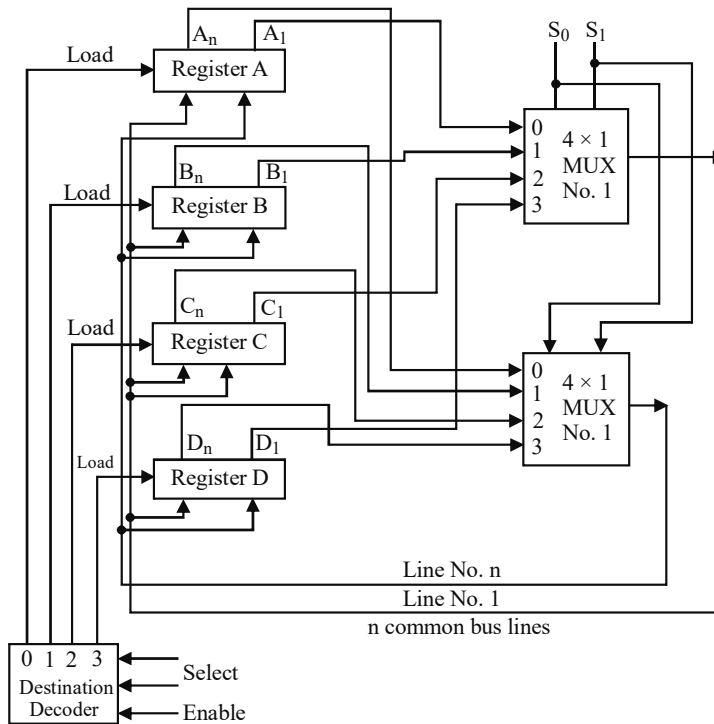


Fig. 2.4 Transfer among Three Registers

If different lines are used involving each register, the number of wires will increase considerably. Hence, a pair of common lines, one line for each bit of the register, is

**NOTES**

used for the transfer. This set of common lines through which binary data is transferred, one at a time, among registers is known as a *bus*. A common bus system is constructed with the help of multiplexers and decoders. The multiplexer selects the source register whose binary information is then placed on the bus and the decoder selects one destination register to transfer the information to, from the bus. The construction of a bus system for four registers is shown in Figure 2.5. Two multiplexers have been used, one for the low order significant bit and one for the high order significant bit. If the register is of  $n$  bits,  $n$  multiplexers are required to produce  $n$  bus lines. These  $n$  lines in the bus are connected to  $n$  inputs of all the registers.



**Fig. 2.5** Bus System for Four Registers

$S_1$  and  $S_0$  are selection lines connected to selection inputs of all  $n$  multiplexers. The selection lines choose  $n$  bits of one register and transfers these to the common bus  $n$  lines. When  $S_1$  and  $S_0 = 00$ , the 0 data inputs of all  $n$  multiplexers are selected and cause the  $n$  bits from register A to transfer to the  $n$ -line common bus, since the output of this register is connected to 0 data inputs of each multiplexer. Similarly, when  $S_1$  and  $S_0 = 01$ , the content of register B is transferred into the  $n$ -line common bus and so on. The register that is selected for the four possible binary values is shown in Table 2.3.

**Table 2.3** Function Table for Bus in Figure 2.5

$S_1$	$S_0$	Register Selected
0	0	A
0	1	B
1	0	C
1	1	D



The shifting of data from a bus to one of the targeted registers is done with the help of the load control of that register. The load control of the particular register is activated by the outputs of the decoder when enabled. If the decoder is not enabled then no information from the bus will be transferred to the register although the multiplexers place the information of the source register onto the bus.

In general, for registers of  $n$  bits,  $n$  multiplexers are needed to construct a bus of  $n$  lines. The size of a multiplexer depends on the number of registers in the system. If there are  $K$  registers, the multiplexer's size will be  $K \times 1$  since it multiplexes  $K$  data lines. To take an example, a general bus of 16 registers of 16 bits each needs 16 multiplexers of size  $16 \times 1$ . Four selection lines are required. Also, the size of the destination decoder will be  $4 \times 16$ .

Consider the following statement:

$$C \leftarrow B$$

The control function that enables this transfer must select register  $B$  as the source and register  $C$  as the destination registers. The content of register  $B$  is located on the bus and the content of the bus is then transferred to register  $C$  by starting its load control input.

### 2.3.1 Bus Organization

A bidirectional bus for carrying data between two units is called a *data bus*. A unidirectional bus used to carry memory addresses is called memory bus.

The manner in which different buses are connected to form a common bus so that the CPU, memory and I/O devices can use the common bus, when required, is called *bus organization*.

A basic computer consists of a memory unit, a control unit and registers. There must be a path that can be used to transfer information between the memory and the registers or among registers. Using a common bus is the most efficient way of transferring information from source to destination in a system with multiple registers. Figure 2.6 shows the connection of eight registers and a  $4096 \times 16$  memory unit of a common bus system. The eight registers are the Address Register (AR), Program Counter (PC), Data Register (DR), Accumulator (AC), Instruction Register (IR), Temporary Register (TR), Input Register (INPR) and Output Register (OUTR). Here, a 16-bit common bus has been used.

The outputs of seven registers and memory are linked to the common bus. The definite output chosen for the bus lines at any time is finalized by the binary value of the selected lines  $S_2$ ,  $S_1$  and  $S_0$  as shown in Table 2.4. The numbers along each output line shows the decimal equivalent of the required binary selection. When  $S_2S_1S_0 = 011$ , the 16-bit outputs of DR are placed on the bus lines.

The lines from the common bus are linked to the input of each register and the data inputs of the memory. The specific register whose LD (load) input is allowed gets the information from the bus. The memory gets the information from the bus when 'A write Input' is allowed.

The memory puts its results on to the bus when the 'Read Input' is on and  $S_2S_1S_0 = 111$ .

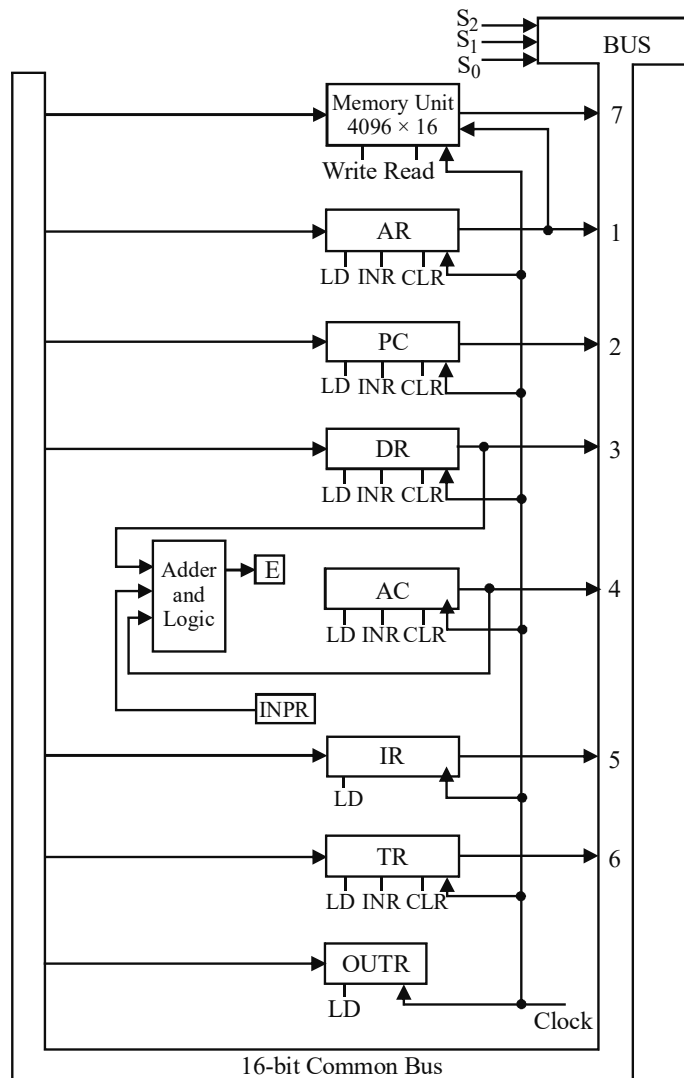
## NOTES

**NOTES**

The registers DR, AC, IR and TR are of 16-bits. Two registers, PC and AR, have 12 bits since they store addresses.

When the contents of AR and PC are placed on the bus, the four most significant bits are set to 0. When AR and PC receive data from the common bus, only the 12 least significant bits are transferred to the register.

The input registers INPR and OUTR have 8 bits because they communicate only with the 8 least significant bits in the bus. INPR is connected to the bus for providing information. However, OUTR is connected to the bus only for receiving information from the bus. INPR receives a character from the I/O device which is transferred to AC and OUTR receives a character from AC and delivers it to an output device. No transfer takes place from OUTR to any of the other registers.



**Fig. 2.6 Common Bus Organization**

The 16-bit common bus receives information from six registers and the memory unit. In addition, the 16-bit common bus is linked to the inputs of six registers and the memory unit. Five registers have three control signals—LD (load), INR (increment) and CLR (clear). Two registers have only LD (load) control signal

connected to the common bus. AR is also connected to the memory address. Thus, AR always specifies the memory address. During a memory write operation, the content of any register can be specified for the memory data and similarly during a memory read operation, any register except AC can receive data from the memory. The 16-bit AC receives inputs from the adder and logic circuit, which receives input from three registers. These three registers are 16-bit AC, 16-bit data register DR and 8-bit inputs, which come from input register INPR. The inputs from DR and AC are used for arithmetic and logic micro-operations. Table 2.4 shows the binary value of selection line  $S_2S_1S_0$  that selects one of the registers.

## NOTES

**Table 2.4** Function Table

$S_2$	$S_1$	$S_0$	Register
0	0	1	AR
0	1	0	PC
0	1	1	DR
1	0	0	AC
1	0	1	IR
1	1	0	TR
1	1	1	Memory

For example, in order to transfer the contents of PC to AR (Address Register), the computer requires the following instructions:

- Set the selection variables  $S_2S_1S_0 = 010$ .
- Transfer the contents of PC to the bus.
- Enable LD input of AR.
- Transfer contents of bus into AR.

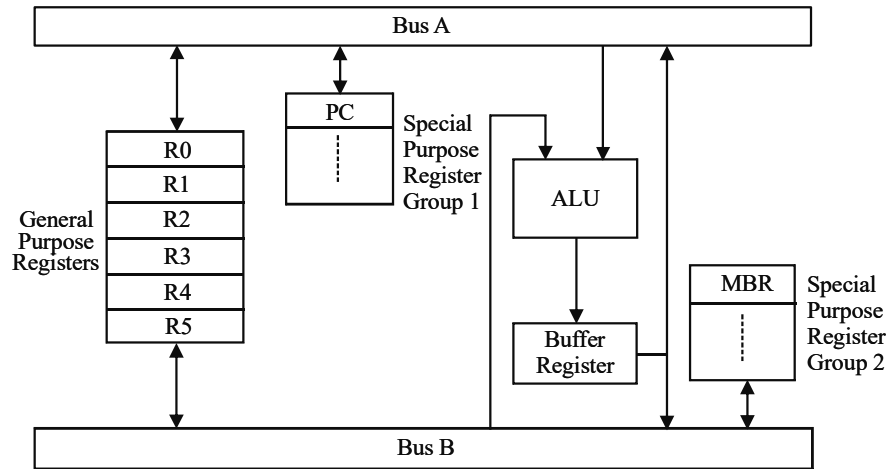
### 2.3.2 Multiple Bus Organization

A two bus structure used to connect the registers and the ALU of a processor is shown in Figure 2.7. All general purpose registers are connected to both buses A and B to form a two bus organization. The two operands required by the ALU are routed in one clock cycle hence, the execution of instruction becomes faster since the ALU does not wait for the second operand, as is the case with single bus organization. Information passed on to the bus may be from general purpose registers or special purpose registers. In addition, the special purpose registers are divided into two groups—one group at the left of the ALU connected to bus A and the other group is at the right of the ALU connected to bus B. The data from two special purpose registers belonging to the same group cannot be transferred to the ALU at the same time.

The output of the ALU may be routed to either general purpose registers or special purpose registers. The ALU does not have any input buffer register and hence, both buses will be busy in carrying the operands during the binary operations. Therefore, the output of ALU is first stored in the output register. Transfer of the required operands and loading of the ALU output buffer register take place in one

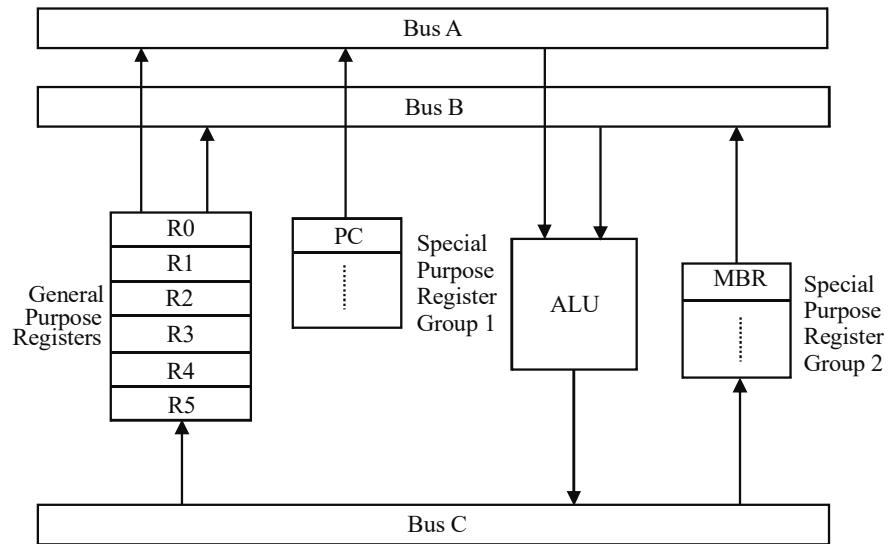
clock cycle. The content of the ALU output register is routed to the destination with the help of either bus A or bus B in the second clock cycle.

**NOTES**



*Fig. 2.7 Two Bus Organization of the Data Path*

The performance of a two bus organization can be further improved by adding a third bus C at the output of ALU. The three bus structure is shown in Figure 2.8. The addition of a third bus allows the system to perform operation, such as  $R3 \leftarrow R1 + R2$  in one clock cycle as there are three separate buses in the system.



*Fig. 2.8 Three Bus Organization of the Data Path*

## 2.4 MICRO-OPERATIONS

An instruction constitutes a set of micro-operations. You can define a micro-operation as an elementary operation that is performed on the information stored in one or more registers during one clock pulse.

The result of the operation may replace the previous binary information of a register or may be transferred to another register. A micro-operation requires only one clock pulse for execution if the operation is done in parallel.

The micro-operations most often encountered in digital computers are classified into the following four categories:

- 1. Register Transfer Micro-Operation:** A micro-operation that transfers binary information from one register to another.
- 2. Arithmetic Micro-Operation:** A micro-operation that performs arithmetic operations on numeric data stored in registers.
- 3. Logical Micro-Operation:** A micro-operation that performs bit manipulation operations on non-numeric data stored in registers.
- 4. Shift Micro-Operation:** A micro-operation that performs shift operations on data stored in registers.

Arithmetic, logical and shift operations are usually performed by the ALU unit. Apart from arithmetic and logical operations, the computer has to implement some more micro-operations on the registers. These include the following:

- Refresh volatile data
- Load (Store) data
- Clear storages (change all bits to 0)
- Increment (and decrement) storages binarily
- Complement storages
- Select individual storage bits
- Counter with parallel load is capable of performing the micro-operations increment and load.
- A bidirectional shift register is capable of performing the shift right and shift left micro-operations.

In order to execute these operations, the register, which is basically a set of flip-flops, has some additional combinational circuits that enable it to execute the listed operations.

Another possible microoperation that changes the information content of the registers can be shown by the following example in the following micro-operation:

$$R1 \leftarrow R1 + R2$$

The content of R1 and R2 registers are provided as an input of Adder circuit in ALU and the result is transferred to R1. Note that the result will overwrite the previous value of R1. However, the register transfer micro-operation will not change the content but just transfer information from source to destination register.

### 2.4.1 Arithmetic Micro-Operations

The basic arithmetic micro-operations are addition, subtraction, increment, decrement, complement of register content, arithmetic shift addition with carry-over subtraction, etc.

Let us describe the following arithmetic operation

$$R1 \leftarrow R2 + R3$$

This instruction specifies an addition operation and states that the contents of register R2 are added to the contents of register R3 and the resultant sum is transferred to register R1 while the content of R2 and R3 remain unchanged. To

## NOTES

implement this statement with hardware we need three registers and the digital component that performs the addition operation.

The basic arithmetic operations are listed in the Table 2.5:

**NOTES**

*Table 2.5 Basic Arithmetic Operations*

Symbolic Designation	Description
$R3 \leftarrow R1 + R2$	Contents of R1 plus R2 transferred to R3
$R3 \leftarrow R1 - R2$	Contents of R1 minus R2 transferred to R3
$R2 \leftarrow R2'$	1's complement of the contents of R2
$R2 \leftarrow R2' + 1$	2's complement of the contents of R2 (Negate)
$R3 \leftarrow R1 + R2' + 1$	R1 plus the 2's complement of R2 (Subtraction)
$R1 \leftarrow R1 + 1$	Increment the contents of R1 by one
$R1 \leftarrow R1 - 1$	Decrement the contents of R1 by one

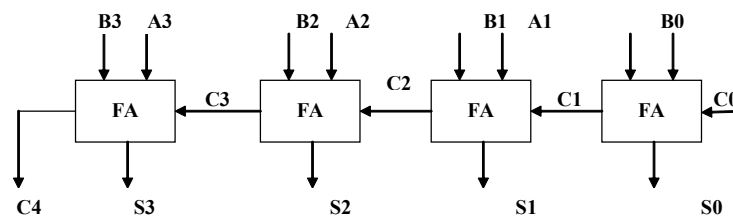
Although multiplication and division are also valid arithmetic operations they are not included in the basic set of micro-operations. This is so, because, in most computers, the multiplication operation is implemented with a repeated addition. Shift micro-operations and division are implemented with a sequence of subtractions and shift micro-operations. To specify the hardware in such a case requires a list of statements that use the basic micro-operations, add, subtract and shift.

**Hardware for Implementing the Arithmetic Operations**

Various arithmetic operations can be computed by adder and subtractor circuits. Multiplication and division can be implemented through successive additions and subtractions respectively. For arithmetic operations a  $n$  bits input comes from one register A and the  $n$  bits input come from another register B. The output is transferred to a third register or to one of the source registers replacing its previous content.

**Binary Adder (for Addition)**

The digital circuit that gives the arithmetic sum of two binary numbers (of any length ) is called a binary adder. The binary adder is constructed by cascading full-adder circuits. An  $n$ -bit binary adder requires  $n$  full adders. For example, to design a four-bit binary adder we need four full adders which are connected such that the output carry from one full-adder is connected as the input of the next full adder. A 4-bit full adder circuit is shown in Figure 2.9. This also called ripple adder.



*Fig. 2.9 4-Bit Binary Adder*

### Binary Adder (For Subtraction)

Negative numbers are represented in complement notation. Hence the subtraction process is also performed by complement method. Binary numbers can be subtracted by adding the complements of subtrahend. 1's complement can be obtained by inverting all bits and adding 1 to the sum through the input carry. The addition and subtraction operations can be combined into one common circuit by including an exclusive OR gate with each full adder (Refer Figure 2.10).

### NOTES

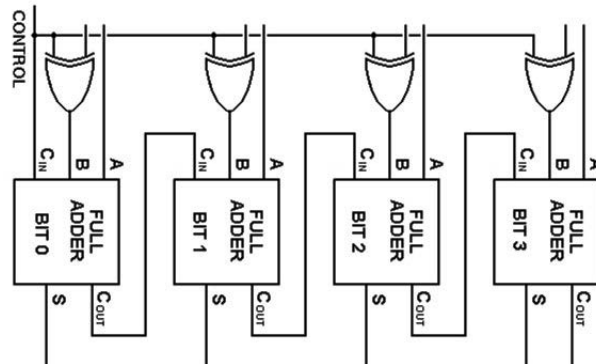


Fig. 2.10 Binary Adder Subtractor

As  $A \oplus 0$  is A and  
 $A \oplus 1$  is A'

So, if control = 0 the circuit acts as an adder circuit and for control = 1 it acts as a subtractor circuit.

### Binary Incrementer

The incrementer circuit adds 1 to a number in a register. This operation can be easily implemented with a binary counter. Every time the count enable is active, the clock pulse transition increments the content of the register by 1. Often, it is required to perform the increment micro-operation with a combinational circuit independent of a particular register. This can be accomplished by means of half-adders.

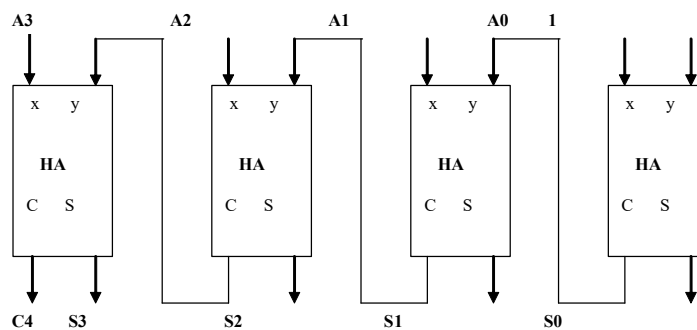


Fig. 2.11 Bit Binary Incrementer Using Half Adders

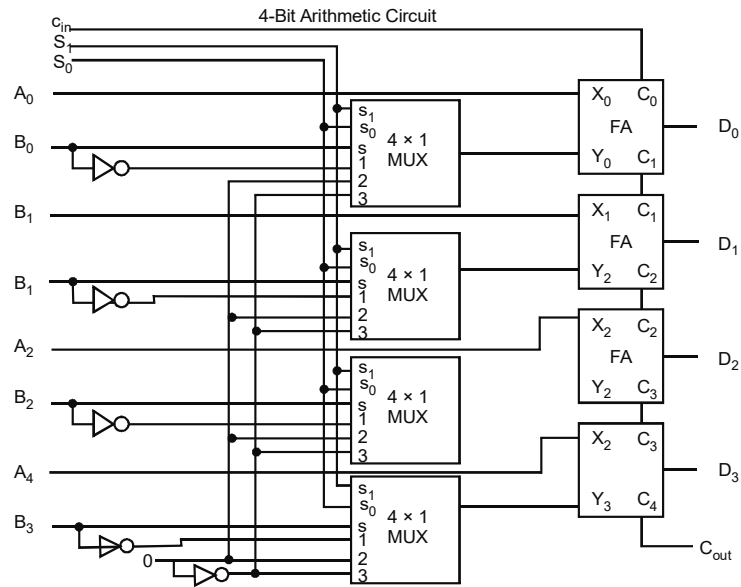
### Arithmetic Circuit

All possible operations related to binary addition and subtraction can be implemented in one circuit by controlling input to parallel adder circuit as shown in Figure 2.11. The arithmetic output of binary adder is calculated to the following expression:

$$\text{Sum} = A + B + C_{in}$$

**NOTES**

Where A and B are inputs and  $C_{in}$  is an input carry, which can have two possible values 0 or 1. We can obtain various possible combinations of output by taking various possible value of input B. In the circuit in Figure 2.12, four possible values of B namely B, B' (B complement), 0 and 1 have been taken. In order to decide what value of B should be input, use  $4 \times 1$  multiplexer, i.e., with two control lines, and an output of multiplexer is input of the binary adder. By controlling the value of B with two selection inputs  $S_0$  and  $S_1$  and two possible values of  $C_{in}$ , it is possible to generate the eight arithmetic micro-operations as given in Table 2.6.



**Fig. 2.12** A 4-bit Adder Circuit

When the values of  $S_1$  and  $S_0$  is 00, input B is applied to the input to adder and the output of the circuit will be  $A+B$  or  $A+B+1$  depending on whether  $C_{in}$  is 0 or 1. When  $S_1$  and  $S_0$  is 01, input B' is applied to input to adder and output of circuit will be  $A-B$  with borrow or  $A-B$  depending on whether  $C_{in}$  is 0 or 1. When  $S_1$  and  $S_0$  is 10 input 0 is applied to input to adder and output of circuit will be A or  $A+1$  depending on whether  $C_{in}$  is 0 or 1. When  $S_1$  and  $S_0$  is 11 as control signal 1 is applied to input to adder and output of circuit will be  $A-1$  or A depending on whether  $C_{in}$  is 0 or 1.

**Table 2.6** Arithmetic Circuit Function Table

Select			Input	Output	Micro-operation
$S_1$	$S_0$	$C_{in}$	Y	$D = A + Y + C_{in}$	
0	0	0	B	$D = A + B$	Add
0	0	1	$\overline{B}$	$D = A + \overline{B} + 1$	Add with carry
0	1	0	$\overline{B}$	$D = A + \overline{B}$	Subtract with borrow ( $A - B - 1$ )
0	1	1	$\overline{B}$	$D = A + \overline{B} + 1$	Subtract
1	0	0	0	$D = A$	Transfer A
1	0	1	0	$D = A + 1$	Increment A
1	1	0	1	$D = A - 1$	Decrement A
1	1	1	1	$D = A$	Transfer A

Ignore B and just give 0s

Ignore B and just give 1s

For example, when subtracting  $A(2n)$  from  $B(2n)$  use the circuit twice (for 2 instances hooked up)



## 2.4.2 Logic Micro-Operations

A logic circuit performs logical binary operations on strings of bits stored in registers. These operations consider each bit of the register separately and treat them as binary variables. Thus, they can manipulate individual bits or a portion of a word stored in a register. Logic micro-operations are very frequently used for making logical decisions, such as testing equality condition, and so on. They are used for changing some bit values, deleting a group of bits or inserting new bit values into the register. With two inputs sixteen different combinations of output are possible as given in Table 2.7. In general, to design any logic circuit we use the four basic logic gates, namely AND, OR, XOR and NOT. Although there are sixteen logic micro-operations, the other operations can be derived using these four logical operations. Table 2.8 shows how using these gates makes it possible to get, sixteen outputs with two binary variables. The complement micro-operation is the same as 1's complement and is denoted by a  $\overline{\text{AND}}$  on the top of the symbol or as  $A'$ , where A is the register name. The symbol is used to denote an OR micro-operation which will be used to denote AND micro-operation.  $\oplus$  is used to denote XOR operation.

### NOTES

Table 2.7 Truth Table for Sixteen Functions of Two Variables

Input A B	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
0 0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0 1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1 0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1 1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Table 2.8 Sixteen Outputs with two Variables

Boolean Function	Micro operation	Name
F0	$F \leftarrow 0$	CLEAR
F1	$F \leftarrow A \wedge B$	AND
F2	$F \leftarrow A \wedge B'$	
F3	$F \leftarrow A$	TRANSFER A
F4	$F \leftarrow A' \wedge B$	
F5	$F \leftarrow B$	TRANSFER B
F6	$F \leftarrow A \oplus B$	Exclusive - OR
F7	$F \leftarrow A \vee B$	OR
F8	$F \leftarrow A' \wedge B'$	NOR
F9	$F \leftarrow (A \oplus B)'$	Exclusive - NOR
F10	$F \leftarrow B'$	Complement B
F11	$F \leftarrow A \vee B'$	
F12	$F \leftarrow A'$	Complement A
F13	$F \leftarrow A' \vee B$	
F14	$F \leftarrow A' \vee B'$	NAND
F15	$F \leftarrow 1$	Set to all 1

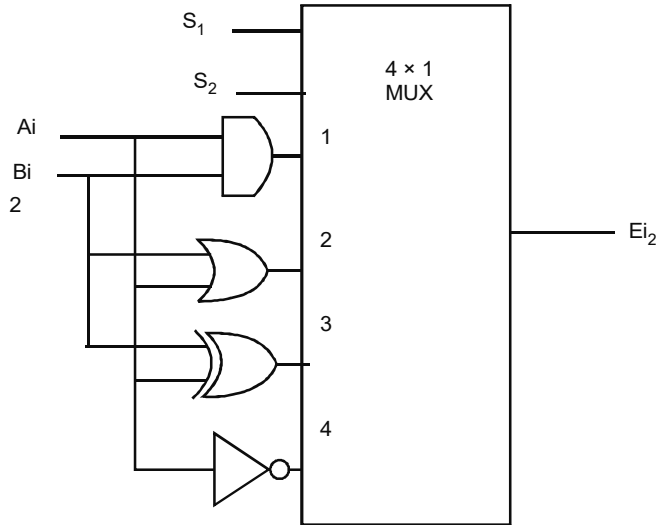
For example,  $P+Q: R4 \leftarrow R1 + R2, R3 \leftarrow R1 \leftarrow R3$

In the above expression, the variable on the left-hand side of the colon are control signals. Thus, + between P and Q is an OR operation between two binary variables of a control function. The right hand side of the colon is the mathematical expression and both logical and arithmetic expressions exist. In the above expression, the content of R1 and R2 is added and stored in R4 denoted by +

between R1 and R2, specifying an add micro-operation and the symbol between registers R1 and R3 show the OR operation between variable R1 and R3 and stored in R3.

**NOTES**

**Hardware Implementation**



*Fig. 2.13 One Stage Logic Circuit*

Figure 2.13 shows the hardware implementation of logic operations.

*Table 2.9 Function Table for Logic Micro-operation*

$S_1$	$S_0$	$E_i$	Operation
0	0	$E = A \wedge B$	AND
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	$E = \bar{A}$	Complement

Using combination of two micro-operations like AND and OR it is possible to have various operations like selective set (setting 1 to those bits in register A where there are corresponding 1's in B register). For example,

1010 A Before  
1100 B (Logic Operand)  
 1110 A After

Similarly, selective complement where those bits in A are complemented corresponding to 1's in B can be implemented with the XOR circuit. Other common operations are selective clear where those bits in A are set to 0 corresponding to 1's in B. This can be implemented with logic operation of AND with B'.

Logic operations allow us to manipulate individual bits which we could not do otherwise. Various logic applications are as follows:

- Selective set
- Selective complement
- Select clear

- Mask
- Insert
- Clear

You will now study these logic operations with examples:

### Selective-Set

‘Selective-Set’ sets to 1 the bits in register A where there is a corresponding 1 in register B. The bit corresponding to 0 in register B remains unchanged. For example:

1011 Content of A before  
1000 Content of B (logic operand)  
1011 Content of A after

This operation is done using the logical-OR operation.

### Selective-Complement

‘Selective-complement’ complements the bits in register A where there is a corresponding 1 in register B. The bit corresponding to 0 in register B remains unchanged. For example:

1011 Content of A before  
1000 Content of B (logic operand)  
0011 Content of A after

This is done using the exclusive-OR operation.

### Selective-Clear

‘Selective-clear’ clears to 0 the bits in register A where there is a corresponding 1 in register B. The bit corresponding to 0 in register B remains unchanged. For example:

1010 Content of A before  
1000 Content of B (logic operand)  
0010 Content of A after

This is done using the logical  $A \text{ AND } B$ .

### Mask

‘Mask’ clears to 0 the bits in register A where there is a corresponding 0 in register B. The bit corresponding to 1 in register B remains unchanged. For example:

1010 Content of A before  
1000 Content of B (logic operand)  
1000 Content of A after

This is done using the logical-AND operation and  $B$ .

### Insert

‘Insert’ inserts a new value into a set of bits in register A. It is used for introducing a specific bit pattern into the A register, leaving the other bit positions unchanged. This involves two steps which are as follows:

## NOTES

## NOTES

1. A mask operation to clear the desired bit positions
2. An OR operation to introduce the new bits into the desired positions

Consider an example. Suppose you want to introduce 1001 into the upper order four bits of A:

0110 1010(original)

1001 1010(desired)

First, we mask out the upper four bits (in our 8-bit value):

0110 1010 Content of A before

0000 1100 Content of B (logic operand)

0000 1010 Content of A after

- In the second step, we insert the new values:

0000 1010 Content of A before

1001 0000 Content of B (logic operand)

1001 1010 Content of A after

- The masking is done using an AND and the insertion is done with an OR.

### Clear

Clear compares A and B and produces all 0s. The bits in the same position in A and B are the same, otherwise they are set to 1. Thus, the bits in register A where there is a corresponding B are same are cleared 0 in register A. For example,

1010 Content of A before

1011 Content of B (logic operand)

0001  $A \leftarrow A \oplus B$

If A and B are both 1 or both 0, it produces 0. This is done using the logical-XOR operation and B.

### 2.4.3 Shift Micro-Operations

Shift micro-operations are used for serial transfer of data, i.e., shifting the content of the register either in the left or right direction. At the same time, when the first flip-flop receives the serial input, the other bits of register are shifted. During the left shift, the serial input is transferred to the rightmost position, i.e., to the Least Significant Number (LSB) and during right shift the serial input transfers the most significant bit, i.e., a bit entered from the leftmost position. There are three types of shift operations possible which are as follows:

- (i) Logical shifts transfer 0 through the serial input, with all the bits involved in the shifting.
- (ii) Arithmetic shifts multiply (or divide) a signed number by 2.
- (iii) Circular shifts circulate the bits of the register around the two ends with no loss of information.

Each of these shift operations can be in the left or right position. Thus in total, six types of shift micro-operations are possible. The control signal determines

the type of shift operation to be performed. Table 2.10 lists the notations used to represent the operation being used.

Table 2.10 Shift Micro-Operations

RTL	Description
$R \rightarrow \text{shl } R$	Logical left shift register R
$R \rightarrow \text{shr } R$	Logical right shift register R
$R \rightarrow \text{ashl } R$	Arithmetic left shift register R
$R \rightarrow \text{ashr } R$	Arithmetic right shift register R
$R \rightarrow \text{cir } R$	Circulate left register R
$R \rightarrow \text{cir } R$	Circulate right register R

**NOTES**

**Logical Shift**

A logical shift transfers 0 through the serial input. During logical shift, the bit transferred to the end bit is assumed to be 0 as shown in Figure 2.14.

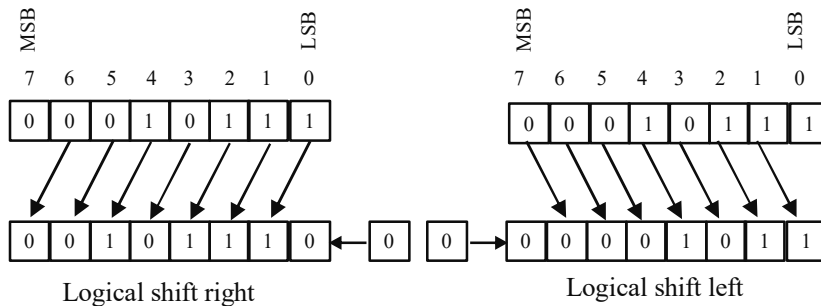


Fig. 2.14 8-bit Logical Shift Register

**Circular Shift (Rotate Operation)**

Another form of shift is the circular shift or bit rotation. Here the bits are ‘rotated’ as if the left and right ends of the register were joined. Here, the serial output act as serial input. This operation is useful if it is necessary to retain all the existing bits and just to shift these positions as frequently used in digital cryptography (Figure 2.15).

**Rotate Right**

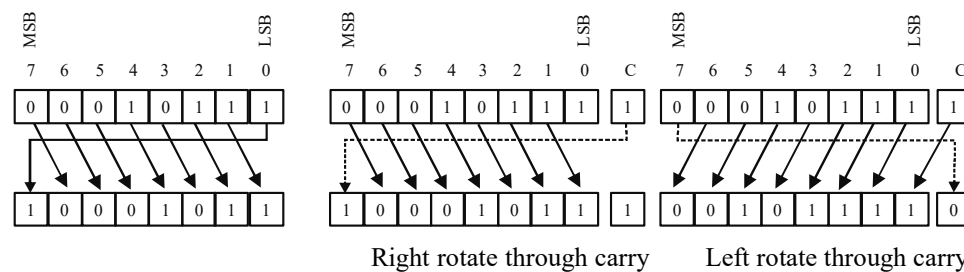


Fig. 2.15 Circular Right Shift with and Without Carry

If rotation takes place including the carry bit we call it rotation through carry. The rotation with carry is similar to the rotation with no carry operation

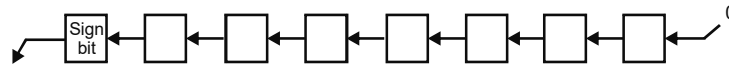
**NOTES**

except the fact that here carry bit is also considered as part register. The current value of the carry flag is accepted as input and new carry is generated. As shown in Figure 2.15, bit that is shifted out (on the other end) becomes the new value of the carry flag.

**Arithmetic Shift**

In an arithmetic shift, shift the signed binary number to the left or right. In the case of a left arithmetic shift, insert a 0 into the least significant bit and shift all other bits. An arithmetic left shift is used for multiplying a signed binary number by 2. The sign bit remains unchanged as the sign of a number does not change on multiplication by 2.

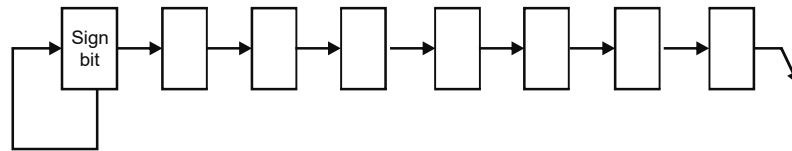
Figure 2.16 shows the arithmetic left shift.



**Fig. 2.16** Arithmetic Left Shift

An arithmetic right shift divides the number by 2. The arithmetic right shift leaves the sign bit unchanged. It is copied as such and 0-bit is inserted next to sign bit. Then shift the number to the right, the LSB position is lost.

Figure 2.17 shows the arithmetic right shift.

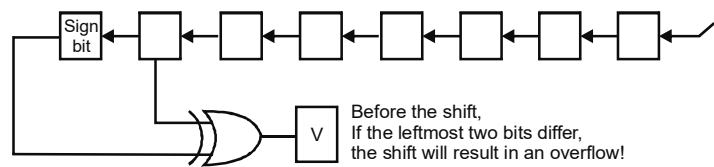


**Fig. 2.17** Arithmetic Right Shift

An overflow can occur after an arithmetic left shift of value  $R_{n-1}$  is not equal to  $R_{n-2}$  before the shift. This is called overflow because in such a case the result will not be the same as expected. An overflow flip-flop can be used to detect whether or not an arithmetic left shift overflow condition can arise.

$$V_S = R_{n-1} \text{ XOR } R_{n-2}$$

If  $V_S = 0$  there is no overflow, but if  $V_S = 1$ , there is an overflow and a sign reversal has taken place after the shift. Figure 2.18.



**Fig. 2.18** Overflow

The logical and arithmetic left-shifts are exactly the same operation. However, the logical right-shift inserts bits with value 0 instead of copies of the sign bit. Hence, the logical shift is suitable for unsigned binary numbers, while the arithmetic shift is suitable for signed 2's binary numbers.

**Shift Unit with a Bi-Directional Shift Register**

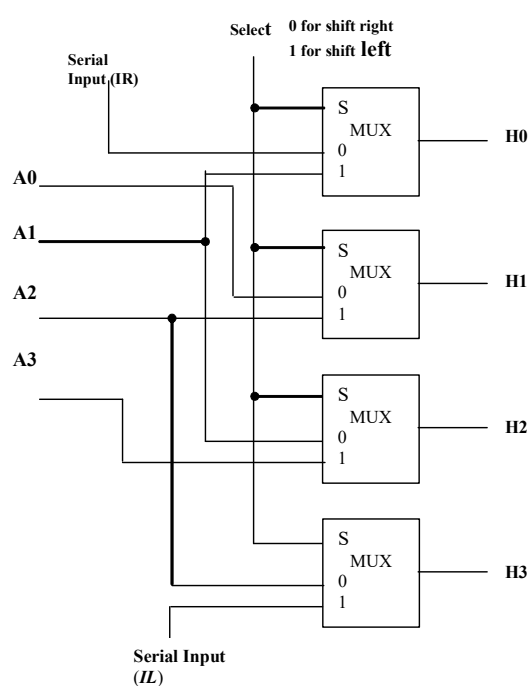
You can implement the shift micro-operation, either using a shift register or designing a circuit using multiplexers. In the first case, a bidirectional shift register with parallel

load is used. Data can be transferred to the register in parallel to all bits location and then shifted to the right or left according to the instruction. This operation takes two clock pulses to complete; the first for loading the data into the register and the other for initiating the shift.

### Shift Unit with Combinational Circuit (Shifter)

Another technique to implement the shift operation is by designing a circuit as shown in Figure 2.19 using multiplexer. Here, the register whose contents are to be shifted are first transferred onto a common bus. These buses are connected to the combinational shifter, and the shifted number is then loaded back into the register. This requires only one clock pulse for loading the shifted value into the register. As shown in Figure 2.19 the 4-bit shifter has four data inputs, A0 through A3, and four data outputs H0 through H3. There is one selection line S that determines left or right shift. When the selection input S=0, the input data is shifted right and if S=1, the input data is shifted left. There are two serial inputs that provide input in the left and right shifts respectively. To design  $n$  bit shifter, we need  $n$  multiplexers.

### NOTES



Select	Output			
S	H0	H1	H2	H3
0	IR	A0	A1	A2
1	A1	A2	A3	IL

Fig. 2.19 4-Bit Combinational Circuit Shifter

### Combined Arithmetic Logic Shift Unit

The arithmetic, logic and shift circuits can be combined into one ALU by using a multiplexer. The desired component can be selected using suitable selection variables as given in Table 2.11. The instruction is decoded and depending on the opcode, the control unit indicates which micro-operation is to be performed and the appropriate selection lines input is decided. Thus, according to the operation to be performed, the required component is selected by the selection line. The data to be operated, also called operands, act as inputs to the ALU, which is stored in registers. ALU performs the operation. The result is transferred to a destination register. The ALU needs one clock pulse to perform a complete operation. This includes register transfer operand from the source registers to the

**NOTES**

ALU, and result to the destination register. The ALU also takes inputs as a set of condition codes from the status register. It can also generate an output which are condition codes and are to be stored in status registers. These codes are used for test conditions like overflow, and divide-by-zero.

**Table 2.11** Function Table for Arithmetic Logic Shift Unit

S3	S2	S1	S0	Cin	Operation	Function
0	0	0	0	0	$F = A + B$	Addition
0	0	0	0	1	$F = A + B + 1$	Add with carry
0	0	0	1	0	$F = A + \bar{B}$	Subtract with borrow
0	0	0	1	1	$F = A + B + 1$	Subtraction
0	0	1	0	0	$F = A$	Transfer A
0	0	1	0	1	$F = A + 1$	Increment A
0	0	1	1	0	$F = A - 1$	Decrement A
0	0	1	1	1	$F = A$	Transfer A
0	1	0	0	x	$F = A \wedge B$	AND
0	1	0	1	x	$F = A \vee B$	OR
0	1	1	0	x	$F = A \oplus B$	XOR
0	1	1	1	x	$F = \bar{A}$	Complement A
1	0	x	x	x	$F = \text{shr } A$	Shift right A into F
1	1	x	x	x	$F = \text{shl } A$	Shift left A into F

**Check Your Progress**

1. What is temporary register used for?
2. Give some examples of dedicated address registers.
3. How is the speed of bus affected?
4. Write the function performed by multiplexer and decoder.
5. What are the types of buses?
6. List the four categories into which micro-operators in digital computers are classified.
7. What is a binary adder?
8. What is the function of a logic circuit?

**2.5 INSTRUCTION AND INSTRUCTION CODE**

An *instruction* is a command given to a computer to perform a specified operation on some given data. These instructions tell the CPU what to do. In other words, an instruction guides the CPU to perform work accordingly. The most common fields found in the instructions are the operation code and the operands. Each field specifies different information for the computer. The two important fields of an instruction are as follows:

- Op code
- Operand



Thus,

Instruction = op code + operand

Op code (Operation Code) is an instruction field that specifies the particular operation to be performed by the instruction. Each operation has its unique op code and may take several micro-operations to accomplish. MOV, ADD and SUB are examples of Intel 8086 op codes.

Operand fields specify where to get the source and destination operands for the operation specified by the op code. The source/destination of operands can be the memory or one of the general-purpose registers. The complete set of op codes for a particular microprocessor defines the instruction set for that processor.

*Instruction sequencing* is the method by which instructions are selected for execution, i.e. the manner in which control of the processor is transferred from one instruction to another.

The simplest method of controlling the sequence of instruction execution is to have each instruction explicitly specify the address of the next instruction to be run. However, explicit inclusion of instruction addresses in all the instructions is disadvantageous as the instruction length increases. This results in increase of cost of memory where the instructions are to be stored.

### 2.5.1 Instruction Execution

The sequence of operations performed by the CPU in processing an instruction is known as an instruction cycle. The time required to complete one instruction is called execution time.

To execute an instruction, the following three steps are required:

- *Fetch step*, during which a new instruction is read from the memory.
- *Decode step*, during which the instruction is decoded.
- *Execute step*, during which the operations specified by the instruction are executed.

The instruction fetch operation is initiated by loading the contents of the Program Counter (PC) into the Address Register (AR) and it sends a read request to the memory. The contents of the PC is the address of the instruction to be run. The instruction read from the memory is then placed in the Instruction Register (IR) and the content of the PC is incremented so that it contains the address of the next instruction in the program. After this, the instruction is decoded to determine the type of instruction that was just read. Finally, the instruction is executed to perform the operation specified by the instruction.

Let us consider an instruction, which adds the content of a memory location specified by register  $R_0$  to the content of register  $R_2$  and the result is to be stored in  $R_2$ . The execution of this instruction is performed in the following steps:

**Step 1** Fetch and decode the instruction

**Step 2** Fetch the operand

## NOTES

**Step 3** Perform the operation (addition)

**Step 4** Store the result in  $R_2$

**NOTES**

**Instruction Codes**

A code is a symbol or group of symbols that stands for something. It is a representation of discrete elements of information, which may be in the form of numbers, letters or any other varying physical quantities. Binary bits 1 and 0 are often used in groups. The codes are used to communicate the information to the digital computer and to retrieve from it. The purpose of the code is that the operator can feed data into computers directly in decimal numbers, alphabets and special characters. The computer in turn converts these data in binary code which it can process and after computation, it again converts the binary data into decimal numbers, alphabets and special characters which we could understand easily.

Certain binary codes are used for arithmetic operations. Other codes facilitate the creation of digital transducers for entering information into a system.

**2.5.2 Binary Coded Decimal (BCD) Code**

Code is a symbolic representation of discrete elements of information, which may be in the form of letters, numbers or any other varying physical quantities. The symbol used is a string of binary digits 0 and 1, and these are arranged according to the rules of code. The group of binary bits (0 and 1) is known as a *binary code*. A group of four binary bits is known as *nibble*, and a group of eight binary bits is known as *byte*. In computers, code provide a means of specifying the characters using only the 1 and 0 binary symbols available. Several binary codes are used to express decimal numbers, alphabets, special characters and for display. Digital systems, for their internal operations, use some form of binary numbers. However, the external world is decimal in nature. In many applications, special codes are used for such auxiliary functions as error detection and correction.

Binary Coded Decimal or BCD uses binary number system to specify the decimal numbers 0 to 9. It is composed of four bits. The weights are assigned according to the position occupied by these digits. The weight of the first (right most) position is  $2^0$  or 1, the second  $2^1$  or 2, the third  $2^2$  or 4 and the fourth  $2^3$  or 8. Reading from left to right the weighting is 8 - 4 - 2 - 1 and the code is also called the 8 - 4 - 2 - 1 code.

The numbers from 0 to 9 are represented as in binary but after 9, the representations are different. For example, the decimal number 12 in binary is  $[1100]_2$  but the same number in BCD is represented as  $[0001\ 0010]_{BCD}$ . Therefore, the six code combinations 1010, 1011, 1100, 1110 and 1111 are invalid in BCD code.

**Example 2.1:** Write BCD for a decimal number 559.

<b>Solution:</b> Decimal number	→	5	5	9
		↓	↓	↓
BCD code	→	0101	0101	1001

$$\therefore [559]_{10} = [0101 \ 0101 \ 1001]_{\text{BCD}}$$

**Example 2.2:** Write the BCD code equivalent for decimal number 96.42.

**Solution :** Decimal number  $\rightarrow$  9 6 . 4 2  
 $\downarrow \quad \downarrow \quad \downarrow \quad \downarrow$   
 BCD code  $\rightarrow$  1001 0110 . 0100 0010

$$\therefore [96.42]_{10} = [1001 \ 0110 \ . \ 0100 \ 0010]_{\text{BCD}}$$

## NOTES

### 2.5.3 Excess-3 Code

The Excess-3 is a digital code that is derived by adding to each decimal digit and then converting the result to four-bit binary. The Excess-3 code is used in some arithmetic circuits because it is self-complementing. Table 2.12 shows Excess-3 codes to represent single decimal digit and its BCD code.

Table 2.12 BCD and Excess-3 Codes

Decimal Digit	BCD				Excess-3 Code			
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0

### 2.5.4 Gray Code

The Gray code pertains to a class of codes called *minimum change codes*, in which only one bit in the code group changes when going from one step to the next. Gray code is not an arithmetic code.

Table 2.13 Gray Code

Decimal Digit	Binary Code				Gray Code			
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	0	0	0	1	1
3	0	0	1	1	0	0	1	0
4	0	1	0	0	0	1	1	0
5	0	1	0	1	0	1	1	1
6	0	1	1	0	0	1	0	1
7	0	1	1	1	0	1	0	0
8	1	0	0	0	1	1	0	0
9	1	0	0	1	1	1	0	1

**NOTES**

10	1	0	1	0	1	1	1	1	1 ←
11	1	0	1	1	1	1	1	1	0 ←
12	1	1	0	0	1	0	1	0	0 ←
13	1	1	0	1	1	0	1	1	1 ←
14	1	1	1	0	1	0	0	0	1 ←
15	1	1	1	1	1	0	0	0	1 ←

Table 2.13 shows the Gray code representation for the decimal numbers 0 to 9 together with the straight binary code.

**2.2.5 Alphanumeric Codes**

Alphanumeric codes are the codes that represent alphabetic characters (letters), punctuation marks and other special characters. Alphanumeric code represents all of the various characters and functions that are found on a computer keyboard.

**The ASCII Code**

The abbreviation ASCII stands for the *American Standard Code for Information Interchange*. The ASCII code is a 7-bit code used in transferring coded information from keyboards and to computer displays and printers. It is used to represent numbers, letters, punctuation marks as well as control characters. For example, the letter *A* is represented by 100 0001.

**EBCDIC Code**

The abbreviation EBCDIC stands for the *Extended Binary Coded Decimal Interchange Code*. It is an 8-bit code in which the decimal digits are represented by the 8421 BCD code preceded by 1111.

**2.5.6 Error-Detecting Codes**

A code that uses  $n$ -bit strings need not contain  $2^n$  valid code words. An error-detecting code has the property that corrupting or garbling a code word will likely produce a bit string that is not a code word (a non-code word).

A system that uses an error-detecting code generates, transmits, and stores only code words. Thus, errors in a bit string can be detected by a simple rule—if the bit string is a code word, it is assumed to be correct; if it is a non-code word, it contains an error.

**Parity**

The most simple and commonly used error detecting method is the *parity check method*, in which an extra bit called *parity bit* is included with the binary message, to make the total number of 1s either odd or even, resulting in two methods; (i) Even-parity method and (ii) Odd-parity method.

The ability of a code to detect single errors can be stated in terms of the *concept of distance*. A code detects all single errors if the minimum distance between all possible pairs of code words is 2.

In general,  $(n + 1)$  bits are needed to construct a single-error detecting code with  $2^n$  code words. The first  $n$  bits of a code word, called *information bits*, may be any of the  $2^n$   $n$ -bit strings minimum error bit.

A code in which the total number of 1s in a valid  $(n + 1)$  bit code word is even; this is called an *even-parity code*.

A code in which the total number of 1s in a valid  $(n + 1)$  bit code word is odd and this code is called an *odd-parity code*. These codes are also sometimes called *1-bit parity codes*, since they each use a single parity bit. The parity bit can be placed at either end of the code word, such that the receiver must be able to understand the parity bit and the actual data.

An  $n$ -bit code and its error-detecting properties under the independent error model are easily explained in terms of an  $n$ -cube. A code is simply a subset of the vertices of the  $n$ -cube. In order for the code to detect all single errors, no code-word vertex can be immediately adjacent to another code-word vertex.

Figure 2.20(a) shows a 3-bit code with five code words. Code word 111 is immediately adjacent to code words 110, 011 and 101. Since a single failure could change 111 to 110, 011 or 101 bits code does not detect all single errors. If we make 111 a non-code word, we obtain a code that does have the single-error-detecting property, as shown in Figure 2.20. No single error can change one code word into another.

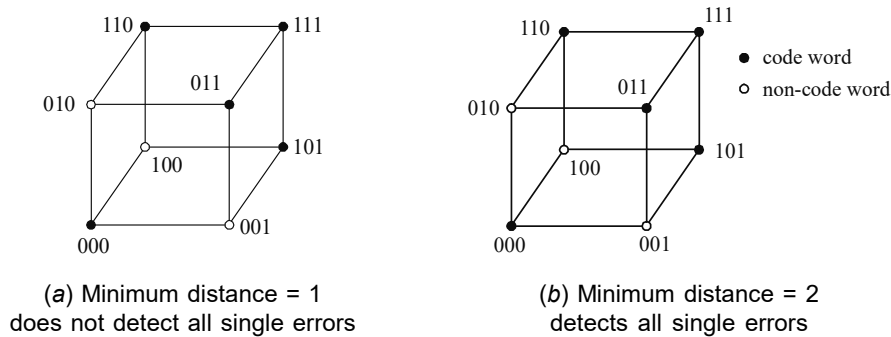


Figure 2.20 Code Words in Two Different 3-Bit Codes

Table 2.14 Distance-2 Codes with Three Information Bits

Information Bits XYZ	Even-parity Code		Odd-parity Code	
	XYZ	P	XYZ	P
000	000	1	000	1
001	001	1	001	0
010	010	1	010	0
011	011	0	011	1
100	100	1	100	0
101	101	0	101	1
110	110	0	110	1
111	111	1	111	0

Table 2.14 shows the distance codes with three information bits. The 1-bit parity codes do not detect 2-bit errors, since changing two bits does not affect the parity. However, the codes can detect errors in any *odd* number of bits. Actually, 1-bit parity codes error-detection capability stops after 1-bit errors. Other codes, with minimum distance greater than 2, can be used to detect multiple errors.

## NOTES

NOTES

**Checksum**

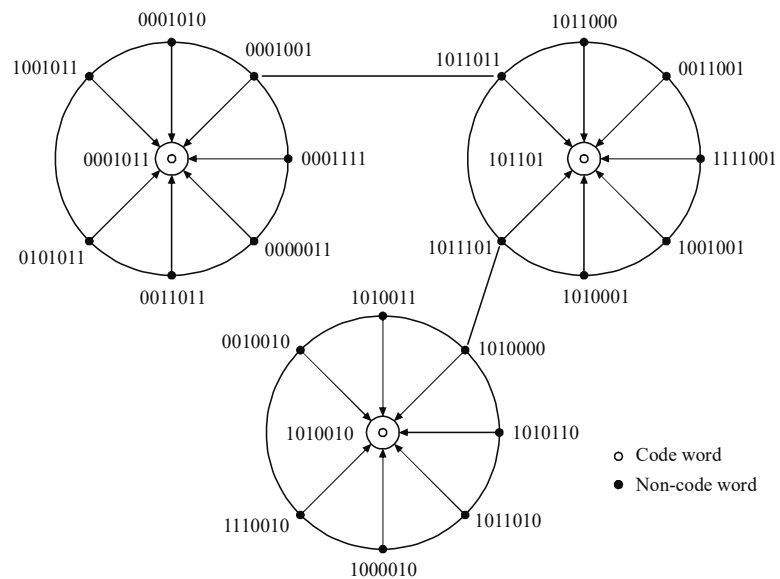
Since the double error will not change the parity of the bits, the parity checker will not indicate any error. The check sum method is used to detect double errors and pinpoint erroneous bits. The working of checksum method is explained as follows:

Initially word A 10110111 is transmitted. Next word B 00100010 is transmitted. Binary digits of the two letters would be added together and the total which is formed would be reserves in transmitter. Later, a letter C is sent and it is added in the earlier total and after that the new total is kept. Hence, every letter is adjoined to earlier total and subsequent passage of complete letters, complete total is called checksum which is sent out. Identical process is also acted out autonomously on beneficiary and the complete total attained is cross-checked with conveyed checksum. If the two totals are equivalent in that case there is no mistake.

**2.5.7 Error-Correcting Codes**

A code that is used to correct errors is called an *error-correcting code*. In general, if a code has minimum distance  $2c+1$ , it can be used to correct errors, that affect up to  $c$  bits. If a code's minimum distance is  $2c + d + 1$ , it can be used to correct errors in up to  $c$  bits and to detect in up to  $d$  additional bits.

Consider a fragment—cube for a code with minimum of 3. There are at least two non-code words between each pair of code words. Now, let us transmit code words and assume that failures affect at most one bit of each received code word. Then a received non-code word with a 1-bit error will be closer to the originally transmitted code word than to any other code word. Therefore, when we receive a non-code word, we can correct the error by changing the received non-code word to the nearest code word, as indicated by the arrows in the Figure 2.21 Deciding which code word was originally transmitted to produce a received word is called *decoding*, and the hardware that does this is an *error-correcting decoder*.

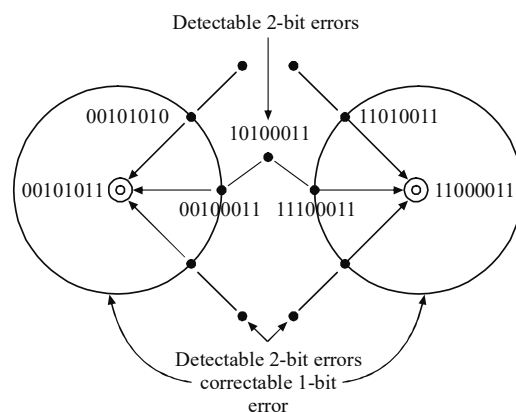


**Fig. 2.21** Some Code Words and Non-Code Words in a 7-Bit Distance-3 Code

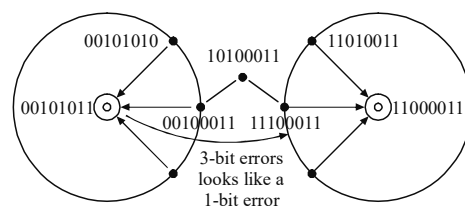
NOTES

For example, consider a fragment of the  $n$ -cube for a code with minimum distance 4 ( $c = 1, d = 1$ ) [Figure 2.22(a)]. Single-bit errors that produce non-code words 00101010 and 11010011 can be corrected. However, an error that produces 10100011 cannot be corrected, because no single-bit error can produce this non-code word, and either of two 2-bit errors could have produced it. So the code can detect a 2-bit error, but it cannot correct it.

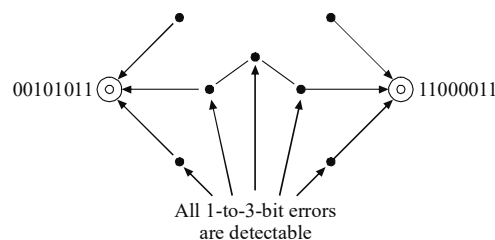
When a non-code word is received, we do not know which code word was originally transmitted; we only know which code word is closest to what we have received. Thus, a 3-bit error may be corrected to the wrong value as shown in Figure 2.22(b). The possibility of making this kind of mistake may be acceptable if 3-bit errors are very unlikely to occur. On the other hand, if we are concerned about 3-bit errors, we can change the decoding policy for the code. Instead of connecting the errors, we can just flag all non-code words as uncorrectable errors. Thus, we can use the same distance 4-code to detect up to 3-bit errors but correct no errors ( $c = 0, d = 3$ ) as shown in Figure 2.22(c).



(a) Correcting 1-Bit and Detecting 2-Bit Errors



(b) Incorrectly Correcting a 3-Bit Errors



(c) Correcting No Errors but Detecting upto 3-Bit Errors

Fig. 2.22 Some Code and Non-code Words in an 3-Bit Distance 4-Code

NOTES

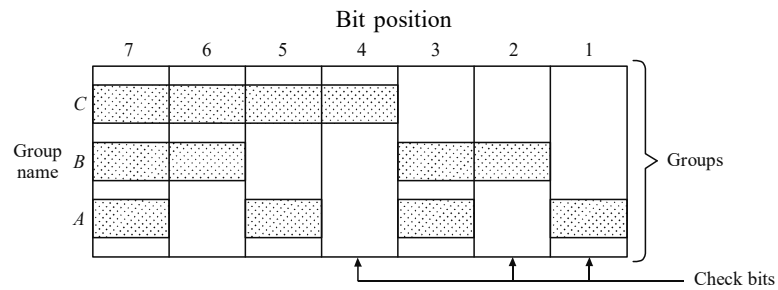
### 2.5.8 Hamming Codes

In 1950, R.W. Hamming developed a system that provides an orderly way to add one or more parity bits to a data character to allow detection or both error detection and correction. The *Hamming distance* between two code words is defined as the number of bits that must be changed for one code word to another. It is actually a method for constructing codes with a minimum distance of 3.

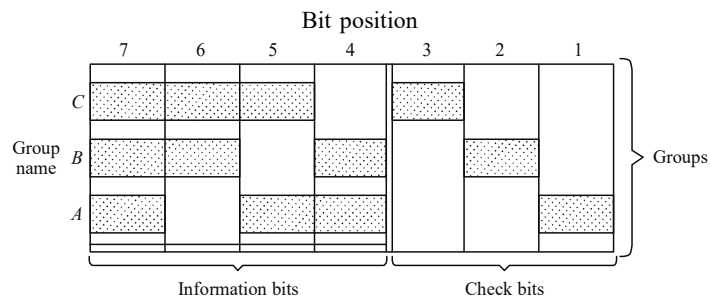
For any value of  $i$ , Hamming code method yields a  $(2^i - 1)$ —bit code with  $i$  check bits and  $2^i - 1 - i$  information bits. Distance-3 codes with a smaller number of information bits are obtained by deleting information bits from a Hamming code with a larger number of bits.

The bit positions in a Hamming code word can be numbered from 1 through  $2^i - 1$ . In this case, any position whose number is a power of 2 is a check bit, and the remaining positions are information bits. Each check bit is grouped with a subset of the information bits, as specified by a *parity-check matrix* as shown in Figure 2.23. Each check bit is grouped with the information positions whose numbers have a 1 in the same bit when expressed in binary. For example, check bit 2 (010) is grouped with information bits 3(011), 6(110) and 7(111). For a given combination of information bit values, each check bit is chosen to produce even parity, that is, so the total number of 1s in its group is even.

The bit positions of a parity-check matrix and the resulting code words are rearranged so that all of the check bits are on the right, as in Figure 2.23(b). The first two columns of Table 2.15 list the resulting code words.



(a) Hamming Codes with Bit Positions in Numerical Order



(b) Hamming Codes with Check Bits and Information Bits Separated

Fig. 2.23 Parity-Check Matrices for 7-bit Hamming Codes



We can prove that the minimum distance of a Hamming code is 3 by proving that at least a 3-bit change must be made to code a word to obtain another code word. We can also prove that a 1-bit or 2-bit change in a code word yields a non-code word.

If we change 1-bit of a code word in position  $j$ , then we change the parity of every group that contains position  $j$ . Since every information bit is contained in at least one group, at least one group has incorrect parity, and the result is a non-code word.

If we change two bits in positions  $j$  and  $k$ , parity groups that contain both positions  $j$  and  $k$  will still have correct parity. This is because parity is not affected when an even number of bits are changed. However, since  $j$  and  $k$  are different, their binary representations differ in at least one bit, corresponding to one of the parity groups. This group has only one bit changed, resulting in incorrect parity and a non-code word.

For the proof of 1-bit errors, the position numbers must be non-zero. For the proof of 2-bit errors, no two positions have the same number. Thus, with an  $i$ -bit position number, we can construct a Hamming code with up to  $2^i - 1$  bit positions.

The proof also suggests how can an error-correcting decoder be designed for a received Hamming code word. First, check all of the parity groups. If all the groups have even parity, then the received word is assumed to be correct. If one or more groups have odd parity, then a single error is assumed to have occurred. The pattern of groups that have odd parity called the *syndrome* must match one of the columns in the parity-check matrix; the corresponding bit position is assumed to contain the wrong value and is complemented. For example, using the code defined by Figure 2.23(b), suppose we receive the word 0101011. Groups  $B$  and  $C$  have odd parity, corresponding to position 6 of the parity-check matrix (the syndrome is  $110_2$ , or  $6_{10}$ ). By balancing a bit in place 6 of established sound, we decide that the accurate remark is 0001011.

A distance-3 Hamming code could with no trouble be customized for augmenting its smallest expense to 4. Adding a check bit more, selected in a way so as to communication of all matters, is constant. In the way the 1-bit even-parity code, this bit makes sure that every inaccuracy that affects an unusual figure of bits is visible.

Generally for detecting and correcting the mistakes in computer memory systems distance-3 and distance-4 Hamming codes are used, particularly in huge mainframe computers in which memory circuits are responsible for lot of system breakdown.

## NOTES

**Table 2.15** Code Words in Distance-3 and Distance-4 Hamming Codes with Four Information Bits

**NOTES**

Minimum Distance-3 Code		Minimum Distance-4 Code	
Information Bits	Parity Bits	Information Bits	Parity Bits
0000	000	0000	0000
0001	011	0001	0111
0010	101	0010	1011
0011	110	0011	1100
0100	110	0100	1101
0101	101	0101	1010
0110	011	0110	0110
0111	000	0111	0001
1000	111	1000	1110
1001	100	1001	1001
1010	010	1010	0101
1011	001	1011	0010
1100	001	1100	0011
1101	010	1101	0100
1110	100	1110	1000
1111	111	1111	1111

**Example 2.3:** Encode data bits 0101 as a seven-bit even-parity Hamming code.

**Solution:**

$D_7$	$D_6$	$D_5$	$P_4$	$D_3$	$P_2$	$P_1$
0	1	0	1	1	0	1

**Example 2.4:** A seven-bit Hamming code is received as 1111101. What is the correct code?

**Solution:**

$D_7$	$D_6$	$D_5$	$P_4$	$D_3$	$P_2$	$P_1$	
1	1	1	1	1	0	1	
					0	1	0

Bits 4, 5, 6 and 7, no error

Bits 2, 3, 6 and 7, error

Bits 1, 3, 5 and 7, no error

Bit 2 is in error, and the correct code is 1111111.

**Example 2.5:** For a received data 1100010, determine whether a single error occurred and, if so, correct the error.

**Solution:** Checking the three parity bit, groups for even-parity, we have:

$P_1$	$P_2$	8	$P_4$	4	2	1
1	1	0	0	0	1	0

$$P_1 + 8 + 4 + 1 = 1 + 0 + 0 + 0 = F$$

$$P_2 + 8 + 2 + 1 = 1 + 0 + 1 + 0 = E \text{ (Even-parity)}$$

$$P_4 + 4 + 2 + 1 = 0 + 0 + 1 + 0 = F \text{ (Failed even parity check)}$$

Any even parity failure indicates an error has occurred, bit 5 was in error. Thus, the correct digit is 6.

1	2	3	4	5	6	7
1	1	0	0	0	1	0
				↓		
1	1	0	0	1	1	0 (= digit 6)

## NOTES

**Example 2.6:** Determine the single error-correcting code for the BCD number 1001 (information bits) using even parity.

**Solution:** First, find the number of parity bits required. Let  $P = 3$ .

Then  $2^P = 2^3 = 8$

Since  $2^P \geq m + p + 1$ , we have  $m + p + 1 = 4 + 3 + 1 = 8$ .

Three parity bits are sufficient.

Total code bits =  $4 + 3 = 7$

Construct a bit position table.

Bit Designation	$P_1$	$P_2$	$M_1$	$P_3$	$M_2$	$M_3$	$M_4$
Bit position	1	2	3	4	5	6	7
Binary position number	001	010	011	100	101	110	111
Information bits			1		0	0	1
Parity bits	0	0		1			

Parity bits are determined in the following steps:

$P_1$  checks bit positions 1, 3, 5 and 7 and must be a 0 in order to have an even number of 1s (2) in this group.

$P_2$  checks bit positions 2, 3, 6 and 7 and must be a 0 in order to have an even number of 1s (2) in this group.

$P_3$  checks bit positions 4, 5, 6 and 7 and must be a 1 in order to have an even number of 1s (2) in this group.

These parity bits are entered into the table, and the resulting code is 0011001.

**Example 2.7:** Determine the single error-correcting code for the information code 10110 for odd-parity.

**Solution:** Determine the number of parity bits required. In this case the number of information bits,  $m$ , is five.

Let  $p = 4$ ,  $2^p = 2^4 = 16$

We know that  $m + p + 1 = 5 + 4 + 1 = 10$ .

Four parity bits are sufficient.

Total code bits =  $5 + 4 = 9$

Construct a bit position table.

**NOTES**

Bit Designation	$P_1$	$P_2$	$M_1$	$P_3$	$M_2$	$M_3$	$M_4$	$P_4$	$M_5$
Bit position	1	2	3	4	5	6	7	8	9
Binary position number	0001	0010	0011	0100	0101	0110	0111	1000	1001
Information bits			1		0	1	1		0
Parity bits	1	0		1				1	

Parity bits are determined as follows:

$P_1$  checks bit positions 1, 3, 5, 7 and 9 and must be 1 to have an odd number of 1s (3) in this group.

$P_2$  checks bit positions 2, 3, 6 and 7 and must be 0 to have an odd number of 1s (3) in this group.

$P_3$  checks bit positions 4, 5, 6 and 7 and must be 1 to have an odd number of 1s (3) in this group.

$P_4$  checks bit positions 8 and 9 and must be a 1 to have an odd number of 1s (1) in this group.

These parity bits are entered into the table, and the resulting combined code is 101101110.

**Example 2.8:** The code word 0011001 is transmitted and 0010001 is received. The receiver does not know what was transmitted and must look for proper parities to determine if the code is correct. Designate any error that has occurred in transmission if even-parity is used.

**Solution:** First, prepare a bit position table:

Bit designation	$P_1$	$P_2$	$M_1$	$P_3$	$M_2$	$M_3$	$M_4$
Bit position	1	2	3	4	5	6	7
Binary position number	001	010	011	100	101	110	111
Received code	0	0	1	0	0	0	1

**First Parity Check:**  $P_1$  checks positions 1, 3, 5 and 7

There are two 1s in this group.

Parity check is good  $\longrightarrow$  0 (LSB)

**Second Parity Check:**  $P_2$  checks positions 2, 3, 6 and 7.

There are two 1s in this group.

Parity check is good  $\longrightarrow$  0

**Third Parity Check:**  $P_3$  checks positions 4, 5, 6 and 7.

There is one in this group.

Parity check is bad  $\longrightarrow$  1 (MSB)

**Result:** The error position code is 100 (binary 4). This says that the bit in the number 4 position is in error. It is a 0 and should be a 1. The correct code is 0011001, which agrees with the transmitted code.

**Example 2.9:** The code 101101010 is received. Correct any errors. There are four parity bits and odd-parity is used.

## 2.6 COMPUTER INSTRUCTION

The primary function of the processing unit in the computer is to interpret the instructions given in a program and carry out the instructions. Processors are designed to interpret a specified number of instruction codes. Each instruction code is a string of binary digits. All processors have input/output instructions, arithmetic instructions, logic instructions, branch instructions and instructions to manipulate characters. The number and type of instructions differ from processor to processor. The list of specific instructions supported by the CPU is termed as its *Instruction set*. An instruction in the computer should specify the following:

- The task or operation to be carried out by the processor. This is termed as the *opcode*.
- The address(es) in memory of the operand(s) on which the data processing is to be performed.
- The address in the memory that may store the results of the data-processing operation performed by the instruction.
- The address in the memory for the next instruction, to be fetched and executed. The next instruction which is executed is normally the next instruction following the current instruction in the memory. Therefore, no explicit reference to the next instruction is provided.

### NOTES

### 2.6.1 Instruction Representation

An instruction is divided into a number of fields and is represented as a sequence of bits. Each of the fields constitutes an element of the instruction. A layout of an instruction is termed as the instruction format.

In most instruction sets, many instruction formats are used, (Refer Figure 2.24). An instruction is first read into an Instruction Register (IR), then the CPU, which extracts and processes the required operands on the basis of references made on the instruction fields, and then decodes it. Since the binary representation of the instruction is difficult to comprehend, it is seldom used for representation. Instead, a symbolic representation is used.

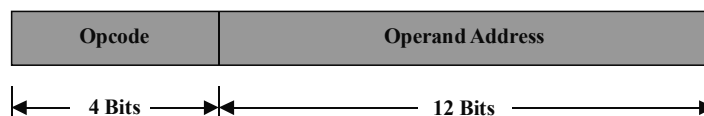


Fig. 2.24 A Sample Instruction Format

Table 2.16 Examples of Typical Instructions

Instruction	Interpretation	Number of Addresses
ADD A,B,C	Operation $A = B + C$ is executed	3
ADD A,B	$A = A + B$ . In this case the original content of operand location is lost	2
ADD A	$AC = AC + A$ . Here A is added to the accumulator	1

Typically, CPUs manufactured by different manufacturers have different instruction sets. This is why machine language programs developed for a particular

CPU do not run on a computer with a different CPU (having a different instruction set). An example of typical instruction is given in Table 2.16.

NOTES

2.7 TIMING AND CONTROLS

Timing and control unit generates timing and control signals. It is necessary for the execution of instructions which provides status, timing and control signals. This unit is necessary for the other parts of the CPU (Central Processing Unit). It acts as the brain of the computer which controls other peripherals and interfaces. It consists of Program Counter (PC) which is used for addressing the program. It contains the eight-level hardware stack for PC storage during subroutine calls and input/output interrupt services.

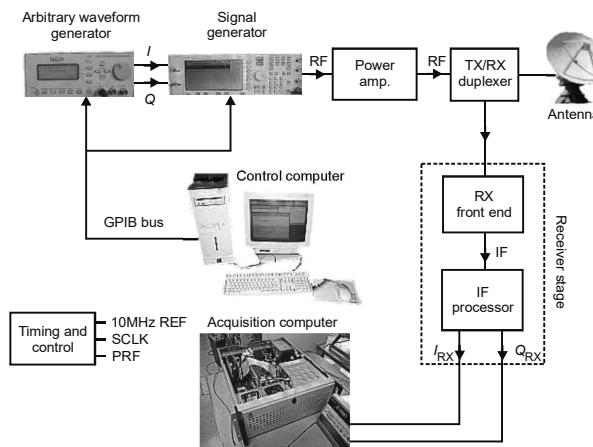


Fig. 2.25 Air-Radar-Attached Timing and Control Unit

The above Figure 2.25 shows how the clock pulses and control signals are collectively generated by both the units for the required operation of the radar system, radar sample clock and the pulse repetition frequency.

The memory control unit works as an interface between the processor and all the memories on-chip or off-chip. Timing is based on the system clock which is either an on-board oscillator or an external clock. In either case, the maximum clock frequency is 50 MHz (Megahertz) when using 32-bit TSR (terminate and stay resident) and 44 MHz when using 64-bit TSR

ALU	Accumulator
	General and special purpose registers
Timing and control unit	

Fig. 2.26 Schematic Diagram of a Control Unit

Figure 2.26 has the schematic diagram of a control unit. The following steps are performed by control unit to fetch and execute the instructions:

- It reads the address of memory location where it lies.
- It reads the instruction from the memory.
- It sends instructions to decoding circuit for decoding.

- It addresses the data which is required for executing and reading from the memory.
- It then sends the result to the memory or keeps it in same register to await its chance in queue.
- It takes help from program counter to fetch next instruction.

## NOTES

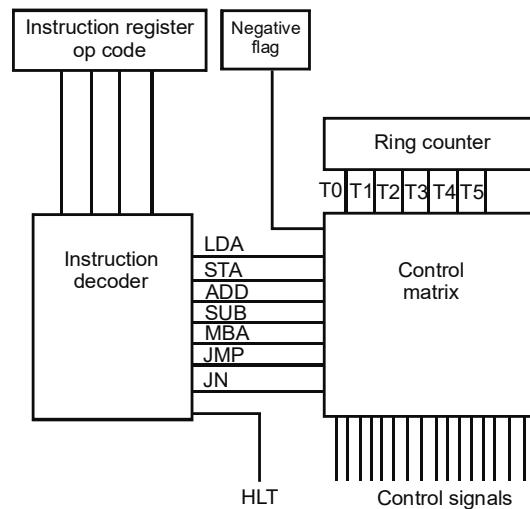


Fig. 2.27 Block Diagram of Control Unit

Figure 2.27 shows the block diagram of the control unit.

### 2.7.1 Functions of Control Unit

The main function of control unit can be see in Figure 2.28.

- Control section or unit controls the entire operation of the computer. It also controls all other devices connected to the CPU. First, it fetches instructions from the memory and then decodes the instruction. After interpreting the instructions, it knows what tasks are to be performed. The last step is to send suitable control signals to other components.
- It executes further necessary steps to run instructions successfully.
- It maintains the set of instructions and directs the operation of entire system.
- It controls the data flow between CPU and main memory.
- Control unit fetches the instructions from the memory one after another for execution unit where all the instructions are run and executed.

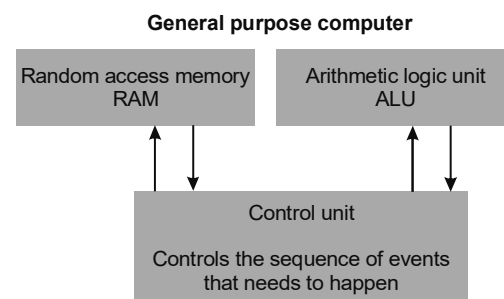


Fig. 2.28 Main Function of Control Unit

Source: [www.teach-ict.com](http://www.teach-ict.com)

## NOTES

### 2.7.2 Instruction Cycle

When a CPU is given an instruction in machine language, this instruction is fetched from the memory by the CPU to execute. The instruction cycle (or fetch-and-execute cycle) refers to the time period, during which one instruction is fetched and executed by the CPU. An instruction cycle has four stages:

1. **Fetch:** In this step, an instruction is loaded from the memory onto the CPU registers. All the instructions must be fetched before they can be executed.
2. **Decode:** In this step, the control unit decodes the instructions.
3. **Derive Effective Address of the Instruction:** In this step, if the instruction has an indirect address then the effective address of the instruction from memory is read.
4. **Execute:** In this step, the action represented by instruction is performed.

Steps 1 and 2, taken together, are called fetch cycle and these steps are the same for each instruction. Steps 3 and 4 are called execute cycle and these steps change with each instruction.

---

## 2.8 MEMORY REFERENCE INSTRUCTIONS

---

The Memory Reference Instructions (MRI) are 32-bits long, with extra 16-bits. It comes from the next successive memory allocation which follows the instruction itself. The effective memory address is addressed by sign-extending the 16-bit displacement to 32-bits. Then it adds to the given index register as follows:

$$ea = r[x] + sxt(displ)$$

Here 'ea' is a variable which contains  $r[x]$ . It refers to the program counter which is indexed.  $r[0]$  index shows the relative address which follows immediate instructions. This allows easy reference to locate the current program text. All memory reference instructions share the assembly language formats as follows:

op code	Keywords
op	Rsrc, Rx, disp, dst
op	Rsrc, label

The first row shows the op code such as Rx, which is one of R1 through R15 and the second row is used for system addressing. The assembler automatically computes disp which is difference between the current location and addressed label.

Memory reference instructions are those instructions in which two machine cycles are required. One cycle fetches the instructions and other fetches the data and executes the instructions. Instructions are based on arithmetic calculations.

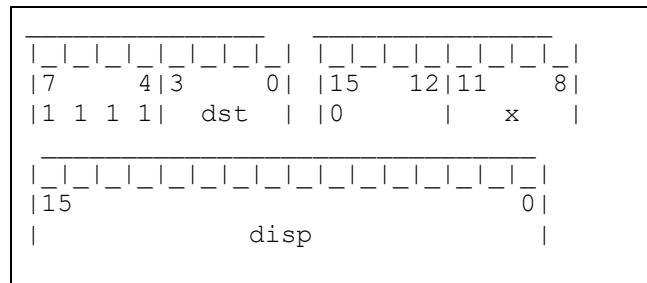


**NOTES**

Memory reference instructions are used in multithreaded parallel processor architecture. These instructions fetch process that two consecutive instructions are tested to determine if both are register load instructions or register save instructions. If both instructions are register save/load instructions then corresponding addresses are tested.

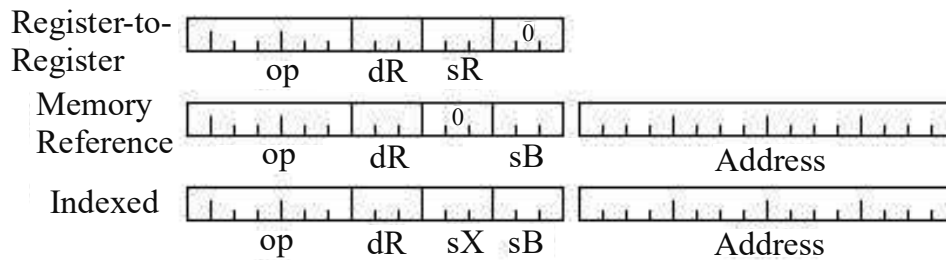
**2.8.1 Memory Reference Format**

Memory reference instructions are arranged as per the protocols of memory reference format of the input file in a simple ASCII sequence of integers between the range 0 to 99 separated by spaces without formatted text and symbols. These are pure sequences of space separated integer numbers. For example, |7 4|.



*Fig. 2.29 Memory Reference Format*

Figure 2.29 shows how 7 4 15 12 ... are arranged in memory reference format. Here dst and disp are keywords, where dst represents destination address and disp refers to displayed memory space.



*Fig. 2.30 Memory Reference Instructions*

Figure 2.30 shows the mode of operation of the computer. The dR and sR fields give the destination register and source register for an operation. It contains any value between 0 and 7. The sB field indicates the base/address register and contains the value from 1 to 7. The sX field indicates the arithmetic/index register and contains the value from 1 to 7. The first two bits of seven op code are 00, 01 or 10. Instructions starting with 11 are used for other instructions. The op code has two parts in which first part indicates the type of number and the second part shows the operation performed according to instruction (Refer Figures 2.25 and 2.26).

NOTES

Table 2.17 First Part of Op Code

Binary Representation	Type of Number	Bit Representation
000	Byte	8-bit integer
001	Halfword	16-bit integer
010	Integer	32-bit integer
011	Long	64-bit integer
1000	Medium	48-floating point
1001	Floating	32-floating point
1010	Double	64-floating point
1011	Quad	128-floating point

Table 2.18 Second Part of Op Code

Binary Representation		Type of Number
0000	000	Swap
0001	001	Compare
0010	010	Load
0011	011	Store
0100	100	Add
0101	101	Subtract
0110	110	Multiply
0111	111	Divide
1000		Insert
1001		Unsigned Compare
1010		Unsigned Load
1011		XOR
1100		AND
1101		OR
1110		Multiply Extensively
1111		Divide Extensively

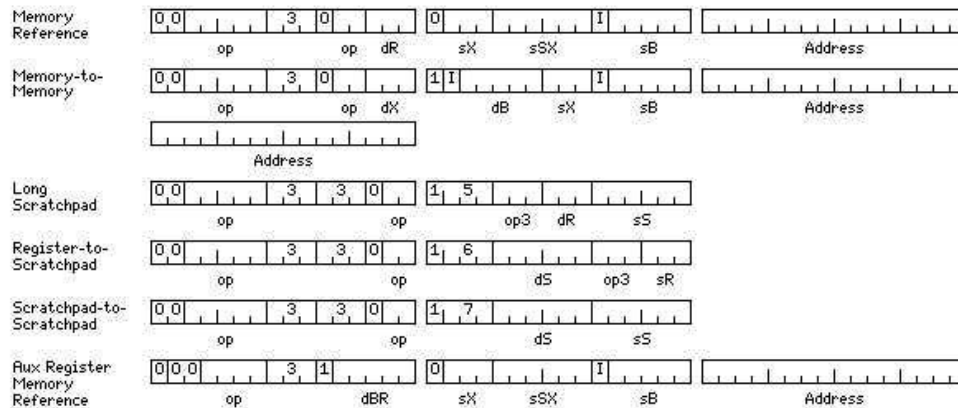


Fig. 2.31 Memory Reference Instructions from Register-to-Register Format

In Figure 2.31, memory reference shows the memory location, memory to memory shows source and destination operands, large scratchpad is the scalar instruction format with sixty-four supplementary registers for source and one general register for destination, whereas Aux register memory reference are scalar instructions. XOR, AND and OR perform the basic logical operations.

**Table 2.19** Logical Operations of XOR, AND and OR

<i>a</i>	<i>b</i>	<i>a XOR b</i>	<i>a AND b</i>	<i>a OR b</i>
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	0	1	1

## NOTES

## 2.9 INPUT/OUTPUT AND INTERRUPTS

Input/output interrupt is an external hardware event which causes the CPU to interrupt the current instruction sequence. It follows an interrupt mechanism to call the special Interrupt Service Routine (ISR). Input/output interrupt services save all the registers and flags. They also restore the registers and flags then resume execution of code they interrupted. Interrupt is essentially a procedure. They can pause the execution of some program at any point between two instructions when an interrupt occurs. If an interrupt occurs in the middle of the execution of some instruction, the CPU follows that instruction before transferring control to the interrupt service routine.

For example, an interrupt occurs between the two executions and it does not follow the instructions. The subroutine statement is as follows:

```

add (a,b);
→ (Interrupts occur here)
mov(b,p);

```

Once interrupt occurs, its control transfers to the appropriate Interrupt Service Routine (ISR) that handles hardware event. When ISR task is completed, IRET (Interrupt Return) instruction is executed, the control returns back to the point of interruption and execution of the original code. Then control returns back to the point of interruption and then follow the MOV instruction.

A device can be used for identification of an input/output interrupt when it registers an input/output interrupt associated with a particular input/output channel.

### 1. Benefits of Input/Output Interrupts

- It is an external analogy to exceptions.
- It allows response to unusual external events without an inline overhead (polling).
- The processor initiates and performs all I/O operations.

- The interrupt can be produced by a device to a processor when it is ready.
- The data is transferred into the memory through interrupt handler.
- The control returns to the program which is currently in use.

## NOTES

### 2. Causes of the Interrupt

The interrupt is caused due to interrupt:

- in any single device
- in a device whose ID number is stored on the address bus
- in processor poll devices

The source of the interrupt is checked and determined by interrupt handler by verifying the associated hardware status registers.

### 3. Interrupts and Response Time

The interrupts are processed in the following way:

- Lower numbers get higher priority.
- Interrupt latency becomes critical for some of the devices.
- Scheduling or ordering has great impact on interrupt latency.
- A non-preemptive priority system gets affected by interrupt and causes delay in packet transmission.
- The interrupt in any device or system cannot be re-interrupted.
- All the pending interrupts are sequentially processed in order of priority.

### 4. Functioning of Input/Output Interrupts

The following are the functioning characteristics of input and output interrupt:

- The processor organizes all the input/output operations for smooth functioning.
- Device takes more than normal time to perform input/output operations.
- After completing the input/output operation the device interrupts the processor.
- Processor then responds to the interrupt and transfers the data to the destination.
- The input/output operation thus successfully completes.

### 5. Examples of Input/Output Interrupt

- Interrupt caused by external devices.
- Interrupt stops the executing code and calls the dynamic procedure.
- Interrupt causes delay in operations.
- Interrupt state is saved (as an exception) and the control passes to the interrupt handler.
- Once the interrupt is handled the control returns to the program it was currently working.

## NOTES



Fig. 2.32 Input/Output Interrupt Device

Figure 2.32 shows how input/output interrupt hardware settings can be used for Input/Output Ports and Interrupt Request. The values are estimated by Windows are not correct.

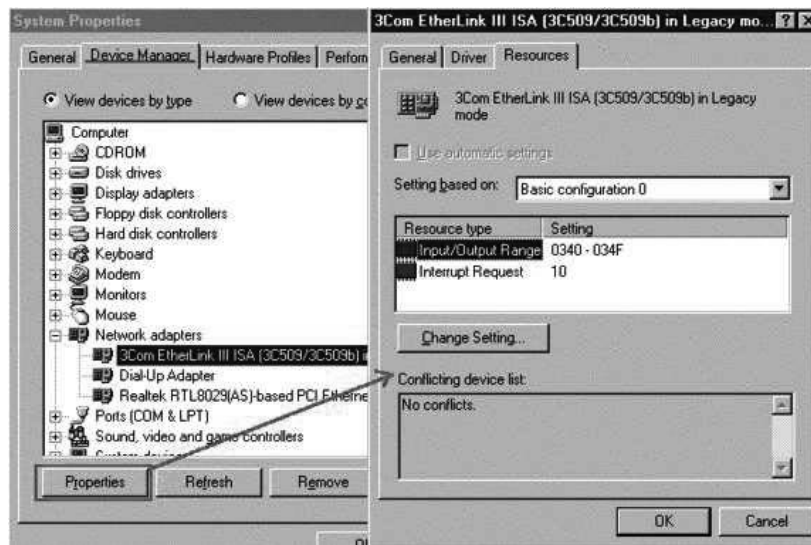


Figure 2.33 Input/Output Interrupt Setting

Figure 2.33 shows when the values estimated by windows are changed, it works properly.

## 2.10 COMPLETE COMPUTER DESCRIPTION

### 1. Speed

Internal processes of computers operate at the speed of light, limited only by the programs that control these processes, and the quantum of data under process. The speed with which computers perform is way beyond human capabilities. To express it differently, a computer does in one minute what a human being could take a lifetime to do.

## NOTES

While referring to the speed of computers, we do not talk in terms of seconds or even milliseconds ( $10^{-3}$ ). The units of speed are in microseconds ( $10^{-6}$ ), nanoseconds ( $10^{-9}$ ), and even picoseconds ( $10^{-12}$ ). A powerful computer is capable of performing about 3–4 million simple instructions per second.

### 2. Accuracy

The accuracy of a computer is consistently high. Errors can occur, but these are almost always due to human error rather than technological weaknesses. Imprecise thinking by the programmer, inaccurate data, or poorly designed systems is the origin of error. Computer errors arising due to incorrect data input or unreliable programs are often referred to as GIGO (Garbage-In-Garbage-Out).

### 3. Diligence

Unlike human beings, a computer does not suffer from limitations associated with living beings like tiredness and lack of concentration, and hence can work for hours at a stretch without errors arising from the above (non-existent) faults. As a result, computers score over human beings in performing routine tasks that require a high degree of accuracy. If a million calculations need to be performed, a computer will perform the millionth calculation with exactly the same accuracy and speed as the first one.

### 4. Versatility

Computers are capable of performing almost any task, provided the task can be reduced to a series of logical steps. For example, a task such as preparing a payroll can be broken down into a logical sequence of operations, and is therefore ideal for computerised processing.

Notwithstanding all this, the computer itself has only limited ability and actually performs only four basic operations:

- Exchange of information with the outside world via Input/Output (I/O) devices
- Transfer of data internally within the Central Processing Unit (CPU)
- Performance of the basic arithmetic operations
- Performance of operations of comparison

In one sense, the computer is not versatile because it is limited to the above-mentioned basic functions. Yet, since so many everyday activities can be reduced to interplay between these functions, it means that computers are effectively, highly ingenious and versatile devices.

### 5. Intelligence

A computer does not possess any intelligence of its own. It can perform only those tasks that can be broken down into a series of logical steps. Therefore, it needs to be told what it has to do and in what sequence.

## 6. Storage

The speed with which computers can process large amounts of information has led to the large-scale information generation, resulting in the information explosion.

Storage of information in a human brain and a computer happens differently. Using its intelligence, the human brain subconsciously shifts through new knowledge and selects what it feels is important and retains it in the memory, and the unimportant information is relegated to the back of the mind or just forgotten. Computers, on the other hand, can store and recall any amount of information by using secondary storage capability (a type of detachable memory). Information can therefore be retained as long as desired and recalled as and when required.

From the 17th century onwards, there have been rapid improvements in the developments of computers. Table 2.20 shows the comparison among various generations of computers.

*Table 2.20 The Comparison among Various Generations of Computers*

First Generation	Second Generation	Third Generation	Fourth Generation	Fifth Generation
Use of vacuum tubes	Use of transistors and diodes	Use of integrated circuits (ICs) and integrated circuits	Use of large scale and very large scale	Use of ICs with ULSI technology (LSI, VLSI)
Limited storage capacity	Increased Storage capacity	More flexibility with input/output	Increased storage	Based on artificial intelligence
Slow speed	Faster speed	Smaller in size and better performance	Considerably faster and smaller	Very fast
Problems of over-heating	Reduction in size and heat generation	Extensive use of high level languages	Modular design, versatility and compatibility	Larger capacity storage (RAID, optical disks)
	High level programming languages (COBOL, FORTRAN)	Remote processing and time sharing	Sophisticated programs and languages for special applications	Support for more complex applications

## NOTES

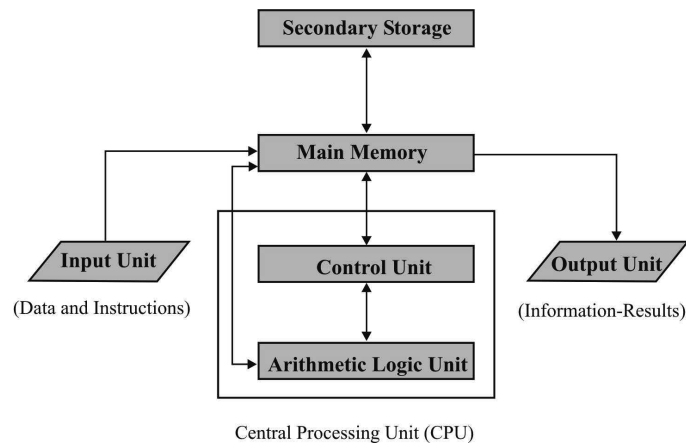
### 2.10.1 Basic Anatomy of the Computer

As seen in earlier sections, the size, shape, cost, and performance of computers have changed over the years, but the basic logical structure has not changed. Any computer system essentially consists of three important parts, namely, input device, Central Processing Unit (CPU) and output device. The CPU itself consists of the main memory, the arithmetic logic unit, and the control unit.

In addition to the five basic parts mentioned above, computers also employ secondary storage devices (also referred to as auxiliary storage or backing storage), which are used for storing data and instructions on a long-term basis.

## NOTES

Figure 2.34 shows the basic anatomy of a computer system.



**Fig. 2.34** Schematic Representation of a Computer System

The five basic operations for converting raw data into relevant information that are performed by all computer systems are the following:

1. **Inputting:** The process of entering data and instructions into the computer system.
2. **Storing:** The process of saving instructions and data so as to make them available for future use, as and when required.
3. **Processing:** Performing arithmetic or logical operations on data, to convert them into useful information. Arithmetic operations include operations of add, subtract, multiply, divide, etc., and logical operations are operations of comparison like less than, equal to, greater than, etc.
4. **Outputting:** This is the process of providing the results to the user. These could be in the form of visual display and /or printed reports.
5. **Controlling:** Refers to directing the sequence and manner of performance of the above operations. Let us now familiarize ourselves with the various computer units that perform these functions.

### Input Unit

Both program and data need to be in the computer system before any kind of operation can be performed. Program refers to the set of instructions which the computer is to carry out, and data is the information on which these instructions are to operate. For example, if the task is to rearrange a list of telephone subscribers in alphabetical order, the sequence of instructions that guide the computer through this operation is the program, whilst the list of names to be sorted is the data.

The Input unit performs the process of transferring data and instructions from the external environment into the computer system. Instructions and data enter the input unit depending upon the particular input device used (keyboard, scanner, card reader, etc). Regardless of the form in which the input unit receives data, it converts these data and instructions into a form that is computer acceptable (Binary Codes). It then supplies the converted data and instructions for further processing to the computer system.



### **Main Memory (Primary Storage)**

Instructions and data are stored in the primary storage before processing and are transferred when there is need, to the Arithmetic Logic Unit (ALU) where the actual processing takes place. Once the processing is completed, the final results are again stored in the primary storage till they are released to an output device. Also, any intermediate results generated by the ALU are temporarily transferred back to primary storage till there is the need at a later time. Thus, data and instructions may move. Thus, data and instructions may move many times back and forth between the primary storage and the ALU before the processing is completed. It may be worth remembering that no processing is done in the primary storage.

### **Arithmetic Logic Unit (ALU)**

After the input unit transfers the information into the memory unit the information can then be further transferred to the ALU where comparisons or calculations are done and results sent back to the memory unit.

Since all data and instructions are represented in numeric form (bit patterns), ALUs are designed to perform the following four basic arithmetic operations: multiply, divide, add, subtract, and logical operations like less than, equal to, greater than.

### **Output Unit**

Since computers work with binary code, the results produced are also in binary form. The basic function of the output unit therefore is to convert these results into human readable form before providing the output through various output devices like terminals, printers, etc.

### **Control Unit**

It is the function of the control unit to ensure that according to the stored instructions, the right operation is done on the right data at the right time. It is the control unit that obtains instructions from the program stored in the main memory, interprets them, and ensures that other units of the system execute them in the desired order. In effect, the control unit is comparable to the central nervous system in the human body.

### **Central Processing Unit**

The control unit and arithmetic logic unit are together known as the *Central Processing Unit* (CPU). It is the brain of any computer system.

### **Secondary storage**

The storage capacity of the primary memory of the computer is limited. Often, it is necessary to store large amounts of data. So, additional memory called secondary storage or auxiliary memory is used in most computer systems.

Secondary storage is storage other than the primary storage. These are peripheral devices connected to and controlled by the computer to enable permanent storage of user data and programs. Typically, hardware devices like magnetic tapes and magnetic disks fall under this category.

## **NOTES**

NOTES

### 2.10.2 Data Representation within the Computer

Information is handled in the computer by electrical components such as transistors, integrated circuits, semiconductors and wires, all of which can indicate only two states or conditions. Transistors may be conducting or non-conducting; magnetic materials are either magnetised or non-magnetised in a direction; a pulse or voltage is either present or not present. All information can therefore be represented within the computer by the presence (ON) or absence (OFF) of these various signals. Thus, all data to be stored and processed in computers is transformed or coded as strings of two symbols, one symbol to represent each state. The two symbols normally used are 0 and 1. These are known as Bits, an abbreviation for BInary digiT. Let us now understand some commonly used terms:

- **BIT:** A bit is the smallest element used by a computer. It holds one of the two possible values. Table 2.21 lists the binary value and its meaning.

Table 2.21 The Binary Value and Its Meaning

Value	Meaning
0	Off
1	On

- A bit which is OFF is also considered to be FALSE or NOT SET; a bit which is ON is also considered to be TRUE or SET. Since a single bit can only store two values, there could possibly be only 4 unique combinations namely,

00 01 10 11

Bits are therefore, combined together into larger units in order to hold a greater range of values.

- **NIBBLE:** A nibble is a group of four bits. This gives a maximum number of 16 possible different values.

$$2^4 = 16 \text{ (2 to the power of the number of bits)}$$

- **BYTES:** Bytes are a grouping of 8 bits (two nibbles) and are often used to store characters. They can also be used to store numeric values.

$$2^8 = 256 \text{ (2 to the power of the number of bits)}$$

- **WORD:** Just like we express information in words, so do computers. A computer 'word' is a group of bits, the length of which varies from machine to machine, but is normally pre-determined for each machine. The word may be as long as 64-bits or as short as 8-bits.

A CPU is the most important component of a digital computer that interprets the instructions and processes the data contained in computer programs. CPU works as a the brain of the computer and performs most of the calculations. It is also referred as processor and is the most important component of a computer. For large computers, a CPU may require one or more Printed Circuit Boards (PCBs) but in the case of PCs it comes in the form of a single chip called a *microprocessor*. PCB is a board that contains the circuitry used to connect the components of a PC. Figure 2.35 shows the block diagram of a CPU. The two main components of a CPU are:

- The ALU, which performs arithmetic and logical operations.
- The Control Unit, which extract instructions from the memory and converts them into a form that the computer can understand and executes them. In this process, it takes the help of the ALU whenever necessary.

## NOTES

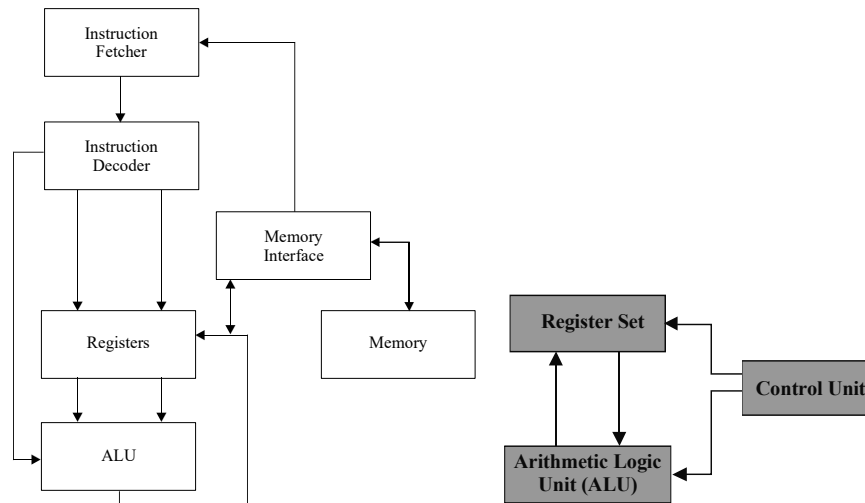


Fig. 2.35 Block Diagram of a CPU Fig. 2.36 Major Components of the PU

Execution of programs is the main function of the computer. The program to be executed is a set of instructions that is stored in the computer's memory. The CPU performs the orders given by the program to complete a specific task. Also, all the major comparisons and calculations are carried out inside the CPU. The CPU is also responsible for controlling and activating the operations of various units of the computer system. It activates the peripherals to perform input or output.

Three major components make up the CPU (Refer Figure 2.36):

- The register set (associated with the main memory) that stores the intermediate data during the execution of instructions.
- The Arithmetic Logic Unit (ALU) which executes the needed microoperations for performing the instructions, and
- The control unit that supervises the information transfer among the registers, and also instructs the ALU for performance of specific operation.

### Control Unit

The Control Unit plays an important role in the functioning of the CPU itself and transfer of data/information from a device to the CPU or vice versa. It does not perform the actual processing on the data but manages and coordinates the entire computer system including the input and the output devices. It retrieves and interprets the instructions from the program stored in the main memory, and issues signals that cause other units of the system to execute them.

It does this through some special purpose registers and a decoder. The special purpose register called the *instruction register* holds the current instruction to be executed, and the *program control register* holds the next instruction to be executed. The decoder interprets the meaning of each instruction supported by

the CPU. Each instruction is also accompanied by a Microcode, i.e., the basic directions to tell the CPU how to execute the instruction.

### 2.10.3 Design of a Basic Computer

#### NOTES

Personal computers (Refer Figure 2.37) are microcomputers commonly used for commercial data processing, desk top publishing (DTP), engineering applications, and so on Figure 2.38 shows computers and its peripherals.

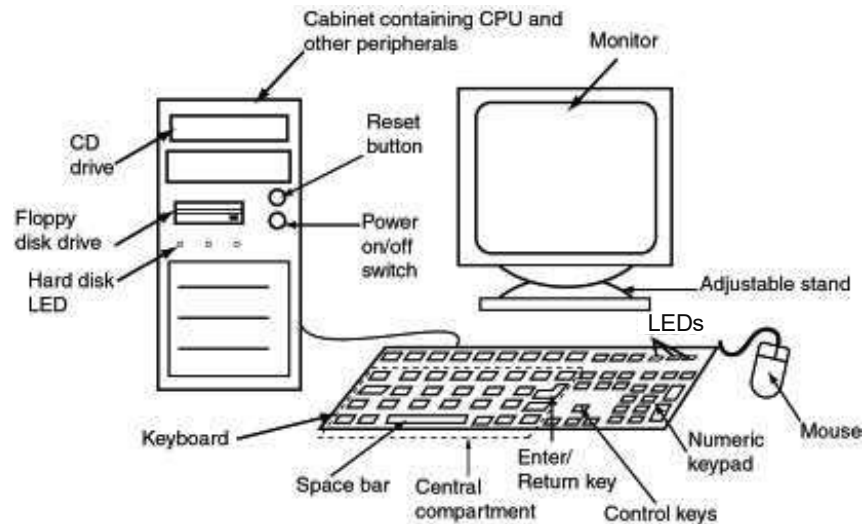


Fig. 2.37 Personal Computer

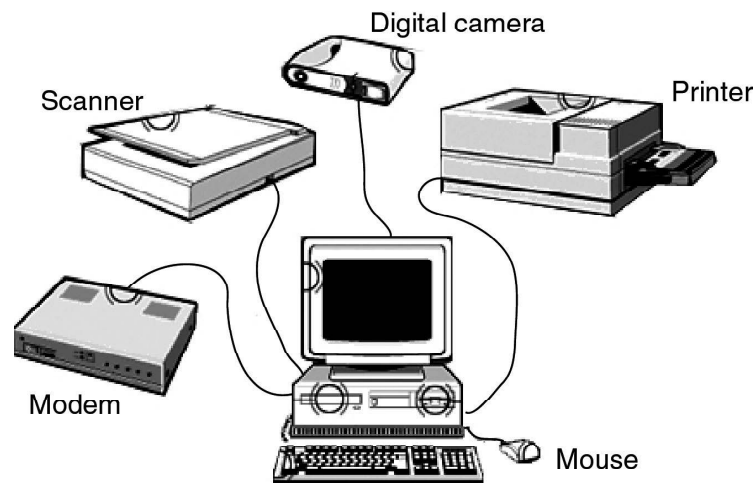


Fig. 2.38 Computer and Its Peripherals

### 2.10.4 Components of a Computer System

A personal computer comprises a Hard Disk Drive (HDD), RAM (Random Access Memory), Processor, a keyboard, a Floppy Disk Drive (FDD), a mouse, a CD drive, a colour monitor, and ROM (Read Only Memory). The RAM, ROM, microprocessor, and other circuitry are connected on the motherboard, which is a single board as shown in Figure 2.39.

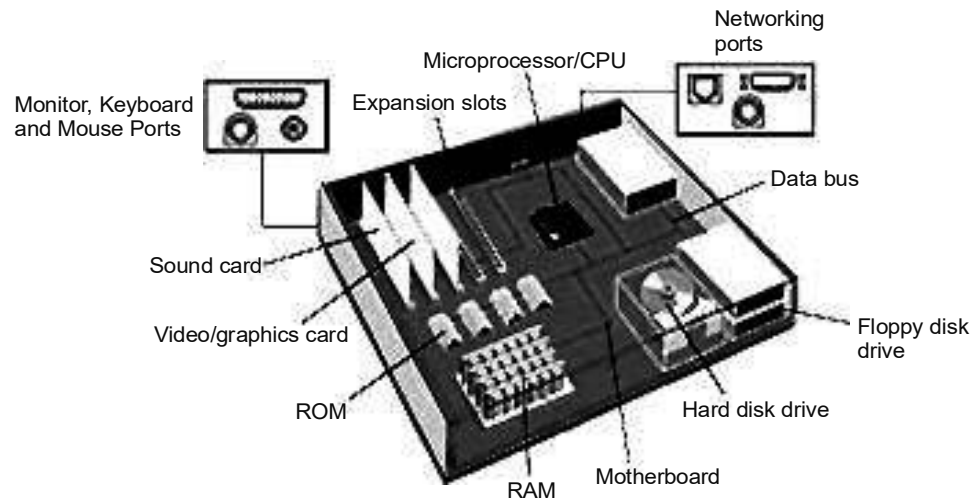


Fig. 2.39 Motherboard and CPU

## NOTES

### Processor (Pentium IV)

Pentium IV processor is the microprocessor which has the control unit, memory unit (register) and arithmetic and logic unit (Refer Figure 2.40). Electronic engineers call this processor the computer. The processing speed of a computer depends on the clockspeed of the system and is measured in Mega Hertz (MHz). The latest Pentium IV processor is available with a clockspeed of 1.6 GHz.

The Intel Corporation's Pentium processors are used in most personal computers. Motorola, Cyrix and AMD (Advanced Micro Devices) are other makers of processors which are also used in personal computers.

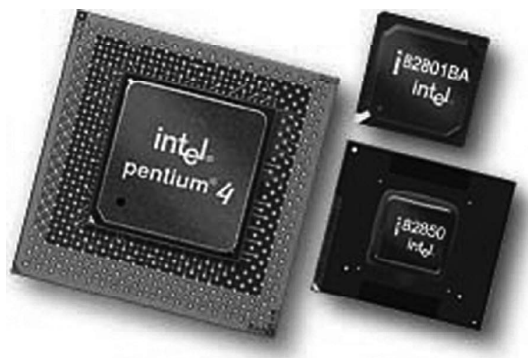


Fig. 2.40 A Microprocessor

### 2.10.5 Machine Language

The computer can understand only a binary-based language. This is a combination of 0s and 1s. Instructions written using sequences of 0s and 1s are known as machine language. First-generation computers used programs written in machine language.

Machine language is very cumbersome to use and is tedious and time consuming for the programmer. It requires thousands of machine language instructions to perform even simple jobs like keeping track of a few addresses for mailing lists.

## NOTES

Every instruction in machine language is composed of two parts – the command itself, also known as the ‘Operation Code’ or opcode (like add, multiply, move, etc.), and the ‘Operand’ which is the address of the data that has to be acted upon; for example, a typical machine language instruction may be represented as shown here.

OP Code	Operand
001	010001110

The number of operands varies with each computer and is therefore computer dependent.

It is evident from the above that to program in machine language, the programmer needs information about the internal structure of the computer. He will also need to remember a number of operation codes and will also need to keep track of the addresses of all the data items (i.e., which storage location has which data item). Programming in machine language can be very tedious, time consuming and still highly prone to errors. Further, locating such errors and effecting modifications is also a mammoth task. Quite understandably, programmers felt the need for moving away from machine language.

### Check Your Progress

9. What is a code?
10. What are the alphanumeric codes.
11. Write the steps taken by control unit to execute any set of instructions.
12. How can you define the stages of instruction cycle?
13. Define the input/output interrupt.
14. What are the basic operations of a computer?
15. Define machine language.

## 2.11 ANSWERS TO ‘CHECK YOUR PROGRESS’

1. Temporary Register (TR) is used for storing the temporary data that is calculated during the processing.
2. Examples of the dedicated address registers are as follows:
  - Segment pointer: In a machine with segmented addressing, a segment register holds the address of the base of the segment in the memory. There may be multiple registers, for example one for the operating system and one for the current process and they may be auto indexed.
  - Index registers: These are used for index addressing scheme and may be auto indexed.
  - Stack pointer: When the programmer visible stack addressing is used, the stack is typically in memory and a dedicated register, called stack

pointer, is used which points to the top of the stack. This allows implicit addressing, i.e., push, pop and other stack instructions need not contain an explicit stack operand.

3. The speed of the bus is affected by its length as well as by the number of devices sharing it.
4. The multiplexer selects the source register whose binary information is then placed on the bus and the decoder selects one destination register to transfer the information to, from the bus.
5. **Data Bus:** It is used for the transmission of data. Data lines and the number of bits in a word are similar.

**Address Bus:** It carries the address of the main memory location from where data can be accessed.

**Control Bus:** It is used to indicate the direction of data transfer and to coordinate the timing of events during the transfer.

6. Micro-operations in digital computers are classified into:
  - (i) Register transfer micro-operation
  - (ii) Arithmetic micro-operation
  - (iii) Logical micro-operation
  - (iv) Shift micro-operation
7. The digital circuit that gives the arithmetic sum of two binary numbers (of any length) is known as the binary adder.
8. A logic circuit performs logical binary operations on strings of bits stored in registers.
9. A code is a symbolic representation of discrete elements of information, in the form of letters, numbers or any other varying physical quantities.
10. Alphanumeric codes are the codes that represent alphabetic characters (letters), punctuation marks and other special characters.
11. The steps taken by control unit to execute a set of instructions:
  - It reads the address of memory location.
  - It reads the instruction from the memory.
  - It sends instructions to decoding circuit.
  - It addresses the data required from the memory.
  - It sends the result to the memory or keeps it in same register to await its chance in queue.
  - It takes help from program counter to fetch next set of instructions.
12. The stages of instruction cycle:
  - (i) **Fetch:** In this step, an instruction is loaded from the memory onto the CPU registers. All the instructions must be fetched before they can be executed.
  - (ii) **Decode:** In this step, the control unit decodes the instructions.

## NOTES

## NOTES

(iii) Derive effective address of the instruction: the effective address of the instruction is read from memory.

(iv) Execute: In this step, the action represented by instruction is performed.

13. Input/output interrupt is an external hardware event which causes the CPU to interrupt the current instruction sequence.
14. Basic operations of a computer are: inputting, storing, processing, outputting and controlling.
15. The computer can understand only a binary-based language. This is a combination of 0s and 1s. Instructions written using sequences of 0s and 1s are known as machine language. First-generation computers used programs written in machine language.

---

## 2.12 SUMMARY

---

- A digital system is a sequential logic system in which flip-flops and gates are constructed. The register transfer logic methods focus on how adders, decoders and registers use expressions and statements which resembles the statements used in programming language.
- Arithmetic microoperations perform arithmetic or number operations; logic performs AND, OR, XOR operation; and shift microoperations perform shift register.
- Digital systems contain the set of registers and their functions in the internal organization of the computer. The main function is that they control signals to initiate the sequence of microoperations to perform the functions.
- If a certain condition is true, microoperation is activated as per requirement.
- In register transfer, control function is similar as 'if' statement in a programming language.
- Control functions use control signal to perform microoperations. If the control signal comes as 1, the operation takes place.
- Data Bus: It is used for the transmission of data. Data lines and the number of bits in a word are similar.
- Address Bus: It carries the address of the main memory location from where data can be accessed.
- Control Bus: It is used to indicate the direction of data transfer and to coordinate the timing of events during the transfer.
- A bidirectional bus for carrying data between two units is called a data bus. A unidirectional bus used to carry memory addresses is called memory bus.
- An instruction constitutes a set of micro-operations. You can define a micro-operation as an elementary operation that is performed on the information stored in one or more registers during one clock pulse.



- Register Transfer Micro-Operation: A micro-operation that transfers binary information from one register to another.
- The incrementer circuit adds 1 to a number in a register. This operation can be easily implemented with a binary counter.
- ‘Selective-Set’ sets to 1 the bits in register A where there is a corresponding 1 in register B. The bit corresponding to 0 in register B remains unchanged.
- ‘Selective-complement’ complements the bits in register A where there is a corresponding 1 in register B. The bit corresponding to 0 in register B remains unchanged.
- The Gray code pertains to a class of codes called minimum change codes, in which only one bit in the code group changes when going from one step to the next. Gray code is not an arithmetic code.
- A code that uses  $n$ -bit strings need not contain  $2^n$  valid code words. An error-detecting code has the property that corrupting or garbling a code word will likely produce a bit string that is not a code word (a non-code word).
- The Hamming distance between two code words is defined as the number of bits that must be changed for one code word to another. It is actually a method for constructing codes with a minimum distance of 3.
- An instruction is divided into a number of fields and is represented as a sequence of bits. Each of the fields constitutes an element of the instruction. A layout of an instruction is termed as the instruction format.
- Control unit fetches the instructions from the memory one after another for execution unit where all the instructions are run and executed.
- Input/output interrupt is an external hardware event which causes the CPU to interrupt the current instruction sequence.
- Internal processes of computers operate at the speed of light, limited only by the programs that control these processes, and the quantum of data under process. The speed with which computers perform is way beyond human capabilities. To express it differently, a computer does in one minute what a human being could take a lifetime to do.
- A computer does not possess any intelligence of its own. It can perform only those tasks that can be broken down into a series of logical steps. Therefore, it needs to be told what it has to do and in what sequence.
- The control unit and arithmetic logic unit are together known as the Central Processing Unit (CPU). It is the brain of any computer system.
- The computer can understand only a binary-based language. This is a combination of 0s and 1s. Instructions written using sequences of 0s and 1s are known as machine language. First-generation computers used programs written in machine language.

## NOTES

## NOTES

---

### 2.13 KEY TERMS

---

- **Data Bus:** It is used for the transmission of data. Data lines and the number of bits in a word are similar.
- **Address Bus:** It carries the address of the main memory location from where data can be accessed.
- **Control Bus:** It is used to indicate the direction of data transfer and to coordinate the timing of events during the transfer.
- **Arithmetic Micro-Operation:** A micro-operation that performs arithmetic operations on numeric data stored in registers.
- **Storing:** The process of saving instructions and data so as to make them available for future use, as and when required.
- **Central Processing Unit (CPU):** The control unit and arithmetic logic unit are together known as the Central Processing Unit (CPU). It is the brain of any computer system.

---

### 2.14 SELF-ASSESSMENT QUESTIONS AND EXERCISES

---

#### Short-Answer Questions

1. Mention the conditions which are required to control the register transfer.
2. Define system service programs.
3. Define the booting process.
4. What is the role of the binary incrementor?
5. What are shift micro-operations?
6. Give the concepts of instruction and instruction code.
7. What does a BCD code do?
8. What is the work of timing and control unit?
9. Give the instruction set supported by CPU.
10. What are the benefits of input/output and interrupt?
12. What is the use of machine language?
12. Define the basic design of a Computer System.

#### Long-Answer Questions

1. Define storage registers. Write the functions and prototypes used in transferring registers.
2. Explain the properties of register transfer.
3. Discuss in detail the significance of bus system giving examples of each types.

4. Explain various arithmetic micro-operations.
5. Explain the functions of the following:
  - (i) Selective-Set
  - (ii) Selective-Complement
  - (iii) Mask
6. Describe the three types of shift operations using examples.
7. What do you mean by instruction cycle? What are the steps taken to execute an instruction? Briefly explain with the help of examples.
8. How is an instruction represented in a computer? Give examples.
9. Explain the functions of control unit.
10. Explain the functions of input/output interrupts.
11. Describe in detail the basic structure of a computer.
12. Discuss the basic anatomy and design of a Computer System giving the significance of each component.
13. What is a machine language? Explain the significant properties of machine language.

## NOTES

---

### 2.15 FURTHER READING

---

- Tanenbaum, Andrew S. 1999. *Structured Computer Organization*, 4th Edition. New Jersey: Prentice-Hall Inc.
- Mano, M. Morris. 1993. *Computer System Architecture*, 3rd Edition. New Jersey: Prentice-Hall Inc.
- Bartee, Thomas C. 1985. *Digital Computer Fundamentals*. New York: McGraw-Hill.
- Mano, M. Morris. 1979. *Digital Logic and Computer Design*. New Delhi: Prentice-Hall of India.
- Leach, Donald P. and Albert Paul Malvino. 1994. *Digital Principles and Applications*. New York: McGraw-Hill.
- Mano, M. Morris. 2002. *Digital Design*. New Delhi: Prentice-Hall of India.
- Kumar, A. Anand. 2003. *Fundamentals of Digital Circuits*. New Delhi: Prentice-Hall of India.
- Stallings, William. 2007. *Computer Organisation and Architecture*. New Delhi: Prentice-Hall of India.



---

# UNIT 3 COMPUTER PROGRAMMING AND MICRO-PROGRAMMING

---

## NOTES

### Structure

- 3.0 Introduction
- 3.1 Objectives
- 3.2 Assembly Language
  - 3.2.1 Assembler
  - 3.2.2 Program Loops
  - 3.2.3 Programming Arithmetic and Logic
  - 3.2.4 Subroutines
  - 3.2.5 Input/Output Programming
- 3.3 Micro-Programmed Control
  - 3.3.1 Address Sequencing
  - 3.3.2 Micro-Program Example
  - 3.3.3 Design of Control Unit
- 3.4 Answers to 'Check Your Progress'
- 3.5 Summary
- 3.6 Key Terms
- 3.7 Self-Assessment Questions and Exercises
- 3.8 Further Reading

---

## 3.0 INTRODUCTION

---

Programming or coding is the art of writing and testing a code written in a symbolic language. A program must be efficient, reliable and usable. In the program loops, programming arithmetic and logic the control section is responsible for decoding the instructions, such as, add, load and store. All these steps are generated and managed by the control unit of the processor.

In address sequencing, apart from execution of instructions, another important function of the microprogrammed control unit is to generate the address of the next sequence. The hardware that controls the address sequencing must be capable of sequencing the microinstructions within a routine and be able to branch from one routine to another and the opcode mapping maps the bits of instruction to an address from control memory.

Control unit in a computer does the job of managing various components of the computer. Various tasks performed by this unit are: reading and decoding of program instructions, converting them into a set of control signals that activates other components of the computer. In advanced computer system, control systems may modify the order of few instructions for bringing improvement in performance.

In this unit, you will study about the assembly language, assembler, program loops, programming arithmetic and logic, subroutines, input/output programming, micro programmed control, address sequencing, micro program example and design of control unit.

---

## 3.1 OBJECTIVES

---

### NOTES

After going through this unit, you will be able to:

- Elaborate on the assembly language
- Define the term assembler
- Describe program loops
- Know programming arithmetic and logic
- Describe subroutines
- Explain input-output programming
- Illustrate address sequencing and microprogramming
- Explain the block diagram of micro-programmed control
- Elaborate on the design of control unit

---

## 3.2 ASSEMBLY LANGUAGE

---

Assembly language was the first step in the evolution of programming languages. It used mnemonics (Symbolic Codes) to represent operation codes and strings of characters to represent addresses. Instructions in assembly language may look as given in Figure 3.1:

Operation	Operation address
READ	M
ADD	L

*Fig. 3.1 Assembly Language Instruction*

Assembly language was designed to replace each machine code by an understandable mnemonic, and each address with a simple alphanumeric string. It was matched to the processor-structure of a particular computer and was therefore (once again) machine dependent. This meant that the programs written for a particular computer model could not be executed on another one. In other words, an assembly language program lacked portability.

A program written in assembly language needs to be translated into machine language before the computer can execute it. This is done by a special program called 'Assembler' which takes every assembly language program and translates it into its equivalent machine code. The assembly language program is called the source program, while the equivalent machine language program is called the object program. It may be useful to know that an assembler is a system program supplied by the computer manufacturer.

Second generation computers used assembly language.

### 3.2.1 Assembler

Assemblers are computer programs that translate programs expressed in the assembly language into the machine code. Each computer, therefore, has its own

assembler. Programs can be written in the assembly language and converted into the machine code before execution using assemblers.

Assembler converts programs written in assembly language to an object file containing machine readable code. Assembler does not define addresses of external function in the assembly source file. This is supplied by the linker when activated. To invoke assembler following command are given at command line:

```
$ as hello.s -o hello.o
```

An -o option is used for output file. In the above code object file 'hello.o' keeps machine instructions for this program. But this program has no defined reference to print file command, printf.

An assembler is a utility program for translating statements written in assembly language into machine code of the target computer. The assembler translates isomorphically, as 1:1 mapping, from mnemonic in these statements to machine instructions.

An assembler completes the work of assembly in two phases and for this the source code is read two times. The assembler, in first pass, reads the program to collect symbols defined with offsets in a table known as symbol table. In second pass, it creates a code in binary format for every instruction in program and then refers to the symbol table to giving every symbol an offset, relating the segment.

Two files are created by the assembler. These are list file and object file. Object file is a binary coded file that has binary code for every instruction in the source code and created on successful assembly of the program without error. Assembler detects syntax errors that vary from one assembler to another. For example take the following lines of code in assembly language:

```
MOVE AX,BX  undeclared identifier MOVE.  
MOV  AX,BL  illegal operands
```

In the first line, there is a word MOVE, that is compared with its mnemonics set and since no matching word in the set is found with this spelling, it does not accept it as a command and takes it as an identifier, and looks in the symbol table for its entry. Since this is not found there, an error 'Undeclared Identifier' is reported. In second line also there is error but for different reasons. Here, two operands are of different type. Assembler for 8086 architecture, according to syntax these two identifiers must be of same type, either in word or in byte. In example above, AX is byte (8-bits) type whereas BL is word type having 16 bits. Logical errors in programs are not detected by assembler.

List file contains source code and it is optional. This contains binary equivalent for each and every instruction with offsets for symbols in source program. This serves the purpose of documentation only.

Commonly known assemblers are, TURBO, MASM (Microsoft Assembler) etc. that run on PCs.

An assembler is a utility program that performs isomorphic translation from statements written in mnemonic into machine instructions and data. There are assemblers that have additional mechanisms for program development, controlling assembly process and help in debugging. Modern assemblers contain macro assemblers that come with macro facility. Modern assemblers create object code

## NOTES

and translate mnemonics into op codes. These assemblers, using symbolic references saves tedious calculations and is a key feature. Macro facilities available with these assemblers perform textual substitution by generating small instructions.

## NOTES

### Types of Assemblers

Assemblers are of two types, One-pass and multi-pass, depending on number of passes it has to make through source code for producing executable codes.

In one-pass assemblers the source code is read once assuming symbols are defined before referencing them in instructions. Such assemblers are fast. Such a feature is now not that important since there are lot of advancements in computer speed and capability.

In two-pass assemblers, a table is created in first pass with all unresolved symbols and then second pass is made for resolving these addresses. In two-pass assemblers there is a flexibility in putting definitions for symbols. These are defined anywhere in the body of the source code. This feature results in coding that is more logical and meaningful. This has made assembly language codes programmer friendly.

There are advancements in design of assemblers that provide language abstraction and these are called high-level assemblers. Such assemblers are more sophisticated. These provide features such as:

- Advanced control structures
- Declaration and invocation of high-level procedure/function
- Abstract data types of high-level that includes sets, structures/records, unions and classes
- Macro processing with sophistication
- Object-Oriented paradigm for encapsulation, inheritance, polymorphism, interfacing, etc.

Normally, in practice, this term is used interchangeably and assembler signifies an assembly language instead of assembler utility. But these two are related and assembly language CP/CMS was written in S/360 assembler and ASM-H was a widely-used S/370 assembler. Programs written in assembly language are the source programs that are to be assembled using assembler which is a utility program.

Thus, assembler is a translator converting each instruction written in assembler language into corresponding instruction in machine language. Assemblers that are less elementary translate source code into a target language that is further combined with other software tools such as library programs, linker, loader, etc., before execution.

### Macroprocessor

Many programs have instruction sequences that are repeated at many places in identical form. A macro processor controls repetitious writing of sequence. A sequence of codes in source language is defined once, and given a name and is subsequently referred to by this name when its use is made. Finding this name in a source program, sequence of codes is substituted at that point. An example of this is given here.



```

A      3, R      ADD contents of R to register 3
A      4, R      ADD contents of R to register 4
A      3, R      ADD contents of R to register 3
A      4, R      ADD contents of R to register 4
R      D C      F '6' Actual value of R in hexadecimal form
    
```

## NOTES

Looking at the preceding program we find that the following sequence is repeated two times.

```

A      3, R
A      4, R
    
```

This repetition could have occurred many times if program needed it. Using macro facility you can attach a name to this and use the name instead of writing same things repeatedly. Every assembler has support for macro. IBM-360 type language is one example supporting a macro language. A macro processor has a language processor with its language.

Name is attached to using macro instruction definition. This is shown below:

Macro definition	MACRO
Macro name	CTR (for example)
Sequence to Instructions	-
	-
	-
END of Macro definition	MEND

With this format, it can be expressed in a macro language as shown below. Here, macro processor replaces each macro cells with the lines as shown.

```

A      3, R
A      4, R
    
```

This process of replacing the sequence of lines of codes is known as expanding die macro. This is shown as follows:

Macro CTR	Expanded Source
A	1, X
A	2, X
MEND	
.	
.	
.	
CTR	A R.
.	A 4, R
.	
CTR	A 3, R
.	A 4, R
.	
R DC F'6'	R DC F'6'

### Linker

To impart modularity to the program, it is broken into several subroutines called modules. A better practice is to put common routines in a separate file. Such

## NOTES

routines are reading a binary number, writing a binary number and like that that could be used by other programs. These files are translated separately and after their successful assembly they are linked together for creating a large file constituting the computer program. Such a program, linking several programs is known as linker.

This linker creates a link file containing binary codes corresponding to compound modules in the program. It also produces a link map containing address information on linked files. Absolute address is not assigned by linker for the program, rather continuous relative addresses are provided to modules linked together, starting from zero. Such a program is relocatable since it can be placed anywhere in main memory for running. Such codes can also run on other machines that are either of the same kind or compatible to the present machine.

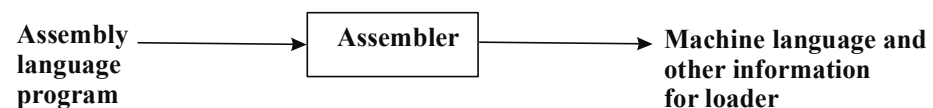
A linker is also known as binder or linkage editor. Input of a linker is a translated program that has its original sources and has symbolic reference with each other. Linker performs the task of resolving these symbolic references to produce a single program. There is not much difference between the source and target languages of linker. Mostly, linking is implemented by a loader only. A linker is considered a part of a loader.

### Loader

A loader puts programs into main memory and then makes preparations for execution. A loader is also a program. Target language of a loader is machine language and its source language is almost machine language. Loading is bound with the task of storage management of operating systems and is mostly performed after assembly. The period of executions of user's program is called execution time.

### Assembler Implementation

An assembly is the process of producing a machine language code equivalent to a source program in an assembly language. This is shown Figure 3.2.



*Fig. 3.2 Assembler*

Externally defined symbols that contain library program have to be indicated to the loader as assembler has no idea about the address of these symbols and it is that task of the loader to locate them in the programs, load them into memory placing values of these symbols in the calling program.

### Assembler and Related Program

Assembler handle program, written in assembly language, contains three types of entities namely, absolute entities, relative entities and object program.

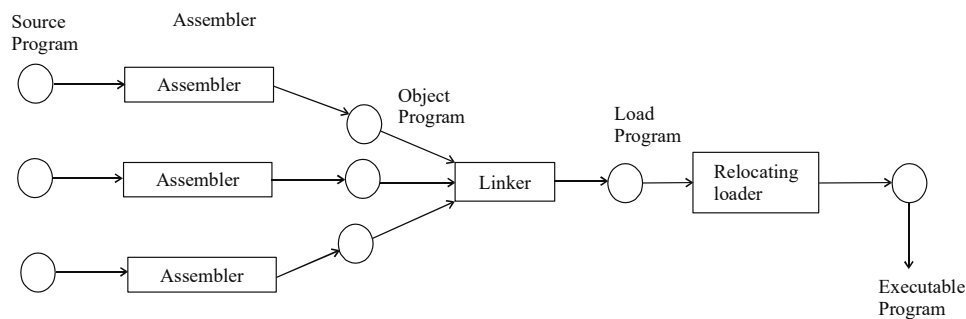
Absolute entities are numeric constants, string constants, fixed addresses, and operation codes. These values signify storage locations that are independent of resulting machine code.

Relative entities have addresses where instructions are stored along with address of working storage.

Object program has identification addresses that are relative. This tells about symbols defined internally as well as externally, for being referenced externally. Linker resolves external references for two or more object programs. The linker may accept many object programs as inputs producing one single program for loading. These are known as a load program.

The module has independence from external references. A module contains machine-code with specification on relative addresses. After actual locations for main storage are known, a relocating loader adjusts relative addresses to these actual locations. The loader output is a program ready to execute, in machine code. These are shown in Figure 3.3 below. If there is a module from single source-language only that does not contain any external references, it does not need a linker to load it and is loaded directly.

## NOTES



**Fig. 3.3** Program Translation Steps

### Load and Go Assembler

This is the simplest assembler program. It takes a program as input whose instructions has one-one correspondence with those of machine language but symbolic names are used for operators and operands. Output of this program is in machine language loaded directly in main memory for execution. The translation is done in a single pass. Resulting machine language program has storage locations, fixed at the time of translation and no change can be made subsequently. These programs can call library subroutines, if possible for them to occupy other locations than those required by the program. There is no provision for combining separate subprograms translated in this manner.

Modern assemblers, especially for RISC based architectures, make optimization of instruction scheduling to make use of CPU pipeline efficiently. Such modern assemblers are MIPS, Sun SPARC, and HP PA-RISC, as well as x86 (-64).

## 3.2.2 Program Loops

### Loop Control Statements

#### NOTES

Loop control structures are used to execute and repeat a block of statements depending on the value of a condition. There are three types of loop control structures/statements in C language.

- (i) `for` statement or `for` loop
- (ii) `while` statement or `while` loop
- (iii) `do-while` statement or `do-while` loop

### **for** STATEMENT OR **for** LOOP

A `for` loop is used to execute and repeat a block of statements depending on a condition. It has the following form.

```
for(<initial value >; <condition>; <increment>)  
{ -----  
    <statement block>  
    ----- }
```

where `<initial value>` is the assignment expression which initializes the value of a variable.

`<condition>` is a relational or logical expression which will have the value **true** or **false**.

`<increment>` is the increment value of the variable which will be added every time.

#### **Example 3.1:**

```
for(i = 1; i <= 10; i++)  
{ s = s + i ;  
  p = p * i ; }
```

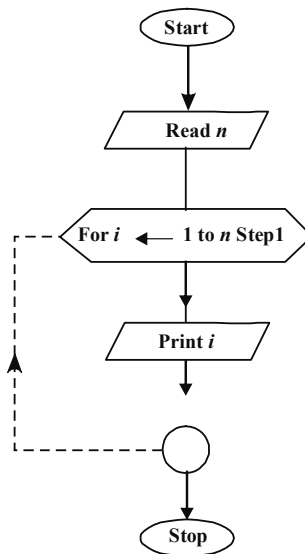
When this statement is executed, the computer assigns initial value to the variable and the condition is evaluated. If the value of the condition is **true**, the statement block will be executed. Now the value of the variable is incremented and the condition is evaluated again and is repeated until the value of the condition is **false**.

1. The braces `{ }` can be omitted when there is only one statement available in statement block.
2. The `<initial value>` is executed only once to initialize the value of the variable.
3. The `<condition>` and `<increment>` are executed on each iteration/repetition.

#### **Example 3.2:**

Write a C program to print natural numbers from 1 to  $n$ .

**NOTES**



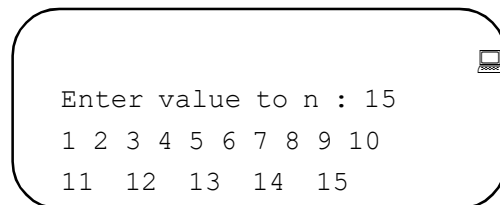
**Solution:**

A for loop can be used to generate and print numbers from 1 to n in steps of 1.

```

/* program to natural numbers upto N */
#include <stdio.h>
#include <conio.h>
main()
{  int n,i;
   clrscr();
   printf("\n Enter value to n : ");
   scanf("%d",&n);
   /* loop to generate and print natural numbers */
   for(i = 1; i <= n; i++)
     printf("%8d",i);
   getch(); }
  
```

When this program is executed, the user has to enter the value of n. The numbers are generated and printed using the for loop as shown below.

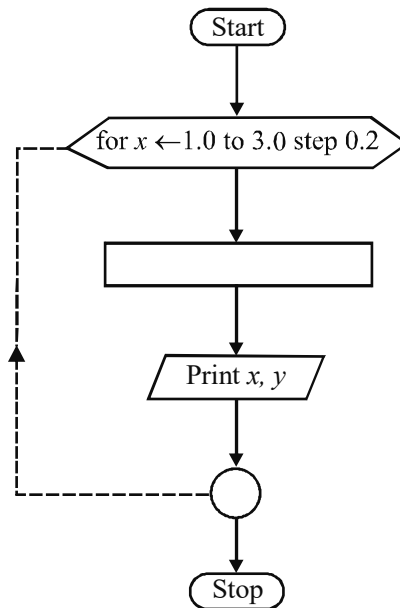


The format "%8d" is used to print 10 values in a line. (Note that about 80 characters can be normally printed in a line).

**Example 3.3:**

The formula  $y = 1.36 \sqrt{1+x+x^3} + x^{1/4} + e^x$  is to be evaluated for x which varies from 1.0 to 3.0 in steps of 0.2. Write a C program to perform this and print a table for various values of x with proper headings.

**NOTES**



**Solution:**

```
/* program to find value of Y */
#include <stdio.h>
#include <math.h>
#include <conio.h>
main()
{ float x,y;
  clrscr();
  printf("\n-----");
  printf("\n  X      Y  ");
  printf("\n-----");
  /* loop to generate and print X and Y */
  for(x = 1.0; x <= 3.0; x = x + 0.2)
  {
    y = 1.36*sqrt(1+x*x*x*x)+pow(x,1.0/4)+exp(x);
    printf("\n %6.2f  %6.2f ", x,y);
  }
  printf("\n-----");
  getch();
}
```

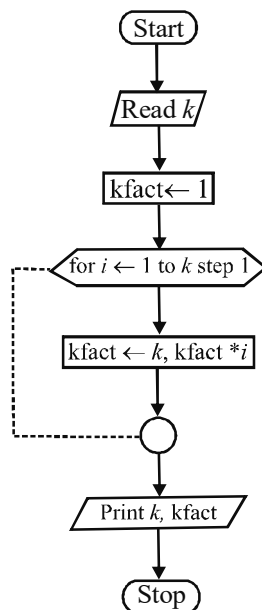
When this program is executed, the computer assigns  $x = 1.0$  and  $y$  is calculated and printed. Then  $x$  is incremented by  $0.2$  and  $y$  is calculated again. The table of values of  $x$  and  $y$  are printed as shown:

x	Y
1.00	6.07
1.20	7.06
1.40	8.23
.	
.	
2.80	24.64
3.0	28.97

The `scanf()` function is not used here; instead a `for` loop is used to generate the value of `x`.

**Example 3.4:**

Write a C program to find factorial of a given integer  $k$ .



**Solution:**

We know that  $k! = 1 \times 2 \times 3 \times \dots \times k$ . The program is written to generate numbers from 1 to  $k$  in steps of 1 and multiply them to get the factorial of  $k$ .

```

/* program to find factorial of a positive integer */
#include <stdio.h>
#include <conio.h>
main()
{ int k,kfact,i;
  clrscr();
  printf("\n Enter an integer : ");

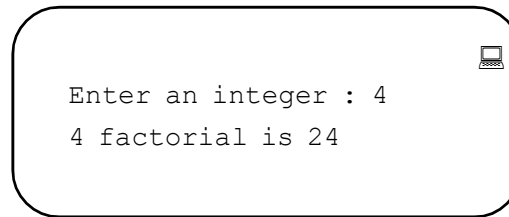
```

**NOTES**

## NOTES

```
scanf("%d",&k);  
kfact = 1;  
/* loop to generate numbers from 1 to n */  
for(i = 1; i <= k; i++)  
    kfact = kfact*i;  
printf("\n %d factorial is %d",k,kfact);  
getch(); }
```

When this program is executed, the user has to enter a positive integer. A `for` loop is used to generate values of `i` from 1 to `k` in steps of 1. Each value is multiplied with `kfact` every time to get factorial of given number `k`.



## NESTED `for` STATEMENT or NESTED `for` LOOP

The statement block of a `for` loop lies completely inside the block of another `for` loop. This is referred as nested `for` loop or nested `for` statement. Any number of `for` statements can be nested. Consider the following example (two loops).

```
for(i=1; i<=3; i++)  
{  
    -----  
    for(j=1; j<=5; j++)  
    {  
        -----  
        <statement block>  
        -----  
    }  
    -----  
}
```

outer loop      inner loop

Note that the block markers `{ }` can be omitted if there is only one statement in the statement block. When this statement is executed, the computer assigns `i = 1` and the inner loop is executed 5 times for `j` values from 1 to 5 in steps of 1. Now `i` is incremented by 1 (i.e., `i = 2`) and the inner loop is executed 5 times for `j` values from 1 to 5. Then `i` becomes 3 and the inner loop is executed 5 times. Note that the statement block is executed and repeated 15 times ( $3 \times 5 = 15$ ).



**Example 3.5:**

Write a C program to sum the following series.

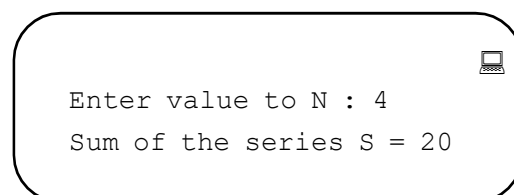
$$S = 1 + (1+2) + (1+2+3) + \dots + (1+2+3+\dots+N)$$

**Solution:**

A nested `for` loop can be written to find the sum of the series. The inner loop is used to find the term values 1, (1+2), (1+2+3) and so on. These are added in outer loop to get the sum of the series.

```
/* program to find sum of series */
#include <stdio.h>
#include <conio.h>
main()
{   int i,j,n,s,term;
    clrscr();
    printf("\n Enter value to N : ");
    scanf("%d",&n);
    s = 0;
    /* outer loop to find sum of series S */
    for(i = 1; i <= n; i++)
    { /* inner loop to find the terms */
        term = 0;
        for(j = 1; j <= i; j++)
        {
            term = term + j;
        }
        s = s + term;
    }
    printf("\n Sum of the series S = %d",s);
    getch(); }
```

When this program is executed, the user has to enter the value to N. The terms are now generated and added to print the sum of the series as shown below.



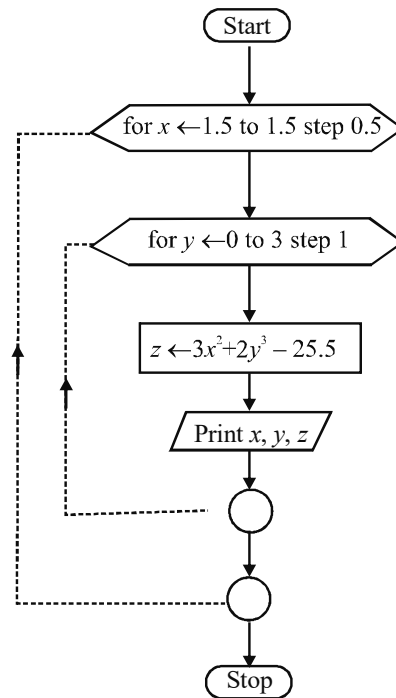
```
Enter value to N : 4
Sum of the series S = 20
```

**Example 3.6:**

$z$  is to be computed as a function of  $x$  and  $y$  according to  $z = 3x^2 + 2y^3 - 25.5$ . Compute the values of  $z$  as  $x$  varies from  $-1.5$  to  $1.5$  in increments of  $0.5$  and  $y$  varies from  $0$  to  $3$  in steps of  $1$ . Write a C program to compute  $z$  for all pairs of  $(x,y)$ .

**NOTES**

## NOTES



### Solution:

A nested for loop can be written to generate various combination of  $x$  and  $y$  and to find  $z$  value for every combination of  $x$  and  $y$ .

```
/* program to find value of Z */
#include <stdio.h>
#include <conio.h>
main()
{ float x,y,z;
  clrscr();
  /* loops to generate values of x and y to find z */
  for(x = -1.5; x <= 1.5; x = x + 0.5)
    for(y = 0; y <= 3.0; y = y + 1.0)
    {
      z = 3*x*x + 2*y*y*y - 25.5;
      printf("\n\n Value of y(%0.2f,%0.2f)
             = %6.2f",x,y,z);
    }
  getch(); }
```

When this program is executed, the computer assigns  $x = -1.5$  and the inner loop is executed for  $y$  values from 0 to 3 in steps of 1. Then  $x$  is incremented to  $-1.0$  and the inner loop is executed again for  $y$  values from 0 to 3 in steps of 1 and so on. The  $z$  value calculated is printed every time along with the values of  $x$  and  $y$ .

The `scanf()` function is not used here; instead `for` loops are used to generate values of  $x$  and  $y$ . The results are printed as follows:

```
Value of y(-1.50,0.00) = -18.75  
Value of y(-1.50,1.00) = -16.75  
.  
.  
Value of y(1.50,3.00) = 35.25
```

## NOTES

### **while STATEMENT OR while LOOP**

A while loop is used to execute and repeat a block of statements depending on a condition. It has the following form.

```
while (<condition>)  
{  
    -----  
    <statement block>  
    ----- }
```

where <condition> is a relational or logical expression which will have the value true or false.

#### **Example 3.7:**

```
i = 1;  
while (i <= 10)  
{  
    s = s + i;  
    p = p * i;  
    i++;  
}
```

When this statement is executed, the computer evaluates the value of the condition. If the value is true, the statement block is executed and is repeated until the value of the condition is **false**. Note that there must be a statement written inside the statement block to change the value of the condition, otherwise the loop is infinity.

### **do-while STATEMENT OR do-while LOOP**

A do-while statement is also used to execute and repeat a block of statements depending on a condition. The form is as follows:

```
do  
{  
    -----  
    <statement block>  
    -----  
}  
while (<condition>);
```

where <condition> is a relational or logical expression which will have the value **true** or **false**.

**NOTES**

**Example 3.8:**

```
i = 1;
do
{
    s = s + i;
    p = p * i;
    i++;
}
while(i <= 10);
```

When this statement is executed, the computer will execute the statement block irrespective of the value of the condition. At the end of statement block, the condition is evaluated. If the value of the condition is **true** the statement block is executed again and is repeated until the condition is **false**.

Note that the statement block is executed at least once for any value (**true** or **false**) of the condition. It is necessary to have a statement inside the block to change the value of the condition, otherwise the loop is infinity.

*Table 3.1 Comparison of the Loop Control Structures*

For Loop	While Loop	Do-While Loop
<p>A <b>for</b> loop is used to execute and repeat a statement block depending on a condition which is evaluated at the beginning of the loop.</p> <p><b>Example</b>  <pre>for(i=1; i&lt;=10; i++) {     s = s + i;     p = p * i; }</pre> </p>	<p>A <b>while</b> loop is used to execute and repeat a statement block depending on a condition which is evaluated at the beginning of the loop.</p> <p><b>Example</b>  <pre>i = 1; while(i &lt;= 10) {     s = s + i;     p = p * i;     i++; }</pre> </p>	<p>A <b>do-while</b> loop is used to execute and repeat a statement block depending on a condition which is evaluated at the end of the loop.</p> <p><b>Example</b>  <pre>i = 1; do {     s = s + i;     p = p * i;     i++; } while(i &lt;= 10);</pre> </p>
<p>A variable value is initialized at the <i>beginning</i> of the loop and is used in the condition.</p>	<p>A variable value is initialized at the <i>beginning</i> or <i>before</i> the loop and is used in the condition.</p>	<p>A variable value is initialized <i>before</i> the loop or <i>assigned inside</i> the loop and is used in the condition.</p>
<p>A statement to change the value of the condition or to increment the value of the variable is given at the <i>beginning</i> of the loop.</p>	<p>A statement to change the value of the condition or to increment the value of the variable is given <i>inside</i> the loop.</p>	<p>A statement to change the value of the condition or to increment the value of the variable is given <i>inside</i> the loop.</p>
<p>The statement block will not be executed when the value of the condition is <b>false</b>.</p>	<p>The statement block will not be executed when the value of the condition is <b>false</b>.</p>	<p>The statement block will not be executed when the value of the condition is <b>false</b>, but the block is executed at least once irrespective of the value of the condition.</p>
<p>A <b>for</b> loop is commonly used by many programmers.</p>	<p>A <b>while</b> loop is also widely used by many programmers.</p>	<p>A <b>do-while</b> loop is used in some cases where the condition need to be checked at the end of the loop.</p>

### goto STATEMENT

The `goto` statement is an unconditional transfer of control statement. It is used to transfer the control from one part of the program to another. The place to which the control is transferred is identified by a statement `label`. The form is as follows:

```
goto label;
```

where `label` is the statement label which is available anywhere in the program.

#### Example 3.9:

```
-----;  
goto display;  
-----;  
-----;  
display:  
-----;
```

When this statement is executed, the control is transferred to the statement label `display` which is followed by a colon. Note that the statements between `goto` and statement label `display` will be skipped because of the transfer.

### break STATEMENT

The `break` statement is used to transfer the control to the end of a statement block in a loop. It is an unavoidable statement to transfer the control to the end of a `switch` statement after executing any one statement block. It has the following form.

```
break;
```

Note that a program can be written without this statement other than in a `switch` statement. Consider the following example.

```
printf("\n press B to break, any other key to continue");  
for(i = 1; i <= 80; i++)  
{  
    ch = getche();  
    if (ch == 'B')  
        break;  
    -----  
    -----  
} ← control is transferred to the end of block
```

When this statement is executed, the control is transferred to the end of the loop and the statements which are written outside the `for` loop will be executed when the key B is pressed. Note that the control can be transferred without repeating the loop 80 times.

### continue STATEMENT

The `continue` statement is used to transfer the control to the beginning of a statement block in a loop. It has the form as follows:

```
continue;
```

## NOTES

## NOTES

Note that this statement is not commonly used. Consider the following example.

```
for (i = 1; i <= 80; i++) ←—————
{ ----- control is
  ch = getche(); transferred to
  if(ch == 'C'  ch == 'c') - the beginning
  { of the block
    printf("\n C for continue is pressed");
    continue; -----
  }
  -----
}
```

When this statement is executed, the control is transferred to the beginning of the loop such that the loop is repeated 80 times irrespective of the key pressed.

Note that the message 'C for continue is pressed' is displayed whenever the key C is pressed.

### **exit () FUNCTION**

The `exit ()` function is used to transfer the control to the end of a program (i.e. to terminate the program execution). It uses one argument in `()` and the value is zero for normal termination or non-zero for abnormal termination. Consider the following example.

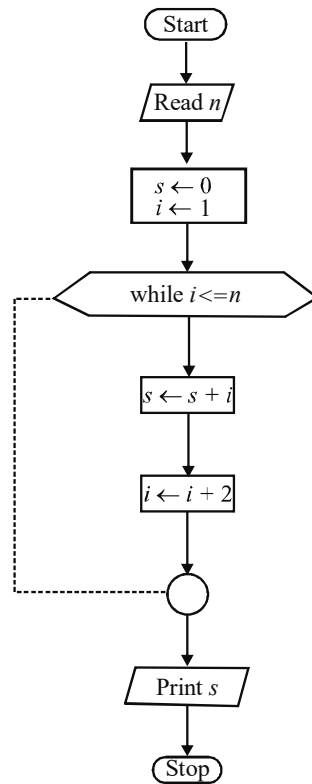
```
if (n < 0)
{
    printf("\n Factorial is not available for negative
numbers");
    exit(0);
}
-----;
```

Note that the program execution is terminated when the value of the variable `n` is negative. The compiler directive `#include <stdlib>` is used when this function is used in a program.

### **Example 3.10:**

Write a C program to find the sum of all odd integers between 1 and  $n$ .

**NOTES**



**Solution:**

A while loop can be used to generate integers from 1 to n in steps of 2 and add them to get the sum of all odd integers.  $S = 1 + 3 + 5 + 7 + \dots + N$

```

/* program to find sum of odd integer between 1 and N */
#include <stdio.h>
#include <conio.h>
main()
{
  int s,i,n;
  clrscr();
  printf("\n Enter value to N : ");
  scanf("%d",&n);
  s = 0;
  i = 1;
  while(i <= n)
  {
    s = s + i;
    i = i + 2;
  }
  printf("\n Sum of odd integers = %d",s);
  getch();
}
  
```

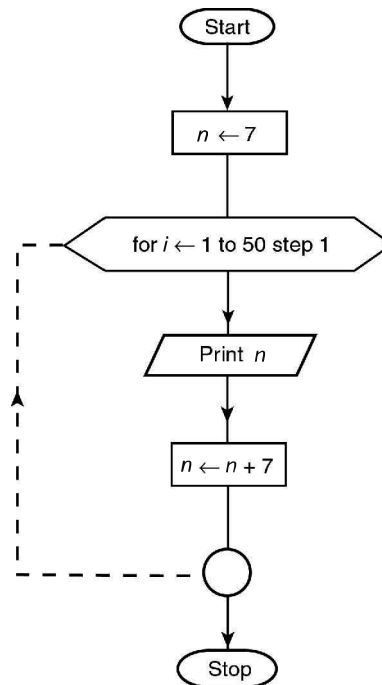
When this program is executed, the user has to enter the value of n. The while loop used will generate odd numbers and add them to get the sum.

## NOTES

Enter value to N : 10  
Sum of odd integers = 25

### Example 3.11:

Write a program to generate the first 50 positive integers that are divisible by 7.



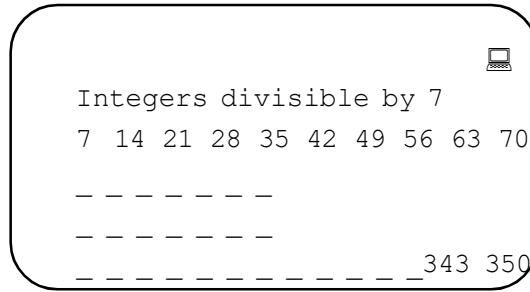
### Solution:

A for loop can be used to print 50 integers and an integer n is used to generate numbers from 7 in steps of 7.

```
/* program to print first 50 positive integers which are
divisible by 7 */
#include <stdio.h>
#include <conio.h>
main()
{ int n,i;
  clrscr();
  printf("\n Integer divisible by 7 \n");
  n = 7;
  for(i = 1; i <= 50; i++)
  {
    printf("%8d",n);
    n = n + 7;
  }
  getch(); }
```



When this program is executed, the computer will not wait for any values, but display automatically the first 50 integers which are divisible by 7.



```
Integers divisible by 7
7 14 21 28 35 42 49 56 63 70
-----
-----
-----
-----343 350
```

## NOTES

### 3.2.3 Programming Arithmetic and Logic

Designing the program requires program specification. It includes technical interpretation of programming requirement. The design of output prepared by report generators and input screen prototypes are considered part of program specifications. It also includes a set of test data to be processed by the program. Testing helps to ensure quality, accuracy and acceptance of the program for use within a system.

In the early years of programming, unstructured programs were written without any standard technique or design method. Programs were written mostly to solve current problems, without much consideration of future needs. An unstructured program uses a linear or top down approach to solve problems. In this approach, the instructions are sequentially followed until a particular condition is reached. The program logic branches off to another part of the program and continues sequentially from there. In a complex program, the branching overlaps.

Now-a-days, structured programs are written with three simple structures as follows:

- (i) Sequence
- (ii) Selection
- (iii) Iteration (Repetition)

Any program could be written with these structures. Programming languages like C, C++ are commonly used to write structured programs.

In a sequence, instructions are followed one after the other in the preset order in which they appear within the program.

#### Example 3.12:

```
printf("\n Enter value to N:");
scanf("%d", &n);
printf("\n Enter value to X:");
scanf("%f", &x);
```

**Selection** means that one of two alternative sequences of instructions is chosen based on a logical condition.

#### Example 3.13:

```
if (n > 0)
    y = 1 + n*x;
```

```
else  
    y = 1 - n*x;  
printf("\n Value of Y = %f", y);
```

## NOTES

*Iteration or Repetition* means that the sequence of instructions is executed and repeated any number of times in a loop until the logical condition is true.

### Example 3.14:

```
do  
{ printf("\n Enter value to N:");  
  scanf("%d", &n);  
  printf("\n Enter value to X:");  
  scanf("%f", &x);  
  if (n > 0)  
    y = 1 + n*x;  
  else  
    y = 1 - n*x;  
  printf("\n Value of Y = %f", y);  
}  
while(x != 0);
```

The three structures are effectively used to organize into modules or routines. A module is a set of instructions that performs one specific function or action within a program. A structured application program contains separate modules for data entry, error checking, processing, and screen or printer output. A module is a small program unit which can be developed by a single programmer and tested separately before combining with the final program. When a program needs modification, only those modules affected have to be changed. This reduces the programming effort and cost.

Consider the following illustration to compare unstructured and structured programs. Unstructured programs are executed in a linear fashion, whereas structured programs execute modules of code.

The logic of unstructured program may be difficult to maintain and any changes need to be done is possible only by the person who developed the program. The structured approach permits flexibility in which programmers may design the same program differently and the designer can link them to form a workable, modifiable program.

### Algorithm

An algorithm is a rough writing of a program. It contains step-by-step instructions to solve a given problem. The steps must appear in the order in which they are executed. The information to be given (input), computed (processing) and printed are identified. Algorithm is discussed in detail in a later chapter.

The sequence of instructions in an algorithm is written with the following characteristics:

- (i) Instructions should be written in the correct sequence in which they are to be executed.

- (ii) Instructions should be precise and unambiguous.
- (iii) Instructions should be executed or repeated only a finite number of times.
- (iv) Check for possible infinite loop.
- (v) Make sure the instructions in the algorithm are written in the correct order.

**Example 3.15:**

Write the algorithm and draw the flow chart to find the biggest of the given two numbers.

**Solution:** For this problem, a new name **big** is used to store the biggest value. Initially **a** is assumed as **big**, then **b** is compared with the existing **big** to get the biggest value.

1. Read  $a, b$
2.  $big \leftarrow a$
3. If  $b > big$  then  
     $big \leftarrow b$
4. Print **big**
5. Stop

**Flow Chart**

Before writing the program code, the sequence of statements and the relationship between various elements are shown with the help of a flow chart or pseudocode. The flow chart is a common method to define the logical steps of flow within the program. It uses various symbols to represent the functions within the program. A flow chart shows the sequence, selection, and iterations within a program. Flow charting is discussed in detail in a later chapter. Consider the following flow chart to find the biggest of given two numbers.

**Rules for Flow Charting**

- (i) Use consistent methods in drawing a flow chart.
- (ii) Use common and easily understandable words.
- (iii) Use consistent words or names in the flow chart.
- (iv) Avoid crossing flow lines in the flow chart.
- (v) Draw the flow chart from top to bottom and left to right.
- (vi) Flow charts that exceed a page should be properly linked using connectors to the portions of the flow chart on different pages.

**Advantages of Flow chart**

- (i) A flow chart can easily explain the program logic to the program development team.
- (ii) A flow chart is useful to prepare detailed program documentation.
- (iii) The flow chart details help prepare efficient program coding.
- (iv) The flow chart helps detect and remove the mistakes in a program.
- (v) A flow chart is useful to test the logic of the program.

**NOTES**

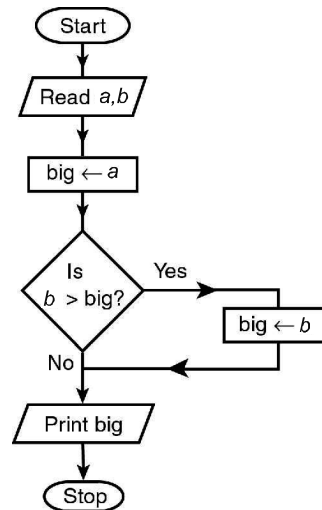
## NOTES

### Limitations of Flow chart

- (i) Flow charting is a laborious process when proper symbols are used.
- (ii) Modifications in flow charting are very difficult and hence consumes more time.
- (iii) Redrawing the flow chart is also a tedious task.
- (iv) No standards are strictly followed to draw the flow chart.

### Example 3.16:

Draw the flow chart to find the biggest of the given two numbers.



### Pseudocode

Pseudocode is a tool used for planning a computer program logic or method. 'Pseudo' means imitation, and 'Code' refers to the instructions written in a computer language. The pseudocode instructions may be written in English, French, German or any vernacular.

Some programmers prefer to write the actual words or **pseudocode** to represent the various steps rather than drawing the flow chart. Pseudocode is useful to design structured programs. A pseudocode looks similar to the actual coding. Consider the following pseudocode to find the biggest of given two numbers.

```
Start program
  Read Number1, Number 2
  If Number 1 > Number 2 then
    Print Number 1
  Else
    Print Number 2
  Endif
End of program
```

The steps in a flow chart or pseudocode can be easily translated into a program code for any high-level programming language, independent of specific language rules.

### **Advantages of Pseudocode**

- (i) Flexibility is a great advantage in using pseudocode.
- (ii) Pseudocode provides a basis for reviewing the program design among all members of program development team and its users.
- (iii) Pseudocode instructions may be easily converted into programming language instructions.
- (iv) Pseudocode instructions can be easily modified.
- (v) Preparing the pseudocode requires less time than drawing a flow chart.

### **Limitations of Pseudocode**

- (i) Reading pseudocode instructions takes more time.
- (ii) There is no standard procedure for preparing pseudocode instructions.
- (iii) Beginners find it difficult to write pseudocode than drawing a flow chart.

### **Writing the Program Code**

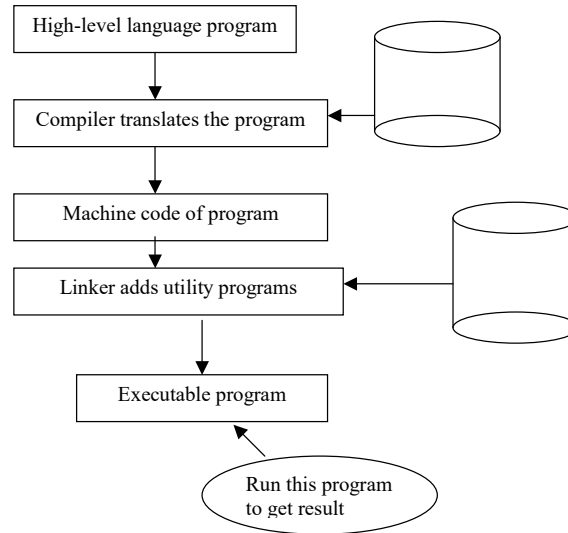
Writing the program code is a straightforward process if sufficient care is taken during program design. The source program code is developed using a high-level language; one instruction or line of code is written for each line of pseudocode. Note that some languages require several lines of code to implement a single line in a pseudocode.

The programs written in a high-level language must be translated to the machine language, which represents internal switch settings as 1's and 0's. Compilers are used to translate high-level language instructions to a machine code. First, the high-level language program code is loaded into the memory along with the compiler. The compiler checks the program for errors in translating the code into machine language. The compiler cannot translate a program code with any syntax error. Note that a syntax error may even be a spelling mistake in the program code.

After compilation of the program, the operating system of the computer activates the linker. The linker program has utilities needed for input, output or processing within the translated program. When linking is complete, the program is ready for use. Note that the input and output devices are linked by the linker program in see Figure 3.4 order to receive data and print results obtained during runtime of the program.

## **NOTES**

**NOTES**



**Fig. 3.4** Functioning of a Program

In order to write computer programs without any logical error, it is recommended programmers prepare a rough writing showing the steps involved in the program. This is called an algorithm.

An algorithm presents step-by-step instructions required to solve any problem. These steps can be shown diagrammatically using a flow chart.

Flow chart is a symbolic or diagrammatic representation of an algorithm. It uses several geometrical figures to represent the operations, and arrows to show the direction of flow. Table 3.2 shows the commonly used symbols in flow charts.

**Table 3.2** Flow Charts: Symbols and Operations

Symbol	Operation	Meaning
	Start/stop	Represents the beginning and the end of the flow chart
	Input/output	Represents the values to be given by the user and the results to be displayed
	Processing	Represents the arithmetic operations to compute a value
	Checking/decision making	Represents the logical checking to decide the flow sequence
	Looping	Represents the looping, which is repeated based on a condition/
	Connector	Represents the continuity of the flow chart in another place/page
	Arrows	Represent direction of flow

It is recommended beginners must practice algorithm and flow charts before starting to write programs.

**Example 3.17:**

Write the algorithm and draw the flow chart to find the sum and product of given two numbers.

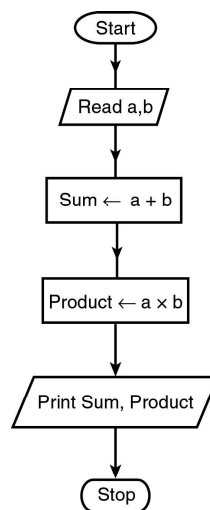
**Solution:** It is necessary to understand the data given in the problem and the results expected.

In this problem, two numbers, say  $A$  and  $B$ , are given (input) and the results, sum ( $A+B$ ) of two numbers and product ( $A \times B$ ) of two numbers, are to be calculated.

*Algorithm*

1. Read  $a, b$
2.  $\text{Sum} \leftarrow a + b$
3.  $\text{Product} \leftarrow a \times b$
4. Print sum, product
5. Stop

*Flow chart*



**Notes:**

- (i) Usually words Read, Accept or Input can be used to represent input operation to give values of variables to the computer.
- (ii) Print, Write or Display can be used to represent output operation to show the results computed by the computer.
- (iii) Back arrow ( $\leftarrow$ ) represents the value obtained by evaluating the right side expression/variable to the left side variable. The symbol '=' can also be used instead of ' $\leftarrow$ ' but it leads to confusion in certain applications. (e.g.  $S = S + X$ ) representing the logical equality and so on.
- (iv) Down arrow ( $\downarrow$ ) is optional.

**Example 3.18:**

Write the algorithm and draw the flow chart to convert the temperature in  $^{\circ}\text{F}$  to  $^{\circ}\text{C}$  using the formula

$$^{\circ}\text{C} = 5/9 (^{\circ}\text{F} - 32)$$

**Solution:** The input variable is F (represents temperature in  $^{\circ}\text{F}$ ) and the output variable is C (representing temperature in  $^{\circ}\text{C}$ ).

**NOTES**

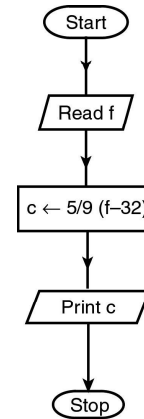
**NOTES**

**Algorithm**

- 1.
- 2.
- 3.
- 4.

**Flow Chart**

- Read F  
 $C \leftarrow \frac{5}{9}(F-32)$   
 Print C  
 Stop



**Example 3.19:**

Write the algorithm and draw the flow chart to find the area of a triangle whose sides are  $a$ ,  $b$ , and  $c$ .

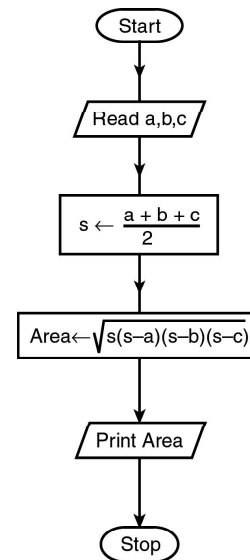
**Solution:** You know that area of a triangle  $= \sqrt{s(s-a)(s-b)(s-c)}$

where,  $s = \frac{a+b+c}{2}$

*Algorithm*

1. Read  $a, b, c$
2.  $s \leftarrow \frac{a+b+c}{2}$
3.  $\text{Area} \leftarrow \sqrt{s(s-a)(s-b)(s-c)}$
4. Print Area
5. Stop

*Flow chart*



**Example 3.20:**

Write the algorithm and draw the flow chart to find the biggest of the given two numbers.

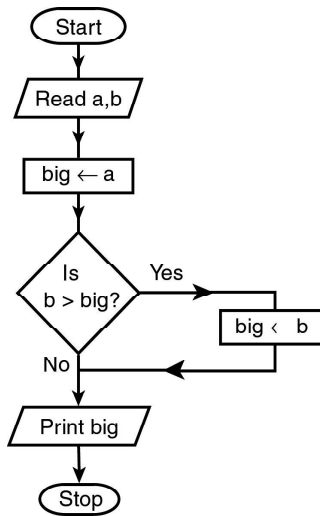
**Solution:** For this problem, a new name `big` is used to store the biggest value. Initially `a` is assumed as `big`, then `b` is compared with the existing `big` to get the biggest value.



**Algorithm**

1. Read  $a, b$
2.  $big \leftarrow a$
3. If  $b > big$  then  
     $big \leftarrow b$
4. Print  $big$
5. Stop

**Flow chart**



**NOTES**

**Note:** There are other methods available to find the biggest value. The method discussed here is the best which can be easily extended for any number of values and is suitable for writing structured programs.

**Example 3.21:**

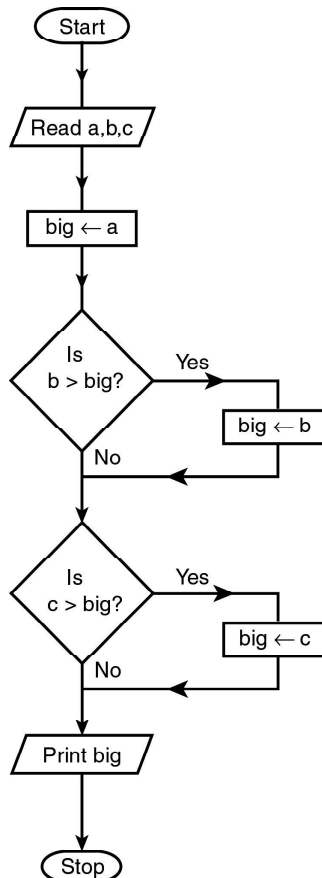
Write the algorithm and draw the flow chart to find the biggest of the given three numbers.

**Solution:** The method discussed in Example 20 is extended for three numbers:

*Algorithm*

1. Read  $a, b, c$
2.  $big \leftarrow a$
3. If  $b > big$  then  
     $big \leftarrow b$
4. If  $c > big$  then  
     $big \leftarrow c$
5. Print  $big$
6. Stop

*Flow chart*



**Example 3.22:**

Draw a flow chart to solve the following series

$$s = x - x^3 + x^5 - x^7 \dots X^n$$

**NOTES**

**Solution:** For this problem, the initial values are assigned as

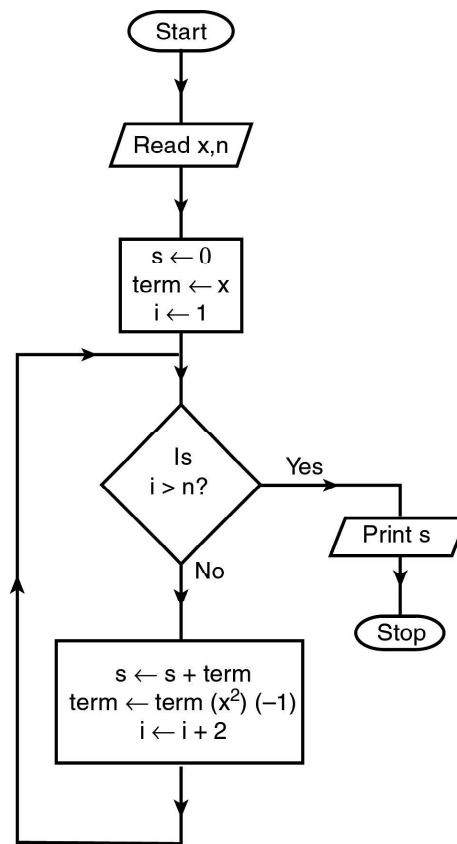
$$s \leftarrow 0$$

term  $\leftarrow x$  represents the first term in the series

$i \leftarrow 1$  represents the power in  $x^1$

Then the value of the term is incremented to get the next term and  $i$  is also incremented accordingly. The term is then added to  $s$  and is repeated until  $i > n$ .

Flow chart



**Example 3.23:**

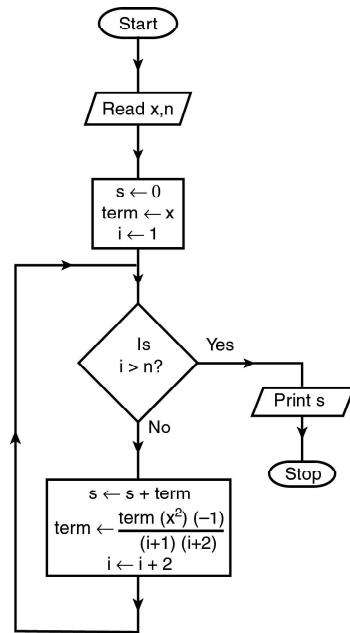
Draw a flow chart to solve the following series:

$$s = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} \dots \frac{x^n}{n!}$$

**Solution:** The method discussed in Example 3.22 is extended by including the denominator.

**NOTES**

*Flow chart*



**Example 3.24:**

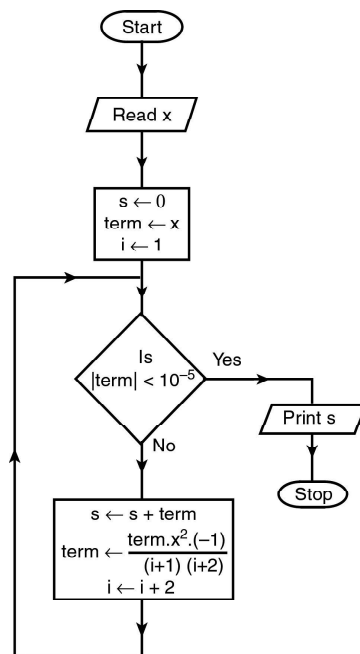
Draw the flow chart to solve the following series (sin x)

$$s = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} \dots \infty$$

omitting those terms which are less than  $10^{-5}$  in magnitude.

**Solution:** The method discussed in Example 3.23 is extended by considering the absolute value of the term.

*Flow chart*



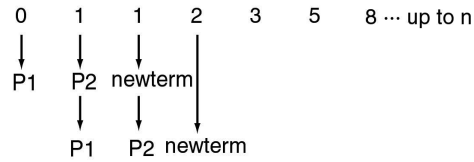
**Note:** The method of solving a series given in this example can be used for any other series of this kind, e.g.  $\cos x$ ,  $e^x$ , and so on.

**Example 3.25:**

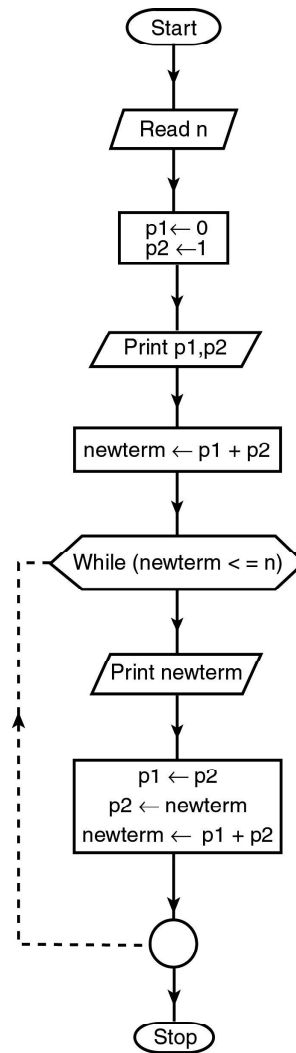
**NOTES**

Draw a flow chart to generate and print the Fibonacci series 0 1 1 2 3 5 8 ... up to  $n$ .

**Solution:** For this series, the preceding two terms are added to get the next term.



*Flow chart*



**Example 3.26:**

Draw a flow chart to find the factorial of a given integer.

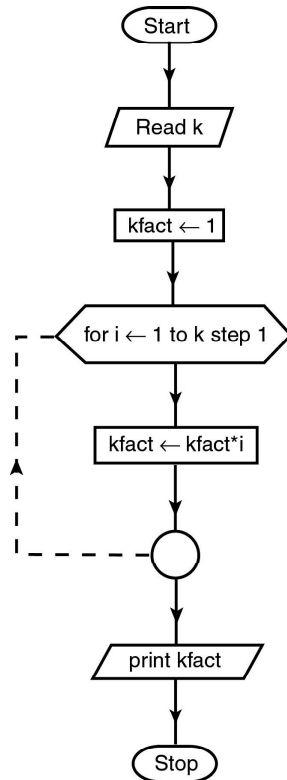
**Solution:** Let 'k' be the integer then

$$k! = 1 \times 2 \times 3 \cdots k$$

To get k!, numbers are generated from 1 to k in steps of 1 and all these numbers are multiplied.

## NOTES

Flow chart



**Note:** Factorial (!) is not available for non-integer and negative numbers. Also note that 0! is 1.

### 3.2.4 Subroutines

A sequence of instructions are known as subroutines. These perform subtask under the head of subprograms. Such subprograms are called subroutines or procedures. The simplest kind of user defined code block is known as a subroutine. Basically, a subroutine is a group of instructions that performs a subtask time delay that is required repeatedly in a program. It is written as a separate unit, apart from the main, and can be called whenever it is necessary. When a main program calls a subroutine, the program execution is transferred to the subroutine and after its completion of its job, the program execution returns to the main program. The microprocessor uses the stack to store the return address of the subroutine. It is also a group of instructions written separately from the main program to perform a function and can be used repeatedly in the main program. For example, if a time delay is required between three successive events, a time delay subroutine can be written once instead of three times. The subroutine is written separately from the main program, and is called by the main program when needed. The subroutine technique enables an efficient use of memory. A subroutine is implemented with two associated instructions: Call (call a subroutine) and Return (return from the

**NOTES**

subroutine). The Call instruction is written in the main program (except in the nested subroutine) to call a subroutine, and the Return instruction is written in the subroutine to return to the main program. When a subroutine is called, the contents of the program counter that is basically the location address of the executable instruction following the Call instruction is stored on the stack, and the program execution is transferred to the subroutine address. When the Return instruction is executed at the end of the subroutine, the memory address stored in the stack is retrieved and the sequence of execution is resumed in the main program. The conditional call subroutines are given in a tabular format in Table 3.3.

*Table 3.3 The Conditional Call Subroutines*

Conditional Call Subroutines	Functions
CALL Z	16-bit Call subroutine if Z flag is set (Z = 1)
CALL NZ	16-bit Call subroutine if Z flag is reset (Z = 0)
CALL C	16-bit Call subroutine if CY flag is set (C = 1)
CALL NC	16-bit Call subroutine if CY flag is reset (C = 0)
CALL M	16-bit Call (On Minus) if S flag is set (S = 1)
CALL P	16-bit Call (On Plus) if S flag is reset (S = 0)
CALL PE	16-bit Call (On Parity Even) if P/V flag is set (P/V = 1)
CALL PO	16-bit Call (On Parity Odd) if P/V flag is reset (P/V = 0)
RET Z	Return if Z flag is set (Z = 1)
RET NZ	Return if Z flag is reset (Z = 0)
RET C	Return if CY flag is set (C = 1)
RET NC	Return if CY flag is reset (C = 0)
RET M	Return (On Minus) if S flag is set (S = 1)
RET P	Return (On Plus) if S flag is reset (S = 0)
RET PE	Return (On Parity Even) if P/V flag is set (P/V = 1)
RET PO	Return (On Parity Odd) if P/V flag is reset (P/V = 0)

A subroutine called by another subroutine is said to be **nested**. The extent of nesting is limited only by the number of available stack locations. When a subroutine calls another subroutine, all return addresses are stored on the stack.

By using instructions of hardware like SCAL and SXIT subroutines are put into practice. Disparate to PCAL, SCAL does not offer a heap indicator of four-word; as a result subroutines have subsequent upgrade and downgrade points:

- Importance in Q and index registers does not alter.
- A related P return address is kept on peak of pile.
- Entire stricture is oriented in relation to the S register and markers cannot be accepted as limits.
- Subroutines don't have limited variables.
- Subroutines are supposed to be placed in identical section to the caller so that SCAL and SXIT don't overpass division limits.
- The entrance and way out of subroutines is faster than actions due to lesser work for commands to finish.

- Subroutines could be announced in the procedures whilst the procedures cannot. They can suggest procedure-local variables as Q is unchanged when a subroutine is described.
- Like all procedures, Subroutines could tackle every worldly variables as DB does not get hampered by working.

## NOTES

### Declaration of Subroutines

Subroutines are stated in a major agenda (global subroutines) surrounded by a procedure (local subroutines). Global subroutines are mentioned merely inside the major program nearby the procedures given that procedures are not in the identical section as main program. Global subroutine declaration should come after procedure announcement.

```
BEGIN
  _____
  | data group |
  _____
  | intrinsics |
  _____
  | and        |
  _____
  | procedures |
  _____
  | subroutines|
  _____
  | main body  |
  _____

  END.
```

Local subroutines should be mentioned simply from procedure in which they are stated. They are announced in part of procedure, following some local data statement, but prior the report of the body.

```
_____
| procedure head |
_____

BEGIN
  _____
  | data declarations |
  _____
  | subroutine decl.  |
  _____
  | statements        |
  _____

  END;
```

The declaration format of a subroutine is identical to that of a procedure, except that there is no option part and no local data group.

## NOTES

```
{ type
{ SUBROUTINE
{ name

head { formal parameters
      { value part
      { specification part

body { statement (possibly compound)
```

For example,

```
INTEGER SUBROUTINE S(A,B,C);
  VALUE A,B,C;
  INTEGER A,B,C;
  S := (A ^ 2) + (B * C);
```

### Invoking Subroutines

Subroutines are invoked by using their identifier in a subroutine call statement and replacing the formal parameters with actual parameters.

```
identifier (parameter list);
```

Parameters can be stacked by asterisk '\*' just as with procedures.

Function subroutines are invoked by using them within an expression:

```
NIX := S(4,5,6) + S(100.20,1);
```

Here is a complete program showing the format of subroutine declarations and invocations:

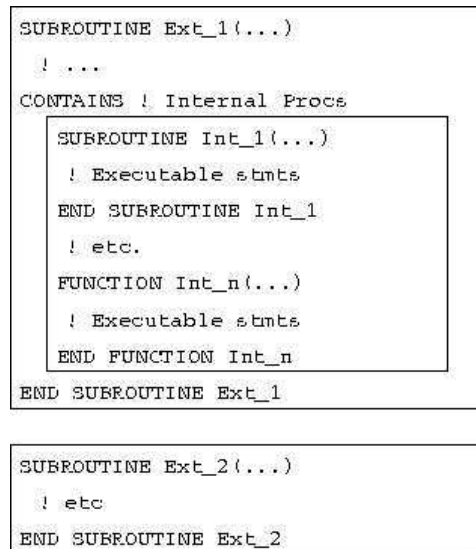
```
TO
    BEGIN <<USE A SUBROUTINE TO SET AN ARRAY
        ZERO>>
        INTEGER ARRAY ADATA(0:50);
        INTEGER I;<<INDEX FOR USE IN SUBROUTINE>>
        { SUBROUTINE ZERO(ARRY,HISUB);
        {     VALUE HISUB;
        {     INTEGER HISUB;
        {     INTEGER ARRAY ARRY;
        { BEGIN
subroutine {I :=0;<<SET INITIAL VALUE INTO SUBSCRIPT>>
declaration {     WHILE I <=HISUB DO
        {     BEGIN
        {     ARRY(I) :=0;
```



```

        {           I ;=I + 1;
        {           END;
        { END<<ZERO>>;
subroutine { <<END OF DECLARATION>>
call      {       ZERO (ADATA, 50) ;<<CALL SUBROUTINE>>
        { END <<MAIN PROGRAM>>.

```



**Fig. 3.5** Declaration of Subroutine

The Figure 3.5 shows the non-recursive subroutine declaration.

### Subroutine Functioning

The features of subroutines are determined by the functioning of the SCAL and SXIT instructions, which work in this manner:

#### SCAL

1. When a subroutine is invoked, the parameters are loaded onto the stack and a SCAL is executed.
2. SCAL loads P + 1 (the return address) onto the stack and branches to a relative address within the current code segment.
3. S relative addressing is used to reference all parameters. Since the top of stack changes constantly, the S relative addresses of the parameters also change constantly.

#### SXIT

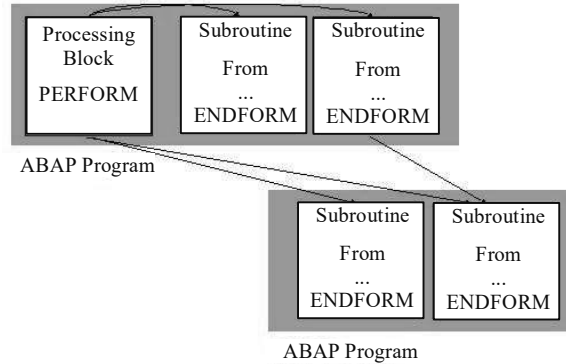
1. On execution of the SXIT, the current top of stack value is used as the P relative return address.
2. Due to S changing constantly, it is possible for the subroutine to use an incorrect return address if the subroutine has explicitly modified the stack.

The above process can be seen in the following example—suppose we have a subroutine SW receiving two integer values. It exchanges them, and exits, leaving them on the stack.

## NOTES

**NOTES**

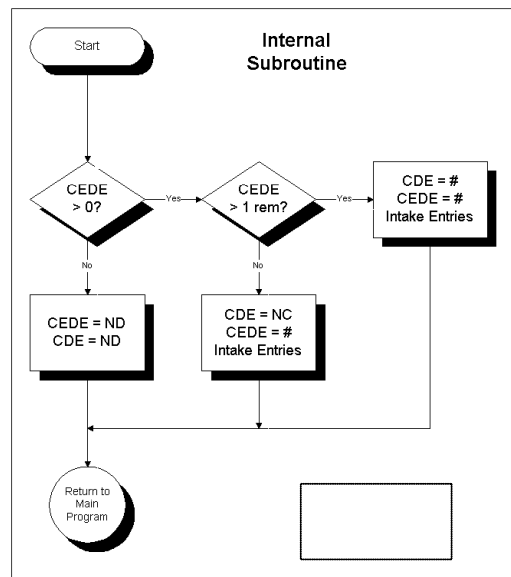
Owing to the nature of the SCAL and SXIT instructions, subroutines are much less flexible and powerful than procedures. Subroutines cannot possess local variables; the only variables available to a subroutine are parameters as DB relative and Q relative (local subroutines only) variables. Also, the user must not explicitly modify the stack within a subroutine without immediately correcting for any changes. All subsequent parameters addressing may be wrong and S may not point to the return address when SXIT is executed.



**Fig. 3.6** Subroutines in ACAP Program

In the Figure, 3.6 data is transferred or passed through the selection screen and ABAP program following the defined parameters. The statement Subroutine is introduced with FORM ... ENDFORM statement.

Subroutine is called in the same program or in external subroutine. If internal subroutine like in Figure 3.7 is called, global data is used to pass values defining parameters between the main program and the defined subroutine. If external subroutine is called, actual parameters are passed through the main program to the formal parameters in the related subroutine.



**Fig. 3.7** Internal Subroutine

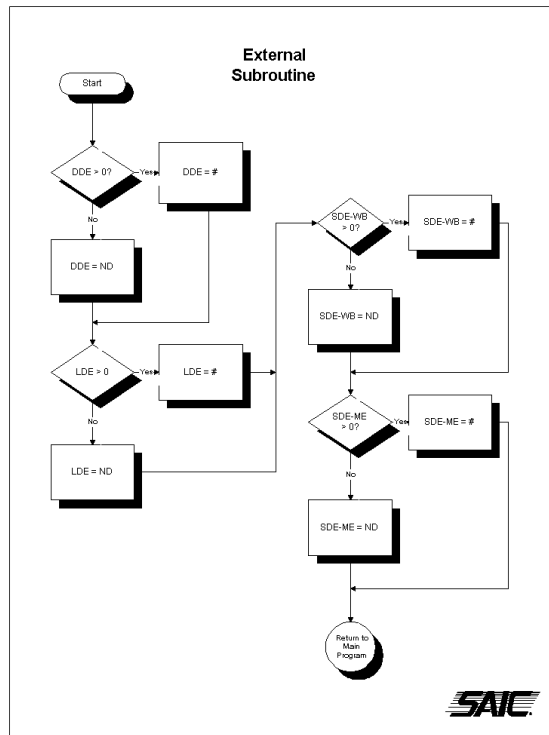


Fig. 3.8 External Subroutine

**NOTES**

**One-Level Subroutine**

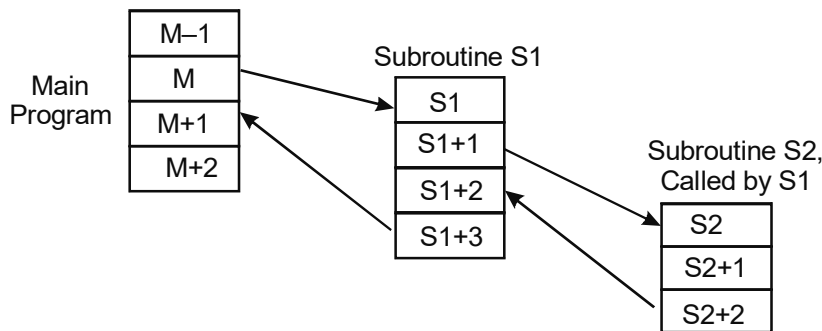


Fig. 3.9 Main Program

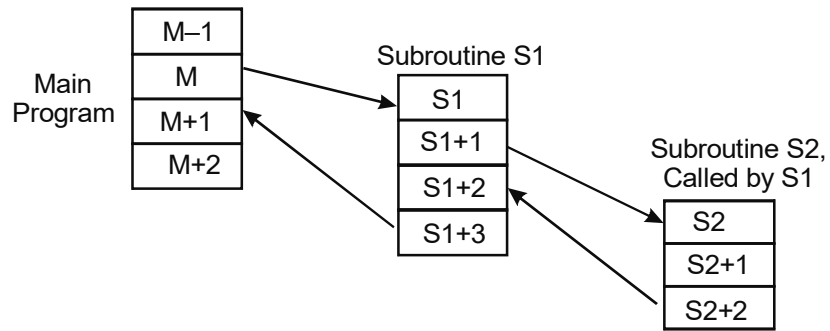
The Figure 3.9 shows the Main Program which defines the sequence of code address in the specific micro-code ROM. The sequences,

$M-1 \rightarrow M \rightarrow S1 \rightarrow S1+1 \rightarrow S2 \rightarrow S2+2 \rightarrow S1+2 \rightarrow S1+3 \rightarrow M+1$  follow the defined steps of the subroutines to complete the task.

**Nested Subroutines**

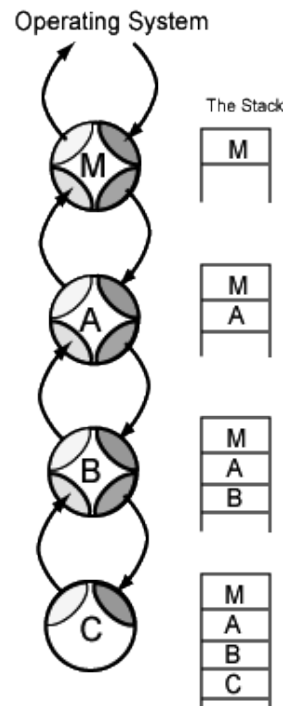
By defining the R-register as Last In First Out Stack, i.e.; 'LIFO Stack', the sequencer can handle nested subroutines.

**NOTES**



*Fig. 3.10 Nested Subroutines*

The Figure 3.10 shows that the subroutines can be nested up to the deepest level of stack. The LIFO stack can be a 4-word memory addressed by 2 bits from an up/down counter known as the stack pointer.



*Fig. 3.11 Nested Subroutine*

The Figure 3.11 shows the main routine connecting to subroutine A, which connects to subroutine B, which links to subroutine C. The subroutines like beads on a double string. Control is passed from the call to prologue and from epilogue back to the caller. All subroutines in a calling chain have all five sections (prolog, call, epilog, body, and regaining control) except the one at the bottom. Each time another subroutine is added to the chain, more data is pushed onto the run-time stack. At the end of the chain of calls the run-time stack has a section of data which is saved register values from each of the subroutines including main. The subroutine which is currently active has its data at the top of the stack subroutine C, in our upside-down stack.

### 3.2.5 Input/Output Programming

Each input/output action returns a value which is 'Tagged' with input and output type in order to distinguish it from other values. This can be defined with function `getChar` used in C as follows:

```
getChar :: IO Char
```

The `IO Char` indicates that when `getChar` is invoked it performs the specific action as defined in the program syntax and returns a character. The actions that do not return any value use the unit type `()`. For example, if you define the `putChar` function in the following way you get no output:

```
putChar :: Char -> IO ()
```

It takes a character input as an argument but returns no useful value. This unit type is similar to type `void` in other programming languages.

The defined keywords and parameters process the sequence of statements and execute each of them in order. A statement can be defined as a set of instructions using `let` or a set pattern to perform the action and give the result using `<-` operator. The braces and semicolons should have proper indentation. Following is an example of simple C program to read and print a character:

```
main :: IO ()
main = do c <- getChar
        putChar c
```

The return function gives an ordinary Boolean value to the realm of input/output operations.

```
f :: Int -> Int -> Int
```

### NOTES

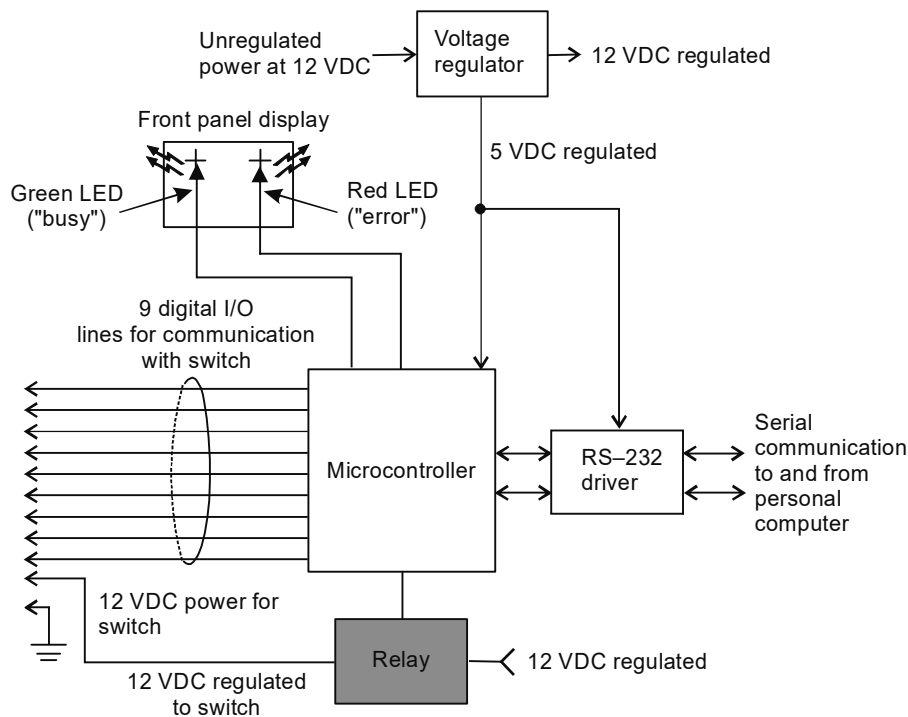
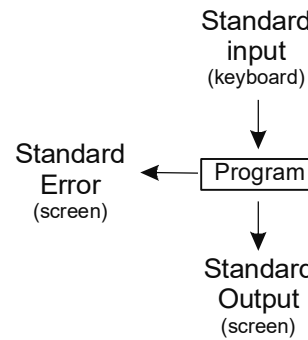


Fig. 3.12 Input/Output Setting in Digital Computer

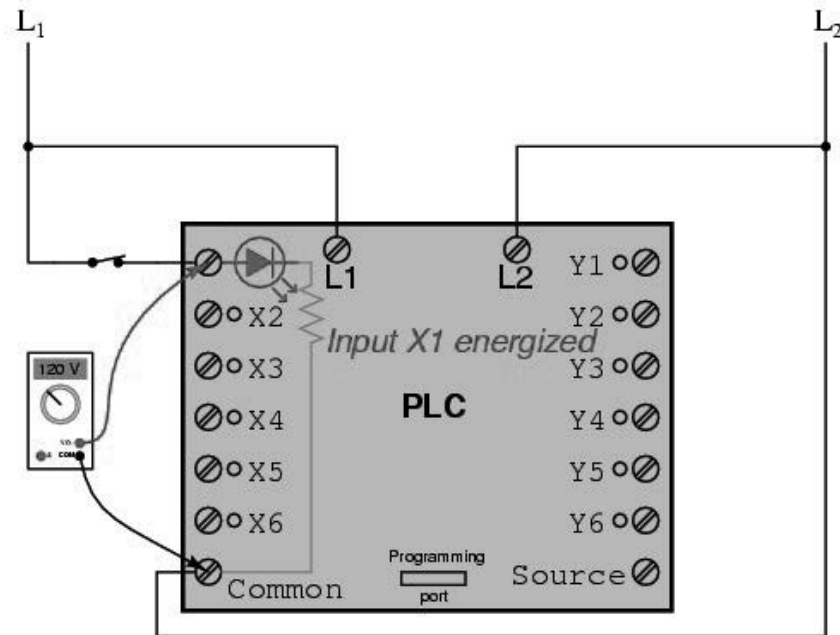
**NOTES**

The Figure 3.12 shows how a digital input/output circuit is needed to control input/output operation. A microcontroller which accepts input values and sends output values controls the interfaces. The Front panel display provides lights such as green LED (busy) and red LED (error) for device programmer who writes input/output programming basic.



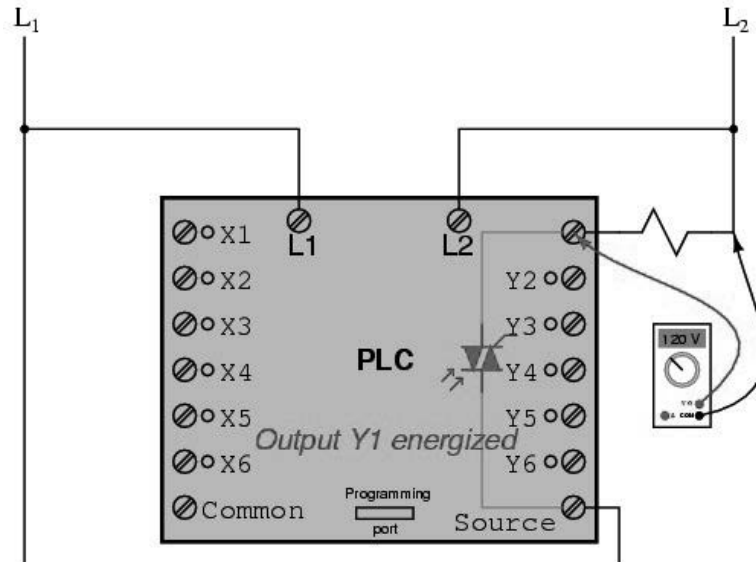
*Fig. 3.13 Input/Output Programming Flow Chart*

The Figure 3.13 shows that the input data for processing uses the standard input device which by default is a **keyboard** and then the processed data is sent for output to standard output device which by default is computer **screen**. In case the program execution encounters an error(s) then the error messages are also sent to standard output error default computer **screen**.



*Fig. 3.14 Input X1 is Put in Digital Circuit*

The Figure 3.14 shows the visual indication of energized input of an indicating LED on the front panel.



*Fig.3.15 Output Y1 is displayed in Digital Circuit*

The Figure 3.15 shows how the program controls the particular output operations. This is to be energized for the given input conditions in a digital circuit microcontroller. Though the program resembles to ladder logic diagram and is specified with switch and relay symbols, whereas actually there are no real switch contacts or relay coils that operate inside the PLC to create logical relationships between the input and output operations.

#### Check Your Progress

1. What do you understand by assembly language?
2. What are two type of assemblers?
3. What do you mean by repetition?
4. Write the rules of flow charting.
5. Define the term pseudocode.
6. Give the functioning of SCAL.

### 3.3 MICRO-PROGRAMMED CONTROL

A control unit with its binary control values stored as words in memory is called a micro-programmed control. Each word in the control memory contains a microinstruction that specifies one or more micro-operations for the system. A sequence of microinstructions constitutes a micro-program. The second one is often fixed at the time of system design and so is usually stored in ROM. Microprogramming involves placing some representation for combinations of values of control variables in terms of ROM for use by the rest of the control logic via successive read operations. The contents of a word in ROM at a given address specify the micro-operations to be performed for both the datapath and the control unit. A micro-program can also be stored in RAM. In this case, it is loaded initially

#### NOTES

## NOTES

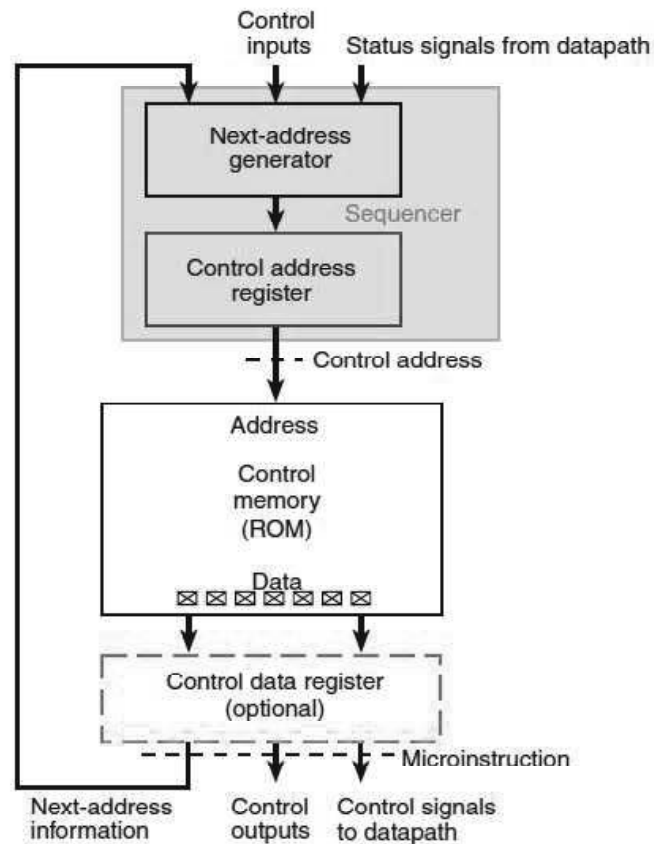
at system startup from the computer console or from some form of non-volatile storage, such as a magnetic disk. With either ROM or RAM, the memory in the control unit is used and hence called control memory. If RAM is used then the memory is referred to as writable control memory. The control memory is assumed to be a ROM within which all control information is permanently stored. The Control Address Register (CAR) specifies the address of the microinstruction. The advantages of micro-programmed control block are as follows:

- It is a flexible and structured design.
- Its testing sequences can be easily incorporated.
- It is easy to document and debug.

The following are the disadvantages of micro-programmed control block:

- It is expensive especially for small designs.
- It is slower than random logic.

Figure 3.16 shows the block diagram of micro-programmed control which uses a general configuration of a micro-programmed control.



*Fig. 3.16 Block Diagram of Micro-programmed Control*

The Control Data Register (CDR), which is optional, holds the microinstruction which is being executed both by the datapath and the control unit. One of the functions of the control word is to determine the address of the next microinstruction to be executed. This microinstruction may be the next one in sequence. Therefore, one or more bits that specify how to determine the address of the next



microinstruction must be present in the current microinstruction. The next address may also be a function of status and external control inputs. While a microinstruction is being executed, the next address generator produces the next address. This address is transferred to the CAR on the next clock pulse and is used to read the next microinstruction to be executed from ROM. Thus, the microinstructions contain bits for activating micro-operations in the datapath and bits that specify the sequence of microinstructions executed. The next address generator in combination with the CAR is sometimes called a micro-program sequencer, as it determines the sequence of instructions that is read from control memory. The address of the next microinstruction can be specified in several ways depending on the sequencer inputs. The CDR holds the present microinstruction while the next address is being computed and the next microinstruction is being read from memory. The CDR breaks up a long combinational delay path through the control memory and the datapath. Insertion of this register is a pipeline platform which allows the system to use a higher clock frequency and hence perform processing faster. The inclusion of a CDR in a system, however, complicates the sequencing of microinstructions particularly when decision-making based on status bits is involved. The ROM operates as a combinational circuit with the address as the input and the corresponding microinstruction as the output. The contents of the specified word in ROM remain on the output lines of the ROM as long as the address value is applied to the inputs.

## NOTES

### 3.3.1 Address Sequencing

Apart from execution of instructions, another important function of the microprogrammed control unit is to generate the address of the next sequence. The hardware that controls the address sequencing must be capable of sequencing the microinstructions within a routine and be able to branch from one routine to another.

While designing the microinstruction sequencing techniques, the two important concerns are:

- Minimizing the size of the control memory by minimizing the size of microinstruction.
- Executing microinstructions as fast as possible; since the maximum time is spent in generating address, the faster the address generation, the faster the execution of instruction.

When we power up the machine, the CAR needs to contain the address of the first microinstruction that is to be executed. In general, microinstructions are of two types: non-branching and branching. A non-branching microinstruction is one in which the next microinstruction to be executed is the one following the current microinstruction, i.e., the next instruction in sequential order. A branching microinstruction is the one where any desired control word can be executed next. There are three ways to determine the address of the next microinstruction to be executed.

- (i) **Next Sequential Address:** In the absence of other instruction, the control units increment the CAR content by one. However, this sequence of microinstructions is relatively small and lasts only for three or four microinstructions.

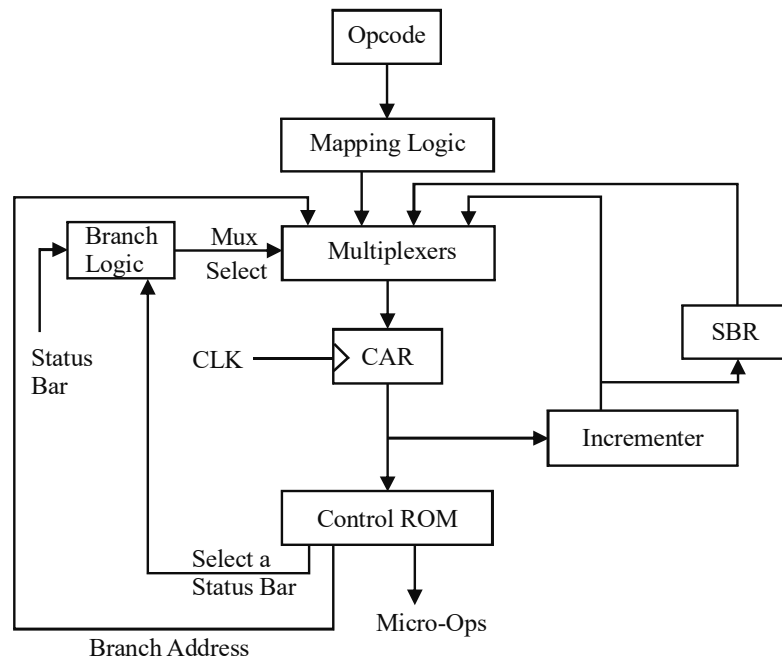
**NOTES**

(ii) **Branching:** In branching, the determination of the desired control memory address and its loading into the CAR are needed. Moreover, branching can be conditional or unconditional. In conditional branching, a condition is tested, which is determined by the status bit of ALU. Conditional branches are necessary in the microprogram as they are required to perform some sequences of microoperations only when certain situations or conditions are satisfied. Thus, if there are eight status conditions, then three bits are used to provide selection lines for multiplex.

In an unconditional branch, the microinstruction branch address from control memory is always loaded in CAR. This can be achieved by fixing the status bit such that the output of multiplexer is always one. One frequently used unconditional branch is subroutine call and return. Many routines have identical microinstruction sequences. If they are put into subroutines, routines become shorter. This saves memory. The microprogram that uses subroutine must have a provision for storing the return address during a subroutine call and restoring the address during subroutine return. Thus, for storing a return address, a stack is used.

(iii) **Interrupt Testing:** Certain microinstructions specify a test for interrupts. If an interrupt has occurred, then it determines the next microinstruction address.

In Figure 3.17, the block diagram of a control memory and hardware required for next address is drawn.



**Fig. 3.17** Selection of Address for Control Memory

Where:

**CAR:** It is used to store address of control memory from where instruction is to be fetched.

**Control ROM:** It is the Control Memory (CM) that holds the control words (CW).

**Opcode:** It is the machine instruction obtained from decoding instruction stored in IR.

**Mapping Logic:** It maps opcode to respective microinstruction address.

**Branch logic:** It determines the procedure which should be adopted to select the next CAR value among the several possibilities.

**Multiplexers:** They implement the choice of branch logic to obtain the next CAR value.

**Incrementer:** It generates  $CAR + 1$  as one possibility of the next CAR value.

**SBR:** It is used to hold return address for the operations of subroutine-call branch.

There are several ways to generate CAR values. One needs to select one from them. It is done by providing all possible values as input into a multiplexer and implementing the special branch logic (selecting appropriate select signal) which determines the next address and passes it to the CAR. For example, if there are four possible ways to determine the next address in a system, we will use an  $N \times 4 \times 1$  multiplexer, where  $N$  is the size of address of a control word (number of bits in the address). The branch logic determines the next address value out of the four possible 'next address values'. This next address value is passed to the CAR. The  $4 \times 1$  multiplexer has two select lines combinations which will decide the output, e.g., a 0 output in multiplexer indicates the transfer of the branch address to CAR, 1 causes the address register to be incremented by one, 2 causes the branching to subroutine, etc.

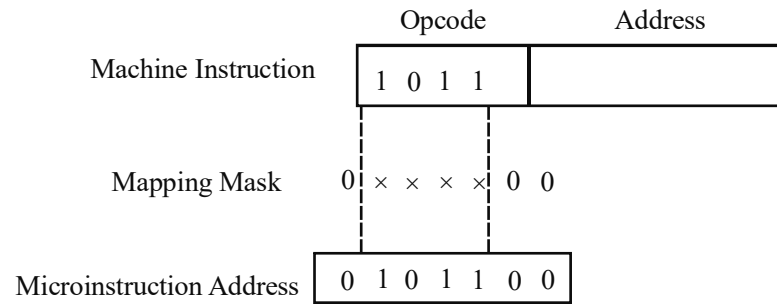
### 3.3.2 Micro-Program Example

The opcode mapping maps the bits of instruction to an address from control memory. For each opcode, there exists a microprogram routine in control memory. When the control memory is designed, the size of control word, say  $k$ , is determined by the length of the machine instruction routines (i.e., the length of the longest one, say  $n$ ) such that  $k = \ln(n)$ .  $k$  should be rounded off so that it is sufficient to implement any routine.

The first instruction of each routine will be located in the control memory at multiples of this length. Let's say this length is  $N$ . The first routine is at 0, the second at  $N$ , the third at  $2 \times N$ , and so on. With this technique, the mapping can be easily accomplished. In order to understand this concept, consider an example where there are 16 possible operations resulting in a four-bit opcode and each opcode is handled by a routine having a length of maximum four microinstructions thus require two bit. The generation of the microinstruction address for an opcode is done by appending two zero bits to the opcode. For example, as shown in Figure 3.18, there is mapping between a control memory having 128 words, requiring 2 address bits to specify the microinstruction in the routine and 4 bits that determine the instruction among 16 different opcodes. It is considered that each microprogram requires four microinstructions.

## NOTES

**NOTES**



**Fig. 3.18** Mapping Instruction Code to Microinstruction Address in Control Memory

Here, 0 is placed in the most significant place. If routine needs more than four microinstructions, it can use control memory address 1000000 through 1111111.

At the beginning of each instruction cycle, the address is determined by the opcode mapping, which is mapping of opcodes to microinstruction addresses of control memory. The  $n$ -bit opcode value can be used as the ‘Address’ input of a  $2^n \times M$ ROM; the selected ‘Word’ in the ROM generates  $M$ -bit CAR address for the beginning of the routine required to implement that instruction. This technique can be extended to allow variable-length routines in the control memory. Mapping provides the flexibility in adding instructions in control memory. As in case of modification, the only thing needed is to update the mapping mask. The mapping function is sometimes implemented by means of an integrated circuit, called Programmable Logic Device (PLD), which uses AND gate and OR gate with internal electronic fuses. In such cases, the mapping is implemented in terms of Boolean expressions that are executed with PLD.

### 3.3.3 Design of Control Unit

Control unit in a computer does the job of managing various components of the computer. Various tasks performed by this unit are: reading and decoding of program instructions, converting them into a set of control signals that activates other components of the computer. In advanced computer system, control systems may modify the order of few instructions for bringing improvement in performance.

Program Counter (PC) is a key component that is common to all CPUs. It is a special register keeping track of location in memory for next instruction to be read from.

Functions of a control unit are being mentioned in brief as given below. This is a simplified description and steps may be performed in different order and even concurrently, that depends on type of CPU.

1. Reading code for next instruction as indicated by program counter
2. Decoding instruction into a set of signals
3. Incrementing program counter for pointing to next instruction
4. Reading data as required by instruction from cells in memory, location of which is stored within the instruction code
5. Providing data to a register or ALU

6. Instructing hardware for performing requested operation if instruction needs some specialized hardware or ALU for completing the operation
7. Writing results from ALU either to a register, to a memory location or may be to an output device
8. Jumping back to step 1.

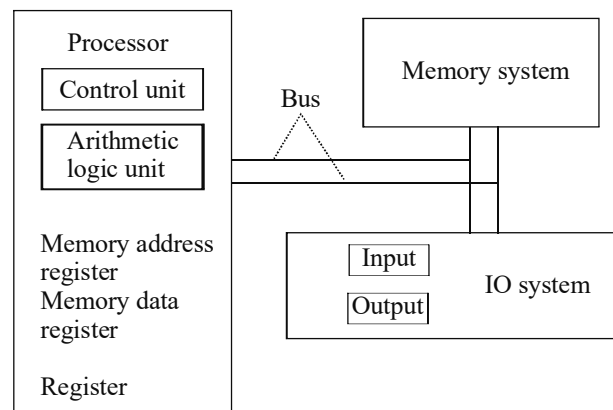
Conceptually program counter is just a set of memory cells that may be changed by computations made in ALU. If we add 20 to program counter, it causes reading of next instruction from 200 locations further down. Instructions modifying program counter are called 'jumps' and allow looping of instructions and execution of conditional instruction.

Thus, control unit is a very important component of the CPU performing following functions:

- data exchange between CPU and memory or I/O modules
- internal operations inside CPU, which are:
  - (i) Data transfer between registers (register transfer operations)
  - (ii) Instructing ALU to operate on data.
  - (iii) Regulation of other internal operations.

### Functional Requirements of a Control Unit

For defining functions of a control unit, resources and means to use these resources must be known. Position of control unit in the CPU and the computer system is shown in Figure 3.19.



*Fig. 3.19 Position of a Control Unit in CPU*

Thus, one must know:

- (a) Basic CPU components
- (b) Microoperation performed in CPU

CPU contains following basic functional components:

**Control Unit:** This unit controls all the operations inside CPU.

**ALU (Arithmetic & Logic Unit):** This performs basic operations, arithmetic and logical.

### NOTES

## NOTES

**Registers:** These are used for storing information within CPU. Memory Address Registers, Memory Data Register and other registers are there in the CPU.

Movements of data take two paths, internal and external.

**Internal Data Paths:** In these paths there is movement of data from one register to another or between ALU and a register.

**External Data Paths:** These data paths link CPU registers with the memory or I/O modules using system bus.

Control unit performs two basic operations:

- Execution of a microoperation
- Forming proper sequence of microoperations as per instruction to be executed

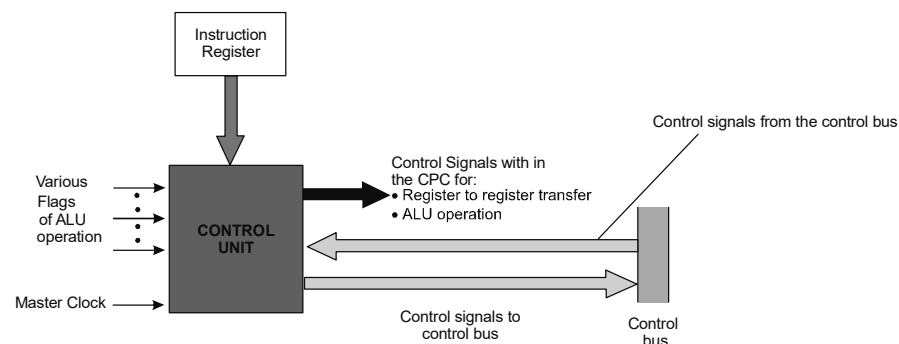
The microoperations can be classified as:

- Register to register data transfer
- Register to external interface data transfer using system bus
- External interface to register data transfer
- Arithmetic and logic operations using registers for input and output

Basic responsibility of control unit lies in controlling various CPU components for performing a specified sequence of microoperations for execution of an instruction.

### Structure of Control Unit

A control unit has a set of input values which is converted to an output control signal that performs microoperations. In the Figure 3.20 below a general model of a control unit is shown.



*Fig. 3.20 General model of Control Unit*

In the above model control unit has been shown as a black box containing certain inputs and outputs.

Inputs to the control unit are:

**Master Clock Signal:** This signal performs microoperations. In a single clock cycle a single or a set of simultaneous microoperations are performed. Some microoperations are performed in one processor cycle whereas others such as memory read, may need more than one processor cycle.

**The Instruction Register:** This register stores fetch instruction. In certain machines only op code is kept. This op code include addressing mode bits of the instruction and performs various cycles and keeps related microoperations to be performed.

**Flags:** Flags give the status of the CPU and outcomes of a previous operation on ALU are also flags. If zero flag is set control unit issues control signals that causes Program Counter (PC) to be incremented by 1.

**Control Signals from Control Bus:** Some control signals are sent to the control unit via control bus and these signals are issued from outside the CPU. Some signals are interrupt signals and acknowledgement signals.

Control units are implemented as hardwired control unit or microprogrammed control unit.

### Hardwired Control Unit

A hardwired control unit is implemented as combinatorial circuit in the hardware. Inputs to control unit consists of instruction register, timing signals, control bus signals and flags. Output signal sequences are generated on the basis of these inputs. This becomes highly complicated if there is large control unit making implementation of all combinatorial circuits very difficult. In such cases micro programmed control unit is used.

### Microprogrammed Control Unit

Hardwired control unit lacks flexibility in design. It becomes very difficult to design, test and implement if number of control lines is very high, usually in terms of hundreds. To overcome this, a programming approach is used which is known as microprogrammed control unit. Such a program consists of instructions that describe microoperations (one or more) for execution and the information on next microoperation for execution. Such instructions are known as microinstruction and program is called microprogram or firmware. The term firmware falls midway between hardware and software and is easier in comparison to hardware for designing but a bit more difficult that software.

Microprograms are usually stored in read only memory. This stored microprogram is also known as control memory. Such control memory is also made read-write type and in such case instruction set of a computer may be modified. A computer tailored for specific applications and having writable control memory is called 'dynamically microprogrammable' due to this reason. Computers having control units of this type have memories in two parts; a control memory and the main memory.

Implementation of microprogrammed control unit is done as a CPU inside the main CPU. It does the execution of microprogrammes stored in the control memory. Hardwired control units are faster than microprogrammed control units. They are better for RISCs (Reduced Instruction Set Computers) and not for CISCs (Complex Instruction Set Computers). For CISC microprogrammed controllers are better.

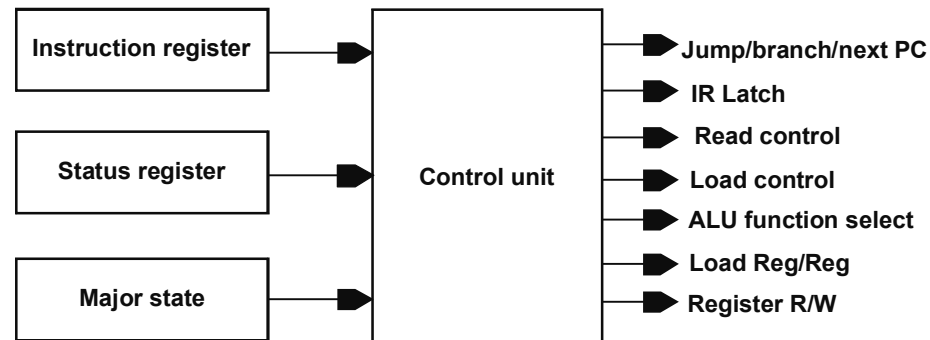
## NOTES

## Design of Control Unit

### NOTES

Control unit can be viewed as a Finite State Machine (FSM) with inputs as IR (Instruction Register), status register containing output status of ALU, with current major state that is contained in the operating cycle. Rules of such a FSM are encoded either in a Read-Only Memory (ROM), random logic, or a Programmable Logic Array (PLA). These are shown in the figure below.

Outputs for instruction/data path are Reg R/W, Load/Reg-Reg, ALU Function Select, Load Control, Read Control, IR Latch and Jump/Branch/NextPC.



‘ALU Function Select’ takes op code in IR (Instruction Register) translating to a function of ALU. This may be a compact binary code or one line per ALU function. The Jump/Branch/NextPC is dependent on instruction type which is, when used in architecture of Reduced Instruction Set Computers (RISC), are directly put in op code. In starting of an instruction cycle, there is read control. IR latch has its occurrence at end of fetch state. Load control comes at end of state of fetch data for load instruction. Load Reg/Reg has dependence over op code. Register R/W comes in the starting of data fetch stage as well as write back stage. Thus, it is dependent on instruction and major state.

CISC architecture makes use of control unit of more complex nature. IR has multiple words mostly and control unit looks at different portions of IR during execution stages.

In RISC architecture, access to registers is made uniformly as a block and hence register file in a particular register can be selected by using a simple decoder. But in CISC architecture, restrictions are put on some particular registers that is used by a particular instruction, by control unit.

We start designing a control unit and for this each control signal is listed in processor’s instruction/data path. Instruction register, status information and a ‘major state’ are inputs. Major state simply keeps track of position related to the execution of an instruction. An instruction always starts with state 0, meaning ‘Fetch’. At that state, output is given by the control unit as necessary signals for routing information of Program Counter (PC) to memory address port and then for making a selection for clock for the memory until there is response by fetching data from that location and subsequently latches this into the IR.

Decoding of an instruction in RISC architecture means decision on working of control unit for remainder of the instructions. If control unit is treated like finite state machine, bits of type field makes selection for ‘next state’ after decoding.



Talking as logic for a program, this is something like selection of branch as is done for Switch-case type statement in which every branch of 'switch' has sequence of steps that is performed for an instruction type. When a jump instruction is decoded, control unit produces output signals combining address part of instruction with the upper bits of Program Counter (PC), loading result to PC. CU (Control Unit) now goes back to 'Fetch'. Thus, Jump has three major states known as Fetch, Decode and Complete. To load memory, CU at first transmits selected values of register containing address, to memory's address port through a multiplexer and sends signals to memory for fetching that location. On return of value by memory, CU transmits signals to multiplexer(s) as well as register file such that memory data moves to destination bus and stored in the designated register. There are four major states for 'load' which are Fetch, Decode, Memory and Write Back.

Thus, for every instruction and every major state, it is required to see the list of control signals for deciding on value that each signal has. This may be thought as a large table of two dimensions that is indexed by type of instruction and major state.

One last bit of control output is for control of major state. This is input to CU as register which is shown above receiving its next value on each clock from CU. Jump begins from 'fetch' state to 'decode' state and then towards 'complete' state and again returning to 'fetch' state. State 0 is fetch state, state 1 is decode state, complete is state 3 and a 'load' is state 4. There are designs where state register does encoding of the instruction type. This way, it refers to various states a Finite State Machine (FSM) acquires instead of important instruction steps. For example, states of FSM for a Load may have a sequence 0, 1, 11, 12. In some other designs, we may find Jump passing through states 0, 1, 2, and Load passing through 0, 1, 2, 3, and type field distinguishes different behavior of the latter states. This is like naming same things in different ways. The central requirement is input to CU that it needs to know about its task on the present clock and next to it. In design process to CU this means ensuring that one of the control signals in the list indicates 'next state' that is specified in every cell of this table.

## NOTES

### Check Your Progress

7. How can you define the tasks needed for execution of MCU?
8. What are the micro-instructions?
9. What are the two important concerns in designing microinstruction sequencing techniques?
10. Why is opcode mapping used?
11. When does the cache process start?

## 3.4 ANSWERS TO 'CHECK YOUR PROGRESS'

1. Assembly language was the first step in the evolution of programming languages. It used mnemonics (symbolic codes) to represent operation codes and strings of characters to represent addresses.

## NOTES

2. Assemblers are of two types, One-pass and multi-pass.
3. Iteration or Repetition means that the sequence of instructions is executed and repeated any number of times in a loop until the logical condition is true.
4. Rules of flow charting:
  - Use consistent methods in drawing a flow chart.
  - Use common and easily understandable words.
  - Use consistent words or names in the flow chart.
  - Avoid crossing flow lines in the flow chart.
  - Draw the flow chart from top to bottom and left to right.
  - Flow charts that exceed a page should be properly linked using connectors to the portions of the flow chart on different pages.
5. Pseudocode is a tool used for planning a computer program logic or method. 'Pseudo' means imitation, and 'Code' refers to the instructions written in a computer language.
6. Functioning of SCAL:
  - When a subroutine is invoked, the parameters are loaded onto the stack and a SCAL is executed.
  - SCAL loads P + 1 (the return address) onto the stack and branches to a relative address within the current code segment.
  - S relative addressing is used to reference all parameters. Since the top of stack changes constantly, the S relative addresses of the parameters also change constantly.
7. Tasks for the execution of MCU:
  - Microinstruction execution: This generates a control signal to execute the microinstruction.
  - Microinstruction sequencing: This provides the next microinstruction from the control memory.
8. Micro-instructions are low-level control instruction in which the machine instruction is used to generate the control signals. They are inputs to the hardwired control unit.
9. While designing the microinstruction sequencing techniques, the two important concerns are:
  - Minimizing the size of control memory by minimizing the size of Microinstruction
  - Executing microinstructions as fast as possible; since the maximum time is spent in generating address
10. The opcode mapping maps the bits of instruction to an address from control memory.
11. Cache process starts when a CPU with cache refers to a memory. This generates the address of the item needed and search is made in the cache.

---

## 3.5 SUMMARY

---

- Computer can understand only binary-based language.
- Machine language besides being cumbersome is tedious and time consuming for the programmer.
- Assembly language was the first step in the evolution of programming languages.
- Assembler converts programs written in assembly language to an object file containing machine readable code.
- A sequence of instructions is known as subroutines.
- Cache memory is located between main memory and CPU. It is a small memory as a high-speed RAM buffer.
- Control unit in a computer does the job of managing various components of the computer.
- A control unit with its binary control values stored as words in memory is called a micro-programmed control. Each word in the control memory contains a microinstruction that specifies one or more micro-operations for the system.
- The Control Data Register (CDR), which is optional, holds the microinstruction which is being executed both by the data path and the control unit. One of the functions of the control word is to determine the address of the next microinstruction to be executed.
- The hardware that controls the address sequencing must be capable of sequencing the microinstructions within a routine and be able to branch from one routine to another.
- Certain microinstructions specify a test for interrupts. If an interrupt has occurred, then it determines the next microinstruction address.
- Cache memory is located between main memory and CPU. It is a small memory as a high-speed RAM buffer.
- Control unit in a computer does the job of managing various components of the computer.

## NOTES

---

## 3.6 KEY TERMS

---

- **Macroprocessor:** A macroprocessor controls repetitious writing of sequence.
- **Linker:** Linker creates a link file containing binary codes corresponding to compound modules in the program.
- **Loader:** A loader puts programs into main memory and then makes preparations for execution.
- **Algorithm:** An algorithm is a rough writing of a program. It contains step-by-step instructions to solve a given problem.

## NOTES

- **Flow Chart:** The flow chart is a common method to define the logical steps of flow within the program.
- **Micro-Programmed Control Unit:** A control unit designed through a micro-program and in which the control signals are generated through the software.
- **Control Address Register (CAR):** Its function is to hold the address of control memory generated by microprogram sequencer.
- **Control ROM:** It is the Control Memory (CM) that holds the Control Words (CW).
- **Opcode:** It is the machine instruction obtained from decoding instruction Stored in IR.
- **Mapping Logic:** It maps opcode to respective microinstruction address.
- **Branch Logic:** It determines the procedure which should be adopted to select the next CAR value among the several possibilities.
- **Registers:** These are used for storing information within CPU. Memory address registers, memory data register and other registers are there in the CPU. Movements of data take two paths, internal and external.

---

### 3.7 SELF-ASSESSMENT QUESTIONS AND EXERCISES

---

#### Short-Answer Questions

1. What do you understand by machine language?
2. What does an assembler do?
3. Define the programming arithmetic.
4. What are the various types of micro programs?
5. What are the functions of a control unit?
6. What is a micro-programmed control?
7. State the function of the Control Data Register (CDR).
8. What steps are involved in the execution of a microcode in one clock pulse?

#### Long –Answer Questions

1. Explain assembly language and its uses.
2. Discuss the purpose served by program loops giving relevant examples?
3. Analyze the process of input-output programming giving relevant examples.
4. Write detailed notes on:
  - (a) Control Memory
  - (b) Address Sequencing
5. Explain the structure and design of control unit with the help of examples.

6. Explain the block diagram of micro-programmed control.
7. Discuss the advantages of a micro-programmed control unit.
8. Discuss the significance of address sequencing and Opcode mapping.

---

### **3.8 FURTHER READING**

---

- Tanenbaum, Andrew S. 1999. *Structured Computer Organization*, 4th Edition. New Jersey: Prentice-Hall Inc.
- Mano, M. Morris. 1993. *Computer System Architecture*, 3rd Edition. New Jersey: Prentice-Hall Inc.
- Bartee, Thomas C. 1985. *Digital Computer Fundamentals*. New York: McGraw-Hill.
- Mano, M. Morris. 1979. *Digital Logic and Computer Design*. New Delhi: Prentice-Hall of India.
- Leach, Donald P. and Albert Paul Malvino. 1994. *Digital Principles and Applications*. New York: McGraw-Hill.
- Mano, M. Morris. 2002. *Digital Design*. New Delhi: Prentice-Hall of India.
- Kumar, A. Anand. 2003. *Fundamentals of Digital Circuits*. New Delhi: Prentice-Hall of India.
- Stallings, William. 2007. *Computer Organisation and Architecture*. New Delhi: Prentice-Hall of India.

### **NOTES**



---

# UNIT 4 CPU, INPUT-OUTPUT AND MEMORY ORGANIZATIONS

---

*CPU, Input-Output and  
Memory Organizations*

## NOTES

### Structure

- 4.0 Introduction
- 4.1 Objectives
- 4.2 Central Processing Unit
  - 4.2.1 Fundamental Concepts
  - 4.2.2 Organization of Registers in Different Computers
- 4.3 General Register Organization
  - 4.3.1 Control Word
- 4.4 Stack Organization
  - 4.4.1 Register Stack
  - 4.4.2 Memory Stack
  - 4.4.3 Reverse Polish Notation
  - 4.4.4 Evaluation of Arithmetic Expression
- 4.5 Instruction Formats
  - 4.5.1 Addressing Modes
- 4.6 Data Transfer and Manipulation
  - 4.6.1 Data Manipulation Instructions
- 4.7 Micro-Programmed Control
  - 4.7.1 Execution of Complete Instruction
  - 4.7.2 Reduced Instruction Set Computer
- 4.8 Peripheral Device
  - 4.8.1 Input-Output Interface
  - 4.8.2 Asynchronous Data Transfer
  - 4.8.3 Mode of Transfer
  - 4.8.4 Priority Interrupt
  - 4.8.5 Direct Memory Access (DMA)
  - 4.8.6 Input/Output Processor
  - 4.8.7 Serial Communication
- 4.9 Memory Unit
  - 4.9.1 Types of Memory
  - 4.9.2 Flash Memory
  - 4.9.3 Associative Memory
  - 4.9.4 Cache Memory
  - 4.9.5 Interleaving
  - 4.9.6 Hit Rate and Miss Penalty
  - 4.9.7 Virtual Memory
- 4.10 Parallel Processing
  - 4.10.1 Overcoming Pipelining Conflicts
- 4.11 Flynn's Classification
  - 4.11.1 Array Processors
- 4.12 Answers to 'Check Your Progress'
- 4.13 Summary
- 4.14 Key Terms
- 4.15 Self-Assessment Questions and Exercises
- 4.16 Further Reading

---

## 4.0 INTRODUCTION

---

### NOTES

A Central Processing Unit (CPU), also called a central processor, main processor or just processor, is the electronic circuitry that executes instructions comprising a computer program which include the arithmetic and logical unit, and the control unit of a computer system. The main components of CPU are control unit, clock, registers, and arithmetic and logic unit. Data manipulation instructions perform operations on data and provide computational capabilities for the computer. To fetch a word of data from memory, the processor gives the address of the memory location in which data is stored on the address bus to activate the read operation.

In general register organization, stack organization, memory stack, and so on we learn the important concepts like instruction formats and addressing modes are explained with the help of examples and figures.

In instructions and addressing modes there are various ways of specifying address of the data to be operated on. These different ways of specifying data are called the addressing modes. Computer instructions include data transfer instructions, data manipulation instructions and program control instructions. The unit discusses the application of these instructions. In microprogrammed control. Each instruction is executed by a set of microoperations which is known as microinstructions. In microprogrammed organization, the control unit is implemented through programming. A hard disk is one of the important I/O devices and is most commonly used as a permanent storage device in any computer. The human-interactive devices can be further categorized as direct and indirect. Direct devices are those that interact with people. Indirect devices do not interact with users. These device are used where humans are not directly involved in accepting the input or producing the output such as a scanner or a printer.

Direct memory access is an important data transfer technique. In this, the data is moved between a peripheral device and the main memory without any direct intervention of the processor.

A computer memory is usually meant to refer to the semiconductor technology that is used to store information in electronic devices. Current primary computer memory makes use of integrated circuits consisting of silicon-based transistors.

Memory is used for storage and retrieval of instructions and data in a computer system. There are two main types of memory volatile and non-volatile. RAM and ROM are considered as the two prime types of computer memory system in which RAM supports high speed memory and is volatile in nature. ROM supports low speed memory and it is non-volatile. You will study about organization of CPU. Designing an instruction set is an important aspect of CPU organization. RISC is a type of microprocessor that is designed with limited number of



instructions. Another topic, array processors, is discussed in this unit. It is a processor that is intended to do calculations on a sized array of data. In the end, you will also learn about Flynn's classification of computers-based on the multiplicity of instruction streams and data streams in a computer system.

In this unit, you will study about the central processing unit, general register organization, instruction formats, addressing mode, data transfer and manipulation, micro programmed control, reduced instruction set computer, peripheral device, input-output interface, asynchronous data transfer, priority interrupt, direct memory access, input/output processor, serial communication, memory unit, parallel processing, overcoming pipelining conflicts, Flynn's classification and array processors.

## NOTES

---

### 4.1 OBJECTIVES

---

After going through this unit, you will be able to:

- Describe processing unit
- Explain the organization of registers in different computers and register transfers
- Understand the different forms of stack organization
- Know the concept of branching and addressing modes
- Elaborate on computer instructions
- Define microprogrammed control
- Explain Reduced Instruction Set Computer
- Understand peripheral devices and input-output devices
- State the enabling and disabling of interrupts
- Explain direct memory access
- Discuss about memory units
- Describe about the parallel processing
- Classify computers on the basis of Flynn's classification
- Discuss array processors in a computer and their types

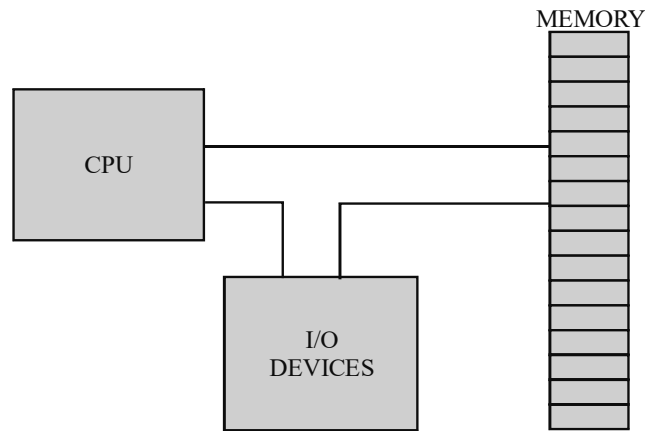
---

### 4.2 CENTRAL PROCESSING UNIT

---

CPU tests and manipulates data and transfers information to and from other components, such as working memory, disk drive, monitor and keyboard. Figure 4.1 shows the structure of a computer system which includes stack of memory and I/O devices.

## NOTES



*Fig. 4.1 Structure of a Computer System*

### **Central Processing Unit**

The Arithmetic and Logical Unit (ALU) and the Control Unit (CU) of a computer system are jointly known as the central processing unit. You may call CPU as the brain of any computer system. It takes all major decisions, makes all sorts of calculations and directs different parts of the computer functions by activating and controlling the operations.

For a computer to start running, it needs to have an initial program to run. This initial program, also known as bootstrap program, tends to be simple. It is stored in CPU registers. The role of the initial program or the bootstrap program is to load the operating system for the execution of the system. The operating system starts executing the first process, namely 'init' and waits for some event to occur. Event is known to occur by an interrupt from either the hardware or the software. Hardware can interrupt through system bus whereas software through system call.

When a CPU is interrupted, it immediately stops whatever it is doing and returns to a fixed location. This fixed location usually contains the starting address where the service routine for the interrupt is located.

### **Input/Output Structure**

There are various types of Input/Output (I/O) devices that are used for different types of applications. They are also known as peripheral devices because they surround the CPU and make a communication between computer and the outer world.

**Input Devices:** Input devices are necessary to convert our information or data into a form, which can be understood by the computer. A good input device should provide timely, accurate and useful data to the main memory of the computer for processing. Keyboard, mouse and scanner are the most useful input devices.

**Output Devices:** Visual Display Unit (VDU), terminals and printers are the most commonly used output devices.

## 4.2.1 Fundamental Concepts

The main components of the CPU are as follows:

- **Control Unit:** The basic role of CU is to decode/execute instructions. It generates the control/timing signals that trigger the arithmetic operations in ALU and also controls their execution.
- **Arithmetic and Logic Unit:** It is used for executing mathematical operations, such as \*, /, + and -; logical operations, such as, AND and OR; and shift operations, such as rotation of data held in data registers.
- **Clock:** There is a simple clock, a pulse generator, that helps to synchronize the CU operations so that the instructions are executed in proper time. A processor's speed is measured in hertz which is the speed of the computer's internal clock. The higher the hertz number, the faster is the processor.
- **Registers:** A CPU consists of several operational registers used for storing data that are required for the execution of instructions.

The design of CPU in modern form was first proposed by John von Neumann and his colleagues for the Institute for Advanced Studies (IAS) computer. The IAS computer had a minimal number of registers along with the essential circuits. This computer had a small set of instructions with each instruction having two parts: opcode and operand. It was allowed to contain only one operand address.

The simplest machine has one general purpose register, called Accumulator (AC), which is used for storing the input or output operand for ALU. ALU directly communicates with AC. Figure 4.2 shows the set of registers for a basic computer using a 16-bit instruction code. 4-bit are used to represent operation code and the other 12-bit holds address of memory location where we have considered that the system has a memory having capacity to store 4096 words such that each word has size of 16 bits.

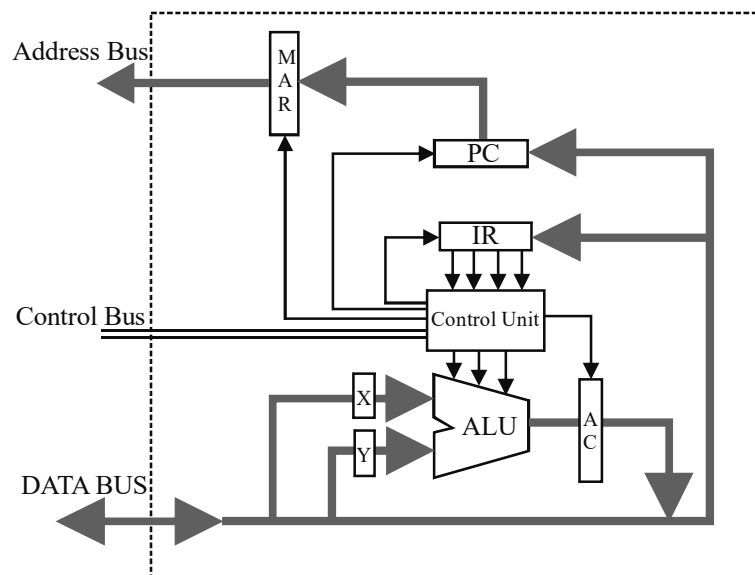


Fig. 4.2 General Organization of a Computer

## NOTES

## NOTES

A computer contains the following parts:

**Program Counter:** Program Counter (PC) contains the address of an instruction to be fetched. It has 12 bits as it also holds a memory address, i.e., the address of the next instruction. Programs are usually sequential in nature. The program counter is updated by the CPU after each instruction is fetched, pointing to the next instruction to be executed. But a branch or skip instruction will modify the contents of the PC to some other value.

**Instruction Register:** The instruction fetched from memory is stored in Instruction Register (IR) where the opcode and operand are analysed (operand can be data itself or it can be the address of memory location which store data) and accordingly, control signals are generated by the control unit for the execution of instructions.

**Temporary Register:** Temporary Register (TR) is used for storing the temporary data that is calculated during the processing.

**Accumulator:** It is a general purpose register which interacts with ALU and stores the results obtained from ALU. These results are transferred to the input or output registers.

**Data Register (DR):** It acts as buffer storage between the main memory and the CPU. It also stores the operand for the instructions, such as ADD DR or  $AC = AC + DR$ . In other words, contents of AC and DR are added by ALU and the results are stored in the accumulator. Thus, data register can also store one of the input operands.

**Memory Address Register:** It is used to provide address of memory location from where data is to be retrieved or to which data is to be stored. Memory Address Register (MAR) has 12-bits as it stores the memory address which is of 12-bit in size.

AR and DR play an important role in the transfer of data between CPU and the memory, i.e., they act as a buffer when the processor wishes to copy information from a register to primary storage, or read information from primary storage to a register. In the computer systems that use a common bus system, AR is directly connected to address bus, while DR is connected to data bus. DR is used for interchanging the data among several other registers.

**Input Register:** INPut Register (INPR) is used for storing input received from input device.

**Output Register:** OUTput Register (OUTR) is used for storing output to be transferred to output device.

The input register and output register only need to be 8 bits since they store 8-bit characters.

### 4.2.2 Organization of Registers in Different Computers

How the various components of control registers are connected to one another and how they communicate data among themselves is shown in Figure 4.3. From a user's point of view, the register set can be classified under the following two basic categories: (i) Programmer Visible Registers (ii) Status and Control Registers.

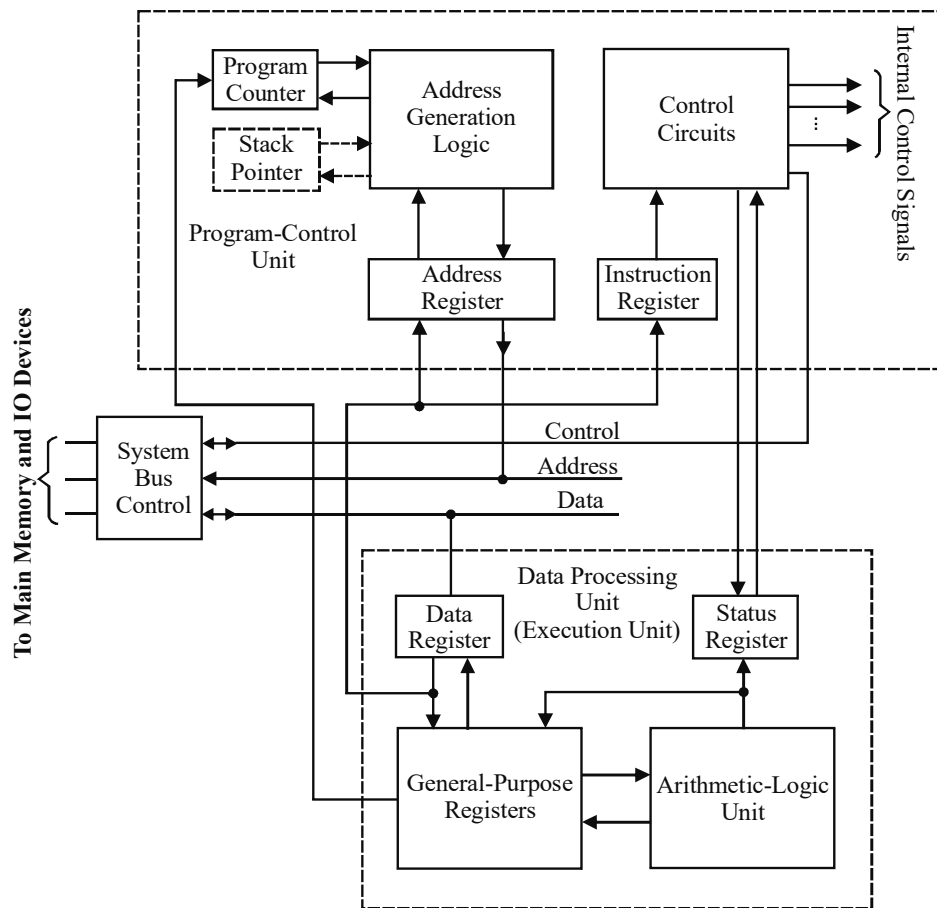


Fig. 4.3 Register Level CPU Organization

**NOTES**

**(i) Programmer Visible Registers**

These registers can be used by machine or assembly language programmers to hold all temporary data to minimize the reference to main memory. Virtually all CPU designs provide for a number of user visible registers unlike a single accumulator, as proposed for IAS computer.

Programmer visible registers can be accessed using machine language. Following are the various types of programmer visible registers:

- **General Purpose Register:** The general purpose registers are used for various functions as required by the programmer. A true general purpose register can contain operand for any opcode address or can be used for the calculation of address operand for any operation code of an instruction. But today's trend favours machines having dedicated registers. For example, some registers may be dedicated to floating point operations. In some cases, general purpose registers can be used for addressing functions, for example register indirect, displacement, etc. In other cases, there is a partial or clear separation between data register and address register.
- **Data Register:** The data registers are used for storing intermediate results or data. They cannot be used for the calculation of operand address.

## NOTES

- **Address Register:** An address register is a general purpose register but in some computers, the dedicated address registers are also used. Examples of the dedicated address registers are as follows:

- o **Segment Pointer:** In a machine with segmented addressing, a segment register holds the address of the base of the segment in the memory. There may be multiple registers, for example one for the operating system and one for the current process and they may be auto indexed.
- o **Index Registers:** These are used for index addressing scheme and may be auto indexed.
- o **Stack Pointer:** When the programmer visible stack addressing is used, the stack is typically in memory and a dedicated register, called stack pointer, is used which points to the top of the stack. This allows implicit addressing, i.e., push, pop and other stack instructions need not contain an explicit stack operand.

One of the key operations where the programmer usable register is used is when a subroutine call is issued. On a subroutine call, all temporary data stored in these registers are stored back in main memory by the call statement and are restored on encountering a return statement from the subroutine. This operation is automatic in most machines. Yet, in certain machines, this is done by the programmers. Similarly, while writing an interrupt service routine, it is required to save some or all programmer usable registers. In this simple project, the use of a stack pointer could be too excessive and complex to be realized. Hence, a stack pointer is exclusively used for executing the subprograms.

### (ii) Status and Control Registers

These registers cannot be used by the programmers. However, they are used by the control unit to control the operation of the CPU and by the operating system programs to control the execution of programs. The control registers hold information used for the control of the various operations. These registers cannot be used in data manipulations. However, the contents of some of these registers can be used by the programmer. Most of them are not visible to the user. Only a few of them may be visible which are executed in a control or operating system mode. The various control and status registers that are essential for the execution of instructions are as follows:

- **Program Counter:** PC is a register that holds the address of the next instruction to be read from memory. The PC increments after each instruction is executed and causes the computer to read the next instruction of program which is stored sequentially in the main memory. In case of a branch instruction, the address part is transferred to PC to become the address of the next instruction. To read an instruction, the content of PC is taken as the address for memory and a memory read cycle is initiated. PC is then incremented by one. So, it holds the address of the next instruction in sequence. Number of bits in the PC is equivalent to the width of a memory address.

- **Instruction Register:** IR is used to hold the opcode of instruction that is most recently fetched from memory.
- **Status Register:** Almost all the CPUs have a status register (also called flag register or processor status word), a part of which may be programmer visible. A register which may be formed by condition codes is called a condition code register and stores the information obtained from execution of the previous condition.

Some of the commonly used flags or condition codes stored in such a register are as follows:

- **Sign Flag:** Sign bit will be set according to the sign of previous arithmetic operation, whether it is positive (0) or negative (1).
- **Zero Flag:** Flag bit will be set if the result of the last arithmetic operation was zero.
- **Carry Flag:** Carry bit will be set if there is a carry result from the addition of the highest order bits or a borrow is taken from subtraction of highest order bit.
- **Equal Flag:** This bit flag will be set if a logic comparison operation finds out that both of its operands are equal.
- **Overflow Flag :** This flag is used to indicate the condition of arithmetic overflow.
- **Interrupt Enable/Disable Flag:** This flag is used for enabling or disabling interrupts.
- **Supervision Flag:** This flag is used in certain computers to determine whether the CPU is executing in supervisor mode or user mode. It is important as certain privileged instructions can be executed only in supervisor mode and certain areas of memory can be accessed only in supervisor mode.

In most CPUs, on encountering a subroutine call or interrupt handling routine, it is desired that the status information, such as conditional codes and other register information, be stored so that it can be restored once that subroutine is over. The register that stores condition code and other status information is known as Program Status Word (PSW). Along with PSW, a computer can have several other status and control registers, such as interrupt vector register in the machines using vectored interrupt, stack pointer if a stack is used to implement subroutine calls, etc. The design of status and control register also depends on the operating system support. Hence, it is always advisable to design register organization based on the principles of operating system as there is some control information that only of specific use to the operating system and hence depends on the operating system that we are using. In some machines, processor itself coordinates the subroutine call which will result in the automatic saving of all user visible registers and restoring them back on return. This allows each subroutine to use the user visible registers independently. While in other machines, it is the responsibility of the programmer to save the contents of the relevant user visible registers prior to a subroutine call.

## NOTES

## NOTES

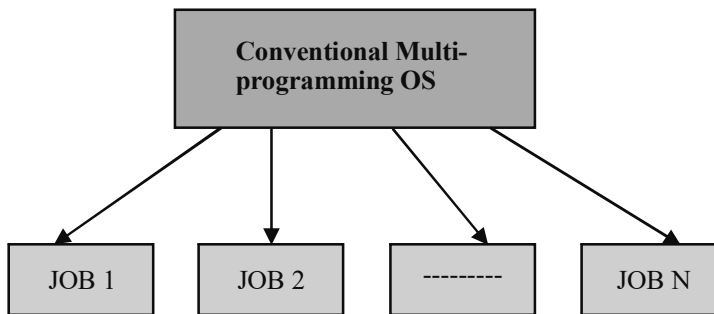
Thus, in second case we must include instructions that can implement the saving of the data in the program.

However there is not a clear separation of registers into these two categories. For example, on some machines, the program counter is user visible, for example Virtual Address eXtension (VAX), while it is not so in case of other machines.

### Virtual Machine Concept

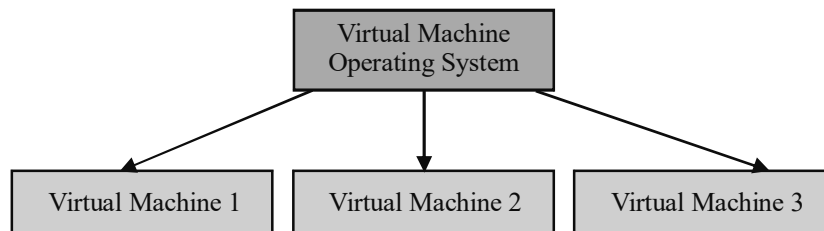
The operating system provides applications with a virtual machine. This type of situation is analogous to the communication line of a telephony company, which enables separate and isolated conversations over the same wire(s). An important aspect of such a system is that the user can run an operating system of his/her choice.

The virtual machine concept can be well understood by understanding the difference between conventional multiprogramming and virtual machine multiprogramming. In conventional multiprogramming, processes are allocated a portion of the real machine resources, i.e., a resource from the same machine is distributed among several resources (Refer Figure 4.4).



*Fig. 4.4 Conventional Multiprogramming*

In the virtual machine multiprogramming system, a single machine gives an illusion of many virtual machines, each of them having its own virtual processor and storage space which can be handled through process scheduling. Figure 4.5 shows the structure of virtual machine multi-programming which involves a number of virtual machines, such as Virtual machine 1, Virtual machine 2, Virtual machine 3. All machines are controlled by virtual machine operating system.



*Fig. 4.5 Virtual Machine Multiprogramming*

Following are the advantages of virtual machine multiprogramming:

- Each user is allocated with a machine which eliminates mutual interference between users.



- A user can select an Operating System (OS) of his/her choice for executing his/her virtual machine. Hence, the user can simultaneously use different operating systems on the same computer system.

### Kernel Approach

Following are the features of kernel:

- Kernel lies below system call interface and above the physical hardware.
- It provides large number of functions, such as CPU scheduling, memory management, I/O management, synchronization of processes, inter-process communication and other operating system functions.

### NOTES

## 4.3 GENERAL REGISTER ORGANIZATION

A bus organization for seven CPU registers is shown in Figure 4.6. The output of each register is connected to two Multiplexers (MUX) to form the two buses *A* and *B*. The selection lines in each multiplexer select one register or input data for the particular bus. Buses *A* and *B* form the inputs to a common ALU. The operation selected in the ALU determines the arithmetic or logic micro-operations to be performed. The result of the micro-operation is available for the output data and also goes into the inputs of these seven registers. The decoder selects the register that receives the information from the output bus. The decoder activates one of the register load inputs, thus providing a transfer path between the output data bus and the inputs of the selected destination register.

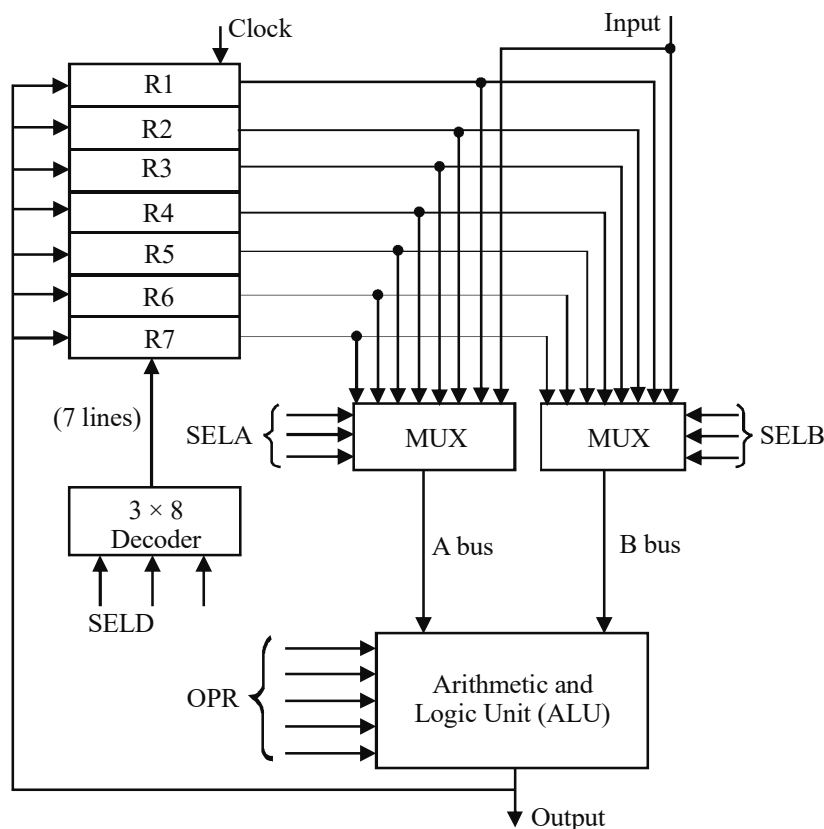


Fig. 4.6 General Register Organization

Let the operation be

$$R1 \leftarrow R2 + R3$$

To perform this operation, the control must provide the following:

**NOTES**

- SELA: Place the contents of  $R2$  into bus  $A$
- SELB: Place the contents of  $R3$  into bus  $B$
- ALU operation selector OPR: Provide the arithmetic addition  $A + B$
- SELD: Transfer the contents of the output bus  $R1$

At the beginning of a clock cycle, the four control selection variables generated by  $R2$  and  $R3$  must be available in the control unit. Two source registers propagate through the multiplexers and the ALU to the output bus and the input of the destination register during the clock cycle interval. At the next clock transition, information from the output bus is transferred to the destination register  $R1$ .

**4.3.1 Control Word**

The group of binary bits assigned to perform a specified operation is known as control word.

There are 14 binary selection inputs in the units, and their combined value specifies a control word. It consists of four fields as shown in Figure 4.7.



*Fig. 4.7 Control Word*

Three fields contain three bits each; one field has five bits. The three bits of SELA select a source register for input  $A$  of the ALU. The three bits of SELB select a register for input  $B$  of the ALU. The three bits of SELD select a destination register using the decoder and its seven load outputs. The five bits of OPR select one of the operations in the ALU.

The 14-bit control word, when applied to the selection inputs, specifies particular micro-operations. The encoding of register selection fields is specified in Table 4.1.

**Table 4.1** Encoding of Register Selection Fields

<i>Binary Code</i>	<i>SELA</i>	<i>SELB</i>	<i>SELD</i>
000	input	input	none
001	R1	R1	R1
010	R2	R2	R2
011	R3	R3	R3
100	R4	R4	R4
101	R5	R5	R5
110	R6	R6	R6
111	R7	R7	R7

When the 3-bit binary code for SELA or SELB is 000, the respective multiplexer selects the external input data as shown in Figure 4.7, and when the 3-bit binary code for SELD = 000, no destination register is selected and the content of the output bus is for the external output.

The OPR field has five bits. The encoding for the 5-bit OPR field is specified in Table 4.2.

*Table 4.2 Encoding of ALU Operation*

<i>OPR</i>	<i>Operation</i>	<i>Symbol</i>
00000	Transfer A	TSFA
00001	Increment A	INCA
00010	Addition	ADD
00101	Subtract	SUB
00110	Decrement A	DECA
01000	AND A and B	AND
01010	OR A and B	OR
01100	XOR A and B	XOR
01110	Complement A	COMA
10000	Shift Right A	SHRA
11000	Shift Left A	SHLA

**NOTES**

Let the micro-operation given by the statement be

$$R1 \leftarrow R4 \wedge R5$$

This statement specifies *R4* for input *A* of the ALU, *R5* for input *B* of the ALU, and *R1* as the destination register. The micro-operation to be performed is the AND operation between *R4* and *R5*. The control word for the above statement according to Tables 4.1 and 4.2 is as follows:

SELA	SELB	SELD	OPR
R1 001	R4 100	R5 101	AND 01000

Thus, the control word is 001 100 101 01000.

---

## 4.4 STACK ORGANIZATION

---

A stack is an ordered collection of items which permits the insertion or deletion of an item to occur only at one end. The insertion operation is known as push and the deletion operation is known as pop.

A stack is also known as a Last-In-First-Out (LIFO) list. The stack can be considered as a storage method in which the items stored last are the first items to be removed. The most common example of the stack phenomenon is a pile of

## NOTES

trays in a cafeteria. A tray which is placed last on top of the pile is the first to be taken off.

A stack in a digital computer is a part of the memory unit. Also associated with the stack is an address register. The latter keeps the address of the last element held in the stack. This address register is known as the *Stack Pointer*.

### Push and Pop Operations

Insertion and deletion of items are operations related with the stack. The process of inserting an item into stack is known as a *push operation*. The process of deleting an item from a stack is known as a *pop operation*. These operations are done by incrementing or decrementing the stack pointer.

#### 4.4.1 Register Stack

A stack can be organized by a finite number of registers or a stack can be a finite number of memory words. The stack pointer contains the address of the word that is currently on top of the stack, and which is a binary value. A 32-word register stack is shown in Figure 4.8. Currently, there are four items X1, X2, X3 and X4 in the stack with X4 at the top, so the content of the stack pointer is 4. Items are removed from the stack by using the pop operation. If we remove the top item X4 from the stack, X3 is then on the top of the stack and the content of SP is now holds the address 3. To insert a new item, first the SP is incremented and then the item is inserted so that the SP points to the top of the stack.

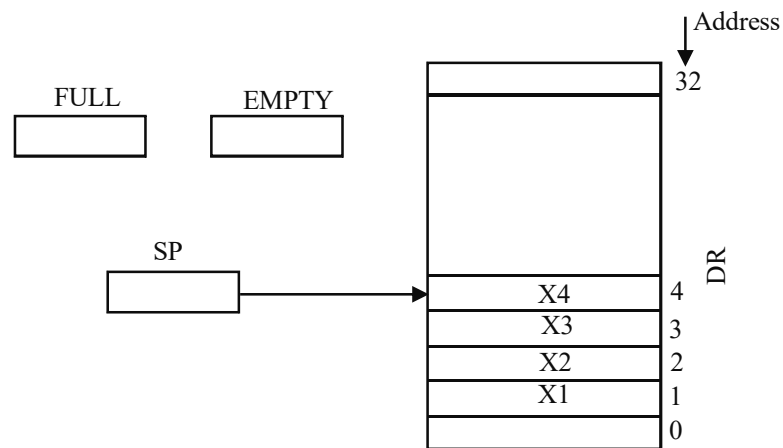


Fig. 4.8 Register Stack

In a 32-word register stack, the address of each location will be of five bits since  $2^5 = 32$ . Thus, the stack pointer will be of five bits and cannot exceed the value 11111. When the SP content is 11111, the one-bit register FULL is set to 1, indicating that the stack is full and there is no empty location for any more items. Similarly, when the content of SP is 00000, another one-bit register EMPTY is set to 1 indicating that the stack is empty and there is no element in the stack that can be deleted. The data register DR holds the item that is to be written into the stack or read out of the stack.

Initially, the SP is cleared to 0 so that the stack pointer points to the word at address 0. Also, the one-bit register FULL is cleared to 0, indicating that the stack

is not full and the register EMPTY is set to 1. A new item is inserted into the stack by the push operation. The operation set of the following micro-operations:

- |  |                              |
|--|------------------------------|
| $SP \leftarrow SP + 1$                   | Increment stack pointer      |
| $M[SP] \leftarrow DR$                    | Add item on top of the stack |
| If $(SP = 0)$ then $(FULL \leftarrow 1)$ | Check if the stack is full   |
| $EMPTY \leftarrow 0$                     | Mark the stack as not empty  |

If the stack is not empty, an item can be deleted from the stack using the pop operation. This operation is implemented by the following set of micro-operations:

- |   |                                     |
|---|-------------------------------------|
| $DR \leftarrow M [SP]$                    | Read item from the top of the stack |
| $SP \leftarrow SP - 1$                    | Decrement the stack pointer         |
| If $(SP = 0)$ then $(EMPTY \leftarrow 1)$ | Check if the stack is empty         |
| $FULL \leftarrow 0$                       | Mark the stack as not full          |

The top item is read from the stack into DR, and then the SP is decremented by 1 so that it points to the top of the stack. The SP is checked whether it is zero or not. If it is zero, EMPTY is set to 1, indicating that the stack is empty.

## NOTES

### 4.4.2 Memory Stack

A stack can also be implemented using the random access memory attached to the CPU. This can be implemented by assigning a portion of the memory for the stack operation, using the processor register as a stack pointer. The computer memory is partitioned into three parts—program, data and stack as shown in Figure 4.9. The program counter indicates the address of the subsequent instruction stored in memory and the stack pointer indicates to the top of the stack.

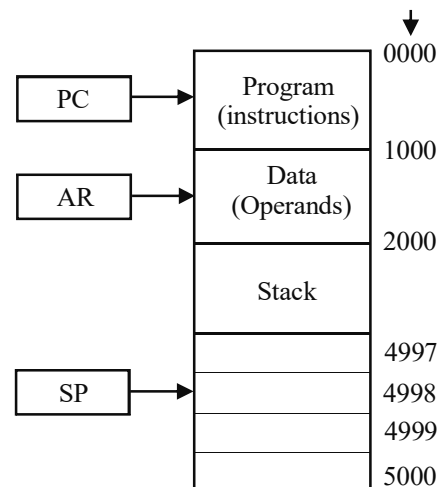


Fig. 4.9 Computer Memory divided into Program, Data and Stack Segments

The initial value of SP is 5000 and the first item stored in the stack is at address 4999; the second item is at address 4998, and so on. The last address that can be used in the stack is 2000, i.e., the final value of the stack is 2000. The stack grows in the reverse order with decreasing addresses. A new item is inserted into the stack using the push operation as follows:

## NOTES

$$SP \leftarrow SP - 1$$

$$M [SP] \leftarrow DR$$

The stack pointer is first decremented so that it points to the next address of the stack and then the item from the data register is inserted into the top of the stack. An item can be deleted from the stack using the pop operation as follows:

$$DR \leftarrow M [SP]$$

$$SP \leftarrow SP + 1$$

The top most item of the stack is read into the data register DR and then the stack pointer is incremented by 1 so that it points to item at the top of the stack.

Most computers do not offer any method to check the overflow or underflow of a stack to determine whether the stack is full or empty. One possible method is to use two processor registers holding the addresses 2000 (upper limit) and 5000 (lower limit), respectively. Then, every time the stack pointer is compared, a push operation takes place with the upper-limit registers and pop operation takes place with the lower-limit register.

### 4.4.3 Reverse Polish Notation

Let us consider an expression  $x + y$ . The plus operator is placed in between the two operands  $x$  and  $y$ . Such a notation is known as an *infix notation*. If the operator is placed before the two operands as  $+xy$ , the notation is said to be a *prefix notation*, also known as a polish notation. If the operator is placed after the two operands as  $xy+$ , the notation is said to be a *postfix notation*, also known as a *reverse polish notation*. Thus, the three notations are:

$x + y$       Infix notation

$+ xy$       Prefix or polish notation

$xy +$       Postfix or reverse polish notation

The reverse polish notation is best suited for stack manipulation. The reverse polish notation for the expression  $A * B + C * D$  is  $AB * CD *$ .

#### Conversion to Reverse Polish Notation

The conversion of an expression from the infix form to the reverse polish form must be done according to the operational hierarchy that follows for the infix notation. First, perform all arithmetic operations inside the inner parentheses, then inside the outer parentheses, and then do then multiplication and division operations. The addition and subtraction operations are performed at the end.

**Example 4.1.** Convert the infix expression  $(x + y) * [z * (w + v) + s]$  into the reverse polish notation.

**Solution.** The two subexpressions  $(x + y)$  and  $(w + v)$  will be solved first. Thus, the postfix expression of these subexpressions will be  $xy+$  and  $wv+$ , respectively.

Now, in the square bracket  $z$  will be multiplied by  $(w + v)$ . Thus, the postfix of this multiplication is  $zwv+*$ .

This multiplication result is then added to  $s$ , which will result in  $zwv+*s+$ .

Finally,  $xy + zwv + *s + *$  will be multiplied together to get  
 $xy + zwv + *s + *$ .

The procedure is shown again as follows:

$$\begin{aligned} & (x + y) * [z * (w + v) + s] \\ &= xy + * [z * wv + + s] \\ &= xy + * [zwv + * + s] \\ &= xy + * zwv + *s + \\ &= xy + zwv + *s + * \end{aligned}$$

**Example 4.2.** Convert the infix notation  $A * B + A * (B * D + C * E)$  into the reverse polish notation.

**Solution.**  $A * B + A * (B * D + C * E)$

$$\begin{aligned} &= AB* + A * (BD* + CE*) \\ &= AB* + A * BD * CE * + \\ &= AB * + ABD * CE * + * \\ &= AB * ABD * CE * + * + \end{aligned}$$

#### 4.4.4 Evaluation of Arithmetic Expression

Consider an expression  $A * B + C * D$  in the infix notation. Its reverse polish notation is  $AB * CD * +$ . This postfix expression will be evaluated as follows:

Scan the expression from left to right. Whenever an operator is found, perform the operation with the two operands on the left side of the operator. Remove the operator and the two operands and replace them with the result of that operation. Continue in the same manner and repeat the procedure for every operator found until there are no more operators.

Thus, for the reverse polish notation  $AB * CD * +$ , first we find the operator  $*$  and the two operands to the left of  $*$ , *i.e.*,  $A$  and  $B$ . Thus, we perform  $A * B$  and replace  $A$ ,  $B$  and  $*$  by their product, and we get

$$(A * B) CD * +$$

The next operator is  $*$  and the two operands to the left of  $*$  are  $C$  and  $D$ . Thus, we perform  $C * D$  and replace  $C$ ,  $D$  and  $*$  by the product, which is

$$(A * B) (C * D) +$$

The next operator is  $+$  and the two operands to the left of  $+$  are the two products  $(A * B)$  and  $(C * D)$ ; thus, the result obtained is:

$$A * B + C * D$$

Any arithmetic expression can be evaluated using a stack in the following manner:

- Convert the given infix expression into its equivalent reverse polish notation
- Scan the expression from left to right
- While scanning, when operands are found, push them into the stack as they appear

#### NOTES

## NOTES

- When operators are found, pop the two topmost operands from the stack, perform the operation involving the operator and then push back the result into the stack
- Continue scanning the expression until there are no more operators
- Finally, the result of the expression will remain on the top of the stack

To illustrate this, consider the expression  $(A + B) * (C + D)$ . In the reverse polish notation, this expression is  $AB + CD + *$ . The relevant stack operation is shown in Figure 4.10.

The arrow ( $\rightarrow$ ) points to the top of the stack.

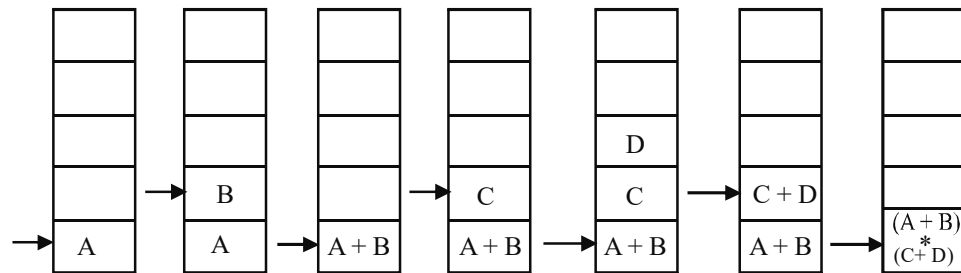


Fig. 4.10 Stack Operation to Evaluate  $(A + B) * (C + D)$

## 4.5 INSTRUCTION FORMATS

An *instruction* is a command given to a computer to perform a specified operation on some given data. The format in which the instruction is specified is known as instruction format.

The most common fields found in the instruction are as follows:

- An operation code field that signifies the operation to be done is known as opcode field.
- An address field that identifies the register address and/or a memory address.
- A mode field that identifies the manner in which the operands or the effective address is decided.

**For example:** ADD R1, R0. ADD is the op-code and R1, R0 are the address field.

Operations identified by computer instructions are run on some data kept in memory or some registers. Operands located on memory are identified by the memory address and operands placed on processor registers are recognized by register address. A register address is a binary number of K-bits that defines one of  $2^K$  registers in the CPU. Thus, if a CPU has processor registers R0 to R15, then the address of each register will be of four bits. For example, the binary information 0101 is the address of register R5.

The instructions may be of numerous different lengths consisting of different number of addresses. The number of address fields in the instruction format of a computer system is dependent on the internal architecture/organization of registers.

The different types of CPU organization are as follows:



- Single Accumulator Organization
- General Register Organization
- Stack Organization

### Accumulator Type Organization

All the operations are done with an indirect accumulator register. The instruction format uses one address field, i.e., only one operand address is specified in the instruction. The other operand is in the accumulator. The result is placed in the accumulator.

**For example:**       $\text{ADD } X, \quad \text{AC} \leftarrow \text{AC} + M[X]$

The  $\text{ADD } X$  instruction means add the contents at the memory location  $X$ , symbolized as  $M[X]$ , with the contents of the accumulator. Thus, the previous value of the accumulator will be lost and it will now contain the result of the above instruction.

### General Register Organization

In this type of computer, the instruction plan requires two or three addresses. The number of addresses in the instruction can be reduced to two from three if the destination register is the same as one of the source registers.

In a two-address instruction, both operand addresses are specified and the result is placed in one of the specified addresses. In a three-address instruction, two addresses are specified for the two operands and one address for the result. Thus, general register type computers employ two or three address fields in the instruction format. Each address field may indicate a processor register or a memory location.

**For example:**  $\text{ADD } R1, R2, R3 \quad R1 \leftarrow R2 + R3$

The above instruction contains three register addresses. The operation performed is the add operation between the content of processor registers  $R2$  and  $R3$  and the result is to be placed in the destination register  $R1$ .

$\text{ADD } R1, R2 \quad R1 \leftarrow R1 + R2$

The above instruction consists of only two register addresses.  $R1$  and  $R2$  are source registers where  $R1$  also serves as the destination register. The instruction specifies the add operation between the contents of  $R1$  and  $R2$  and the result to be stored into  $R1$ .

$\text{MOV } R1, R2 \quad R1 \leftarrow R2$

Mnemonic MOV is used to transfer instructions. The instruction contains only two register addresses  $R1$  and  $R2$ , where  $R2$  is the source register and  $R1$  is the destination. Thus, in a transfer type instruction, only two addresses are required. The instruction specifies moving the contents of  $R2$  into register  $R1$ .

$\text{Add } R1, X \quad R1 \leftarrow R1 + M[X]$

This instruction has two address fields,  $R1$  is the register address and  $X$  is a memory address.

### NOTES

## NOTES

### Stack Organization

Stack-oriented machines do not contain any accumulator or general-purpose registers. Computers with stack organization have push and pop instructions, which require an address field. Thus, the instruction

$$\text{PUSH } X \quad \text{TOP} \leftarrow M[X]$$

will push the word/data at address  $X$  to the top of the stack. The SP is automatically updated. The operation instruction does not contain any address field because the operation is performed on the two topmost operands of the stack.

#### For example: ADD

The instruction ADD consists of only an operation code with no address field. This instruction pops the top two operands from the stack, adds the numbers and then pushes the result into the stack.

To show how the number of addresses affects a computer program, the following arithmetic statement will be evaluated:

$$X = (A + B) * (C + D)$$

using three, two, one or zero address instructions.

The ADD, SUB, DIV and MUL mnemonics are used for arithmetic operations, and MOV is used for the transfer operation. LOAD and STORE mnemonics are used for transfers to and from the memory and Accumulator (AC). It can be assumed that the operands are in memory addresses  $A, B, C$  and  $D$ , and the result should be stored in memory address  $X$ .  $R1$  and  $R2$  are the registers and  $T$  is the temporary memory location used to store intermediate results.

#### Three-Address Instruction

$$\text{ADD } R1, A, B \quad R1 \leftarrow M[A] + M[B]$$
$$\text{ADD } R2, C, D \quad R2 \leftarrow M[C] + M[D]$$
$$\text{MUL } X, R1, R2 \quad X \leftarrow R1 * R2$$

The symbol  $M[A]$  indicates the operand at the memory address symbolized by  $A$ . The merit of the three-address format is that it results in short programs when evaluating arithmetic expressions.

#### Two-Address Instruction

$$\text{MOV } R1, A \quad R1 \leftarrow M[A]$$
$$\text{ADD } R1, B \quad R1 \leftarrow R1 + M[B]$$
$$\text{MOV } R2, C \quad R2 \leftarrow M[C]$$
$$\text{ADD } R2, D \quad R2 \leftarrow R2 + M[D]$$
$$\text{MUL } R1, R2 \quad R1 \leftarrow R1 * R2$$
$$\text{MOV } X, R1 \quad M[X] \leftarrow R1$$

#### One-Address Instruction

A one-address instruction uses an Accumulator (AC) register for all data manipulation.

LOAD $A$	$AC \leftarrow M[A]$
ADD $B$	$AC \leftarrow AC + M[B]$
STORE $T$	$M[T] \leftarrow AC$
LOAD $C$	$AC \leftarrow M[C]$
ADD $D$	$AC \leftarrow AC + M[D]$
MUL $T$	$AC \leftarrow AC * M[T]$
STORE $X$	$M[X] \leftarrow AC$

Amidst the AC register and a memory operand all operations are complete.  $T$  is the address of a temporary memory location used for storing intermediate outcome.

### Zero-Address Instruction

To evaluate an arithmetic expression for a zero-address machine, the expression must be in the reverse polish notation. Also, instructions like ADD and MUL do not require an operand field. They simply pop up the two topmost operands from the stack, perform the operation and place the result on top of the stack.

However, the push and pop instructions require an address field to indicate the operand that interacts with the stack. TOS means top of stack.

The reverse polish notation of the expression is as follows:

$$\begin{aligned}
 X &= (A + B) * (C + D) \text{ is evaluated as} \\
 &= (AB+) * (CD+) \\
 &= AB + CD + *
 \end{aligned}$$

PUSH $A$	$TOS \leftarrow A$
PUSH $B$	$TOS \leftarrow B$
ADD	$TOS \leftarrow A + B$
PUSH $C$	$TOS \leftarrow C$
PUSH $D$	$TOS \leftarrow D$
ADD	$TOS \leftarrow C + D$
MUL	$TOS \leftarrow (A + B) * (C + D)$
POP $X$	$M[X] \leftarrow TOS$

### 4.5.1 Addressing Modes

Addressing modes form the part of instruction set architecture. The instruction set is an important aspect of any computer organization. A simple ADD operation along with opcode must also provide the information about how to fetch the operands and where to put the result. Operands are commonly stored either in main memory or in the CPU registers. If operand is located in the main memory, the location address has to be given the instruction in the operand field. Thus, if memory addresses are 32 bits, a simple ADD instruction will require three 32 bits addresses in addition to opcode. The recent architecture provides a large number of registers so that compilers can keep local variables in registers, eliminating memory references. This results in a reduced program size and execution time.

### NOTES

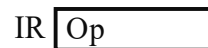
## NOTES

As it not possible to put all variables in registers, a memory reference is required. It attempts to refer a large range of locations in main memory or virtual memory. One possibility is that they contain the memory address of the operand but this will require large field to specify full memory address. Also, the address must be determined at compiling time. Other possibilities also exist which provide both shorter specifications and the ability to determine addresses dynamically. To achieve this objective, a variety of addressing techniques have been employed. These techniques trade off between address range and/or addressing flexibility on the one hand and the number of memory references and/or complexity of address calculation, on the other. Basically what an operand stores is the effective address. The Effective Address (EA) of an operand is the address of (or the pointer to) the main memory or register location in which the operand is contained, i.e., operand = EA. Each instruction of a computer specifies an operation on certain data. There are various ways of specifying address of the data to be operated on. These different ways of specifying the data are called the addressing modes. There are two ways by which the control unit determines the addressing mode used by an instruction:

- Opcode itself explicitly specifies the addressing mode used in the instruction.
- It uses a separate mode field which indicates that the addressing mode is being used in the instruction.

### Implied Mode

The operand is specified implicitly in the definition of the instruction as in the case of an Accumulator (AC). Only the accumulator holds the operand and a stack organization where the operand is the data stored on the top of stack. In both the cases, only one operand is available for manipulation. So, an instruction just tells us about the opcode and no field is required for operand, as shown in Figure 4.11.



*Fig. 4.11 Implied Addressing Mode*

### Immediate Addressing

Following is the type of implied mode:

Immediate addressing is the simplest form of addressing where the operand is actually present in instruction, i.e., there is no operand fetching activity as the operand is given explicitly in the instruction (Refer Figure 4.12). This mode can be used to define and use constants or set initial value variables. Examples of immediate addressing are as follows:

MOV 15, R1 (Load binary equivalent of 15 in register R1)

ADD 15, R1 (Add binary equivalent of 15 in R1 and store the result in R1)

ADD 5 (Add binary equivalent of 5 to contents of accumulator)

**Advantage:** The advantage of immediate addressing is that no memory reference other than fetching of the instruction is required. As no memory reference is required to obtain the operand, it has very small instruction cycle. Also, it is fast as memory

reference is reduced to one. It is commonly used to define and use constants or set initial values.

**Disadvantage:** The disadvantage of immediate addressing is that the size or the number of operations is the same as that of the address field which in most instruction sets, is small as compared to the word length. Further, it has a limited utility.



Fig. 4.12 Immediate Addressing Mode

**Absolute Mode**

In this mode, the operand’s address is explicitly given in the instruction. This address can be in either a register or in a memory location, i.e., the EA of the operand is given in the instruction (Refer Figure 4.13).

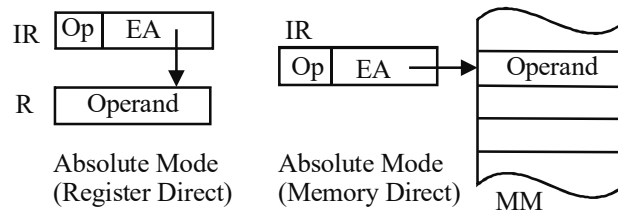


Fig. 4.13 Absolute Mode of Addressing

Absolute mode involves direct and indirect addressing modes which are discussed below:

**Direct Addressing**

The simplest addressing mode where an operand is fetched from memory is direct addressing. In direct addressing, the address field contains the effective address of the operand (Refer Figure 4.14).

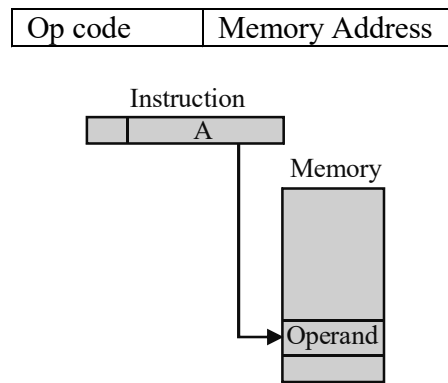


Fig. 4.14 Direct Addressing Mode

This technique was common in earlier generations of computers. It requires only one memory reference. As address field contains address of operand, no special calculation is required for calculating effective address.

$$EA = A.$$

**NOTES**

## NOTES

For example, ADD A

The value of operand is obtained from memory location whose address is A and is added to content of accumulator. The obvious limitation in this scheme is that it provides only a limited address space.

### Register Addressing

Register addressing is a way of direct addressing where the address field refers to a register rather than the main memory address (refer Figure 4.15).

$$EA = R$$

The address field should store the reference of register. As 8–32 general purpose registers can be referenced, we need 3–5 address bits. As the CPU registers are frequently used, register addressing is heavily used. There are a limited number of registers compared with the main memory locations. So, they must be used efficiently. It is up to the programmer to decide which values should remain in registers and which should be stored in main memory. Most modern CPUs employ multiple general purpose registers, placing the burden of efficient execution on the assembly language programmer, for example compiler writer. Thus, we should have good assembly programmer or compiler who avoids frequent data transfer from register to memory, leading to reduction in wastage of time in fetching data. So, if the operand in a register is used in multiple operations, it results in a real saving.

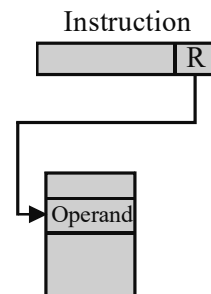


Fig. 4.15 Register Addressing Mode

**Advantages:** As there are only few registers in this mode, very small address field is needed when compared to memory access addressing modes, resulting in short instructions. Further, its execution is fast as no memory access is required.

**Disadvantages:** Address space is limited. Speed is achieved only when a good assembly programming or compiler writing is used.

### Indirect Mode

In this mode, the register or the main memory location holds the EA of the operand (Refer Figure 4.16). The location where the operand is stored is calculated from address given in the instruction.

$$EA = (A)$$

Indirect addressing mode is used where the address of the operand is contained in register pair. To implement such an instruction, first we look in A, then find address (A) and fetch operand from that address. For example, in instruction ADD (A), add contents of cell pointed to by contents of A (content of A is memory location) to accumulator.

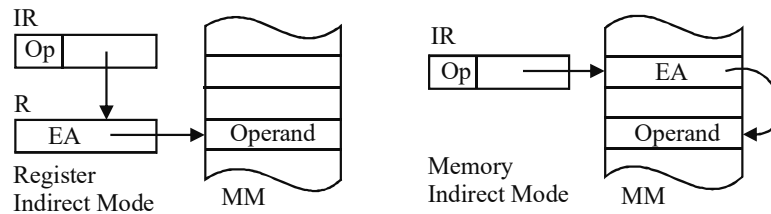


Fig. 4.16 Indirect Mode of Addressing

In the direct addressing mode, the length of the address field is usually less than the word length. Thus, there is a limited address range. To overcome this problem, one can use the address field that refers to the address of a word in memory which in turn, contains a full length address of the operand. The obvious advantage of this approach is that for a word of length  $N$ , an address space of  $2^N$  is available. Its disadvantage is that the instruction execution requires two memory references to fetch the operand: one to get its address and the other to get its value.

Although the number of words that can be addressed in this mode is equal to  $2^N$ , the number of different effective addresses that may be referenced at any one time is limited to  $2^K$ , where  $K$  is the length of the address field. In a virtual memory environment, all the effective address locations can be confined to page 0 of any process. Because the address field of an instruction is small, it will naturally produce the low numbered direct addresses which would appear in page 0. When a process is active, there will be repeated references to page 0, causing it to remain in main memory. Thus, an indirect memory reference may involve more than one page fault.

A rarely used variant of indirect addressing is multilevel or cascaded indirect addressing and is expressed as follows:

$$EA = (\dots(A)\dots)$$

In this case, one bit of a full word address is an indirect flag (I). If the I bit is 0, then the word contains EA. If the I bit is 1, then another level of indirection is invoked. There does not appear to be any particular advantage to this approach. However, its disadvantage is that three or more memory references could be required to fetch an operand in it. The multiple memory accesses to find an operand makes it slower.

Register indirect addressing and displacement addressing are the two types of indirect addressing mode which are discussed below:

### Register Indirect Addressing

Just as register addressing is analogous to direct addressing, register indirect addressing is analogous to indirect addressing. In both cases, the only difference is whether the address field refers to a memory location or to a register. Thus, for a register indirect address is as follows:

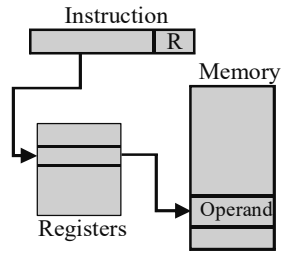
$$EA = (R)$$

In register indirect addressing mode, the operand field of an instruction holds the address of the address register to calculate the true address of the operand (Refer Figure 4.17). The advantages and disadvantages of register indirect addressing are basically the same as of indirect addressing. In both the cases, the address space limitation (limited range of address) of the address field is overcome by

## NOTES

referring that field to a word-length location containing an address. In addition, register indirect addressing uses one less memory reference than indirect addressing.

## NOTES



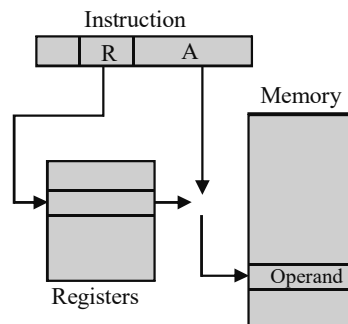
*Fig. 4.17 Register Indirect Addressing Mode*

## Displacement Addressing

Displacement addressing is a very powerful mode of addressing because it combines the capabilities of direct addressing and register indirect addressing. It is known by a variety of names depending on the context of its use. However, the basic mechanism is the same (to represent an expression. It is expressed in the following way.

$$EA = A + (R)$$

Displacement addressing requires that the instruction should have two address fields, in which at least one is explicit. The value contained in one address field is used directly, as in above case. The other address field can be an implicit reference based on opcode which refers to a register whose contents are added to A to produce the effective address (Refer Figure 4.18).



*Fig. 4.18 Displacement Addressing Techniques*

Now we will discuss the following three most common uses of displacement addressing:

- **Relative Addressing:** This is used to address an operand in the main memory whose address is specified in relation to the current instruction. Effective address is obtained by adding a constant, which here is the offset, that represents the displacement from current position to the location of the operand (Refer Figure 4.19). This offset can be the content of PC (the implicitly referenced register) and also can be negative. The constant is either explicitly given in the instruction by the assembly programmer, or is calculated by the assembler on the basis of the knowledge of the main



memory locations of the program and the desired operand. Relative addressing exploits the concepts of locality. If most memory references are relatively nearer to the instruction being executed, then the use of relative addressing saves address bits in the instruction.

**NOTES**

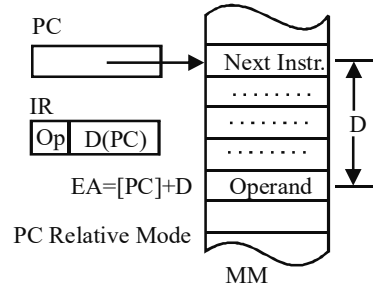


Fig. 4.19 Relative Addressing Mode

- Base Register Addressing:** The referenced register contains a memory address, while the address field contains a displacement usually an unsigned integer representation from that address. The register reference may be explicit or implicit. The base register addressing also exploits the locality of memory references. It is a convenient means of implementing segmentation, for example segment registers in  $80 \times 86$ . This can be implicit using a single segment base register or the programmer may choose a register to hold the base address of a segment, and the instruction must reference it explicitly. In the latter case, if the length of the address field is  $K$  and the number of possible registers is  $N$ , then one instruction can reference any one of  $N$  areas of  $2^K$  words.
- Indexing:** In index addressing mode, the address field references a main memory address and the reference register contains a positive displacement from that address. The EA of the operand is generated by adding a constant value (given in the instruction) to the content of a register (specified in the instruction). This is used to address elements of an array  $A = [A[1], A[2], \dots, A[n]]$ . The starting address of  $A$  is constant and the index  $i$  is contained in the register (Refer Figure 4.20). Element  $A[i]$  ( $i=1, 2, \dots, n$ ) can be addressed by this mode with different index  $i$ .

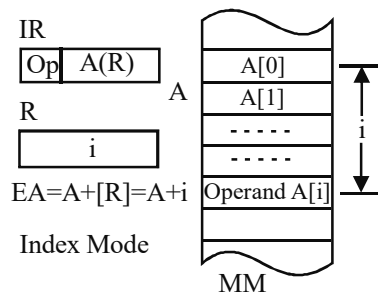


Fig. 4.20 Index Addressing Mode

It can be noticed that the usage of indexing is just the opposite of the interpretation for base register addressing. The address field is considered to be a

## NOTES

memory address in indexing, and it generally contains more bits than an address field in a comparable base-register instruction. There are some refinements to indexing that would not be as useful in the base register context. However, the method of calculating EA is the same for both base register addressing and indexing, and in both cases, the register reference can be explicit or implicit (depending on CPU types).

Indexing provides an efficient mechanism for performing iterative operations. Consider, for example a list of numbers stored in memory starting from location A. Suppose that we want to add 1 to each element on the list. For this, we need to fetch each value, add 1 to it, and store it back. The sequence of effective addresses that we need is A, A + 1, A+2, ....., up to the last location on the list. With indexing, this is easily done. The value A is stored in the instruction's address field, and the chosen register, called an index register, is initialized to 0. After each operation, the index register is incremented by 1. As increment or decrement of the index register is done after each reference, some systems will automatically do this as part of the same instruction cycle and in others it is done explicitly. This is known as auto indexing. If certain registers are devoted exclusively to indexing, then auto indexing can be invoked implicitly and automatically. If general purpose registers are used, the auto index operation may need to be signalled by a bit in the instruction. Auto-indexing using increment can be depicted as follows:

$$\begin{aligned}EA &= A + (R) \\(R) &\leftarrow (R) + 1\end{aligned}$$

In some machines, both indirect addressing and indexing are provided, and it is possible to employ both in the same instruction. The two possibilities: post indexing and pre-indexing are discussed below:

**Post Indexing:** If the indexing is performed after the indirection, it is termed as post indexing. It is represented in the following way:

$$EA = (A) + (R)$$

First, the contents of the address field are used to access a memory location containing a direct address. This address is then indexed by the register value. This technique is useful for accessing one of the various blocks of data of a fixed format. Thus, the addresses in the instructions that reference the block could point to a location (value = A) containing a variable pointer to the start of a process control blocks. The index register contains the displacement within the block.

**Pre-Indexing:** If the indexing is performed before the indirection, it is termed as pre-indexing. It is represented in the following way:

$$EA = [A + (R)]$$

An address is calculated as in simple indexing. In this case, the calculated address contains not the operand but the address of the operand. An example of the use of this technique is to construct a multi way branch table. At a particular point in a program, there may be a branch of the various locations depending on conditions. A table of addresses can be set up starting at location A. By indexing into this table, the required location can be found.

Typically, an instruction set will not include both pre-indexing and post indexing.

### Stack Addressing

The final addressing mode that we consider is stack addressing. A stack is a linear array of locations. It is sometimes referred to as a push down list or Last In First Out (LIFO) queue. It is a reserved block of locations. Items are appended to the top of the stack so that, at any given time, the block is partially filled. Associated with the stack is a pointer whose value is the address of the top of the stack. The stack pointer is maintained in a register. Thus, references to stack locations in memory are in fact register indirect addresses.

The stack mode of addressing is a form of implied addressing. The machine instructions need not include a memory reference but should implicitly operate on the top of the stack.

Another important issue is how to determine the addressing mode to be followed. Virtually all computer architectures provide more than one addressing mode. The question arises as to how the control unit can determine the address mode to be used in a particular instruction. Several approaches are taken. Often, different opcodes will use different addressing modes. Also, one or more bits in the instruction format can be used as a mode field. The value of the mode field determines which addressing mode is to be used. Table 4.3 summarizes the principal advantages and disadvantages of various addressing modes.

*Table 4.3 Various Addressing Modes*

Mode	Algorithm	Principal Advantage	Principal Disadvantage
Immediate	Operand = A	No memory reference	Limited operand magnitude
Direct	EA = A	Simple	Limited address space
Indirect	EA = (A)	Large address space	Multiple memory references
Register	EA = R	No memory reference	Limited address space
Register Indirect	EA = (R)	Large address space	Extra memory reference
Displacement	EA = A + (R)	Flexibility	Complexity
Stack	EA = top of stack	No memory reference	Limited capability

**Notation:**

- A = Contents of an address field in the instruction.
- R = Contents of an address field in the instruction that refers to a register.
- EA = Effective (actual) address of the location containing the referenced operand.
- (X) = Contents of location X.

## 4.6 DATA TRANSFER AND MANIPULATION

Data transfer instructions transfer data from one location to another without changing the data content. These transfers can be between two processor registers or between the memory location and processor registers or between the processor registers and input or output. The different data transfer instructions (with their mnemonics) used in various computers are listed in Table 4.4.

### NOTES

**Table 4.4** Data Transfer Instructions

<i>Name</i>	<i>Mnemonic</i>
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

**NOTES**

Different computers can use different mnemonics for the same instruction. The load instruction is used to transfer data content from the memory location to the processor register called accumulator. The store instruction is used to transfer data content from the processor register to some memory location. The move instruction is used to transfer data content from one processor register to another. Whenever it is required to swap information between two registers or a register and a memory location, the exchange instruction is used. To transfer data content from processor registers and input or output terminals, respectively, the input and output instructions are used. The push and pop instructions are used to transfer data content between processor registers and a memory stack.

The different addressing modes for the load instruction are shown in Table 4.5, where ADR is the memory address, NBR is the number or operand, *X* is the index register, *R1* is the processor register and AC is the accumulator.

**Table 4.5** Addressing Modes for Load Instructions

<i>Addressing Mode</i>	<i>Instruction</i>	<i>Register Transfer</i>
Direct	LDADR	AC ← M[ADR]
Indirect	LD@ADR	AC ← M[M[ADR]]
Register	LDR1	AC ← R1
Register Indirect	LD(R1)	AC ← M[R1]
Relative	LD\$ADR	AC ← M[PC + ADR]
Index	LDADR(X)	AC ← M[ADR + XR]
Immediate	LD#NBR	AC ← NBR

The character @ before the memory address indicates an indirect address. In case of the register indirect mode, the register that holds the memory address is enclosed in parentheses. The character \$ before the memory address makes the address relative to the program counter PC. The character # before the operand indicates immediate mode instruction.

### 4.6.1 Data Manipulation Instructions

Data manipulation instructions perform arithmetic, shift or logic operations to manipulate data. Thus, data manipulation instructions are broadly divided into the following three basic categories:

- (i) Arithmetic Instructions
- (ii) Shift Instructions
- (iii) Logic Instructions

#### (i) Arithmetic Instructions

Addition, subtraction, multiplication and division are the four basic arithmetic operations. Most computers provide instructions to perform these operations. Increment (or decrement) instructions add 1 (or subtract 1) to the value stored in a register or some memory word. A list of standard arithmetic instructions is shown in Table 4.6.

**Table 4.6** Arithmetic Instructions

<i>Name of Instruction</i>	<i>Mnemonic</i>
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Increment	INC
Decrement	DEC
Negate (2's complement)	NEG
ADD with Carry	ADD C
Subtract with Borrow	SUB B

#### (ii) Shift Instructions

Shift operations can be a circular or arithmetic shift, or a simple logical shift in which the bits of a word are moved to the left or to the right. For both the cases, i.e., logical shift left or logical shift right, 0 is inserted at the end bit position. The rotate instruction produces a circular shift. It circulates the bits around the two ends without loss of information. Instructions like rotate through carry treats a carry bit as an extension of the register whose word is being rotated. Thus, the rotate left through carry instructions transfer the carry bit into the rightmost bit position and transfer the leftmost bit into the carry and at the same time shift the entire register to the left. The arithmetic shift left instruction inserts 0 at the rightmost bit position. The arithmetic shift right instruction leaves the sign bit unchanged and shifts the bit (including the sign bit) to the right, and the rightmost bit is lost. The basic shift instructions are listed in Table 4.7.

### NOTES

**NOTES**

**Table 4.7** Shift Instructions

<i>Name of Instruction</i>	<i>Mnemonic</i>
Logical Shift Left	SHL
Logical Shift Right	SHR
Rotate Left	ROL
Rotate Right	ROR
Arithmetic Shift Left	SHLA
Arithmetic Shift Right	SHRA
Rotate Left through Carry	ROL C
Rotate Right through Carry	ROR C

**(iii) Logic Instructions**

Various logical instructions are listed in Table 4.8. **AND**, **OR** and **XOR** instructions provide the corresponding logical operations. The complement instruction produces the 1's complement of the operand. A clear instruction replaces all bits of the operand by 0's. **Clear carry, set carry and complement carry** are instructions that are performed on the individual bits. If the instruction is clear carry, the carry bit is cleared to 0. If it is set carry, the carry bit sets to 1. Similarly, if the instruction is complement carry, the carry bit complements and the bit changes from 0 to 1 or from 1 to 0.

**Table 4.8** Logical Instructions

<i>Name of Instruction</i>	<i>Mnemonic</i>
AND	AND
OR	OR
Exclusive-OR	XOR
Complement	COM
Clear	CLR
Clear Carry	CLRC
Set Carry	SETC
Complement Carry	COMC
Enable Interrupt	EI
Disable Interrupt	DI

**4.7 MICRO-PROGRAMMED CONTROL**

The other alternative for implementing control operation is through microprogrammed organization where the control unit is implemented through programming. In this case, the execution of microoperation is done through stored program logic instead of hardwired control circuitry.

Each instruction is executed by a set of microoperations, termed as microinstructions. For each microoperation, the control unit generates a set of

control signals. A complete instruction is executed by generating a sequence of control signals (group of microoperations) that are appropriately timed. This sequence of microinstructions is termed as a microprogram or firmware. Such a program comprises several instructions, with each instruction describing the following:

- One or more microoperations that are to be executed
- Information about the microinstruction that is to be executed next.

The microprogram is essentially an interpreter, written in microcode and stored in firmware (ROM, PROM or EPROM) which is often referred to as the control memory or control store. As it is halfway between hardware and software, it is also known as firmware. Compared to hardware, firmware is easier to design, whereas compared to software, firmware is difficult to write. In the microprogrammed organization, the complete control information is stored in a control memory. The control memory is programmed for initiating the required sequences of operations. This program converts machine instructions (stored in binary format) into control signals. There is essentially one subroutine for each machine instruction in this program.

The term microprogram was first coined by M.V. Wilkes in early 1950. During his work on a stored program computer, called the Electronic Delay Storage Automatic Calculator (EDSAC), Wilkes noticed that the sequencing of control signals within the computer was very similar to that required in a regular program. This made possible the use of a stored program for representing the sequences of control signals. The first paper on this technique, termed microprogramming, was published by Wilkes in 1951. In this paper, he proposed an approach to design a control unit in an organized and systematic way, avoiding the complexities of a hardware implementation. As it was impossible to manufacture fast control stores in the 1950s, microprogramming could not turn out to be a mainstream technology at that time.

Later, in the late 1950s, John Fairclough's research at IBM's Laboratory in Hursley, England, led to the development of a read only magnetic core matrix for use in the control unit of a small computer. In 1961, his research played an important role in IBM's decision to pursue a full range of compatible computers, which was announced in 1964 as the System/360. Since then, microprogramming has become popular in variety of applications, one of which being the use of microprogramming to implement the control unit of a processor, particularly in Intel 80x86 and Motorola 680x0 processors, whose instruction sets are essentially evolved from the 360 original. In fact, IBM still produces mainframes that use the same architecture.

Each microinstruction cycle is made of two parts: fetch and execute. The fetch cycle includes the fetching of instructions that leads to initiation of a series of microinstructions stored in control memory. The function of these microinstructions is to issue the micro-orders to the CPU. Micro-orders generate the effective address of operand, execute the instruction and prepare it again for fetching the next instruction from the main memory. To execute an instruction by Microprogrammed

## NOTES

**NOTES**

Control Unit (MCU), the following two tasks are performed:

- **Microinstruction Execution:** This generates a control signal to execute the microinstruction.
- **Microinstruction Sequencing:** This provides the next microinstruction from the control memory.

To implement the control process, the microprogrammed units have the following components (Refer Figure 4.21):

- **Instruction Register:** This holds the instruction to be executed.
- **Microinstruction Address Generation:** This generates the address where the instruction (to be executed) is stored in the control memory. This address generator uses clock signal, input from flag or status register, instruction that is stored in instruction register and produces an output which is the address of the control memory.
- **Control Store Microprogram Memory:** This stores the control words.
- **Microinstruction Buffer:** This stores the instruction.
- **Microinstruction Decoder:** This decodes the instruction into a sequence of control signals.

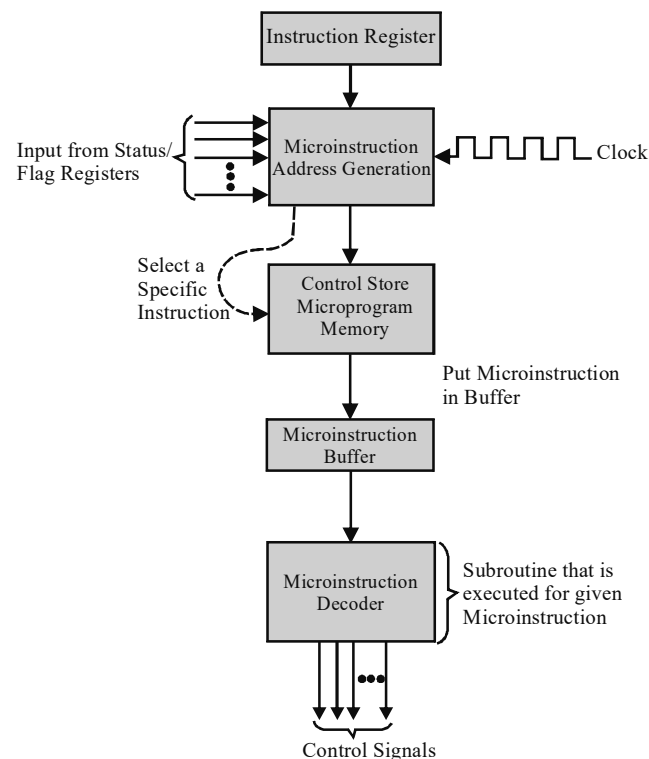


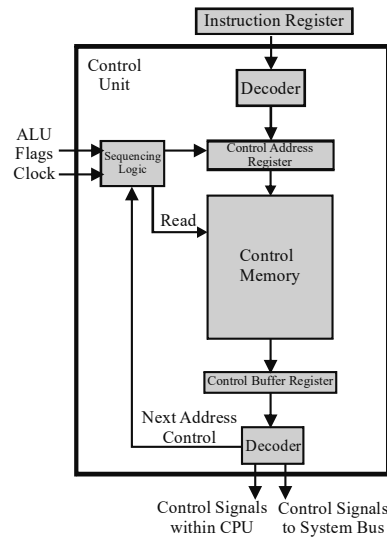
Fig. 4.21 Microprogrammed Control Unit

### 4.7.1 Execution of Complete Instruction

For the execution of a program, the series of microinstructions must be determined. For different instructions, there are different series of microinstructions. In order



to perform these tasks, a Microprogrammed Control Unit (MCU) should have the following components (Refer Figure 4.22).



## NOTES

*Fig. 4.22 Fundamental Components of a Microprogrammed Unit*

- **Microprogram Sequencer (MS):** It is also called next address generator. It generates the address of the next microinstruction that needs to be retrieved from the control memory.
- **Control Address Register (CAR):** Its function is to hold the address of control memory generated by microprogram sequencer.
- **Control Memory (CM):** It stores all microprograms that are constituted of control words. It is usually a ROM.
- **Control Buffer Register (CBR):** It basically performs three functions: (i) Holding the control word retrieved from control memory, (ii) Generating/propagating the control function values to the MCU and (iii) Providing the information required for generating the next address. The CAR and CBR are such registers that may be used and modified in parallel. Thus, CBR leads to the execution of a collection of microoperations. It is also simultaneously used to generate the next address (via the sequencer) for the CAR.
- **Decoder:** The upper decoder translates the opcode of IR into a control memory address. The lower decoder is not used for horizontal microinstruction; it is used for vertical microinstruction.

In general, a microcode execution involves the following steps to be executed in one clock pulse:

- The instructions are fetched from main memory and are stored in IR.
- Opcode is decoded.
- To execute an instruction, the microprogram sequencer issues a READ command for control memory.
- Microinstructions are retrieved from CM from the address specified in CAR.

## NOTES

- The microinstruction is read from control memory and is transferred to a control buffer register.
- The content of control buffer register generates control signal and address information for the sequencing logic unit.
- The microprogram sequencer loads a new address into CAR based on the next address information from control buffer register and ALU flags.

### 4.7.2 Reduced Instruction Set Computer (RISC)

Pronounced as 'Risk', RISC is a type of microprocessor that is designed with limited number of instructions. With the advent of X86 series, computer manufacturers tended to design a CPU with a complex and large instruction set, resulting in a complex hardware. But in a research by U.C. Berkeley and IBM in the early 1980s, researchers found that most computer language compilers and interpreters used only a small subset of the instructions of a CISC. In his research in 1972, John Cocke of IBM Research found that a computer used only twenty per cent of the instructions, i.e., the remaining eighty per cent were superfluous. Hence, a demand arose to design a processor with a simpler and less orthogonal instruction set, so that their execution is fast as well as less expensive. CPUs became fast with calculations involving less memory access.

To implement this concept, computer designers experimented to design a processor using large sets of internal registers. A processor based upon this concept has a small instructions set, requiring fewer transistors. This makes its manufacturing cheaper. Also, as the use of transistors and instructions is restricted to only those that are most frequently used, the computer works more in less time.

The term 'RISC' was later coined by David Patterson, a teacher at the University of California in Berkeley. After the emergence of RISC computers, it became a practice to refer conventional computers as CISC's. RISCs generally had larger numbers of registers, accessed by simple instructions set like load and store, for transferring data to and from memory. The result was a very simple core CPU running at very high speed and supporting all type of operations that compilers were using.

The RISC architecture is basically designed on Harvard architecture, as contrast to the Von Neumann (Stored Program) architecture on which most pre-RISC processors were designed. In the Harvard architecture, machine program and data are two different entities. So, separate memory devices are used to store the program and data, but these are accessed simultaneously. The Von Neumann architecture, on the other hand, considers the data and programs as the same entity and stores them in a single memory device. As both of them are accessed sequentially, it produces the so-called 'Von Neumann bottleneck'.

The major disadvantage of RISC architecture is that compilers have to generate longer sequences of the simpler instructions to accomplish the same results. So, the programs that run on RISC machine are usually large compared to CISC. However, recently, engineers have found ways to compress the reduced instruction sets that can fit in memory systems that are smaller than CISCs, e.g., the ARM's 'Thumb' instruction set. The main users of RISC processors are those systems that require low power or have small size, such as embedded systems. The 32-bit embedded systems market is now dominated by the RISC chips, while the smaller

RISC chips are becoming popular in the 8-bit embedded-system market. Power PC series are another examples of RISC processors. Even some CISC processors (which were created before RISC became popular) translate instructions into a RISC-like instruction set internally. As Intel X86 designs dominate the vast majority of desktop sales, CISC architecture still dominates the market. The Apple, Sun and SGI desktop computer lines are few examples which are based on the RISC architecture. Apart from desktop computers, processor chips are dominantly used for embedded systems in cars and other household equipments which are primarily based on the RISC architecture.

## NOTES

### Characteristics of RISC

The characteristics of RISC are as follows:

- Small instruction set; less than 150 machine instructions
- Simple instructions, usually register-based instructions to allow fast execution
- Simple addressing modes (less than 2) to allow fast address computation, as instructions are register based so complex addressing modes are not used.
- Fixed-length instructions, i.e., all instructions have the same length of 32 bits (or 64 bits)
- Small number of the instruction format; less than 2
- Fields aligned in instruction to allow fast instruction decoding
- Presence of both operands in registers to allow short fetch time
- Large number of General Purpose Registers(GRPs); more than 32
- Only one main memory access per instruction
- Onlyread/write (load/store) instructions to access the main memory
- Translation of the complex tasks into simple operations by the compiler, increasing the compiler complexity and compiling time
- Compiler in RISC processors not developed for a specific chip; instead, it is developed in conjunction with the chip to produce one unit
- Simpler and faster hardware implementation
- Very suitable for pipelined architecture
- Single cycle execution
- Design dominated by hardwired control unit
- Supportive to the High-Level Language (HLL)
- All operations related to registers task
- Registers managed in the form of a variable window in some RISC processors, allowing a 'Look' at certain register files instead of using registers as ax, bx, etc.

### Advantage of RISC machines

RISC machine has the following advantages over CISC machine:

- Smaller instruction set
- Single-cycle execution resulting in faster execution

## NOTES

- Fast instruction decoding because of fixed format
- Easy implementation of pipelining in instruction through interleaving many instructions
- Memory access done only by load/store instructions; execution of all other instructions using internal registers only
- Simple design and short design time
- Best target for the state-of-the-art optimizing compilation techniques
- Simplified interrupt service logic

### Check Your Progress

1. What is the function of CPU?
2. What is a control word?
3. Define Stack.
4. What are push and pop operations?
5. What does a stack pointer contain?
6. What is a memory stack?
7. Define the three basic notations.
8. List the different types of CPU organization.
9. How immediate addressing is useful in addressing modes?
10. Why displacement addressing is considered as a very powerful mode of addressing?
11. How are computer instructions classified?
12. State different arithmetic operations?
13. What is RISC?

---

## 4.8 PERIPHERAL DEVICE

---

The peripheral devices can be thought of as transducers which can sense physical effects and convert them into machine-tractable data. For example, a computer keyboard, which is one of the most common input devices, accepts input by the pressing of keys, or by physically moving cursor using mouse. Such physical actions produce a signal that the processor translates into a byte stream or bit signal so that it can understand it. Similarly, if we consider an output device like a computer monitor screen, it accepts a bit stream generated by a processor which is further translated into the signal that controls the movement of the electronic beam that strikes the screen. The pixel combination produces a picture on the monitor screen. Some devices mediate both input and output, e.g., memory or a disk drive.

These input and output devices are usually designed on the complex electromechanical principles. The principles on which these device are designed, principles on which they operate and their working are quite different from the purely electronic devices that are used for designing CPU and its components.

Thus, accepting the data signal and converting it into binary format that a system can understand is a complex technology. So is converting binary data into signal which the device can understand to produce an output. Thus, So, in probability if an architect wants to makes it processor better, it requires only altering a few gates with Computer Aided Design (CAD) tool or altering kind of memory used. Every technology is generally alike and its constant responding is comparatively easy to guess. But it is not the same for I/O devices as the principles on which they work are different and based on techniques not same as those on which a computer architect works. Thus, in order to improve an I/O device, the designer can only suggest the desired improvements to the manufacturers. Also, in many devices, the standards are set for some other device which is later considered as a part of input output system. For example, in case of a CD-ROM, when the standards were being developed, they were set for the audio compact disk, for storing audio signals like music numbers. The incorporation of a standard for computer data was largely an afterthought, i.e. CDs are designed for sequential access rather than random access. The display devices used in computers are derived from standard television technology. Keyboards originated from typewriters, and the layout of a keyboard follows the same standard, i.e. they are designed to reduce the typing speed and increase effort. The designs of many forms of I/O devices are driven by concerns other than optimal performance. Also, often a device is developed specifically for some specialized end users. As mentioned earlier, I/O devices are classified as storage devices, networking and data transmission devices and human-interactive devices. Let us discuss the various types of input output devices.

## NOTES

### Storage Devices

**Hard Disk:** A hard disk is one of the important I/O devices and is most commonly used as permanent storage device in any processor. Most PCs have hard disk having capacities in the range of 20 GB to 80 GB. Due to improvement in technology and density of magnetic disk, it has become possible to have disk with larger capacity and at a cheaper rate. A hard disk basically comprises several metal platters coated with magnetic material. These plates are rotating with high speed (5000 revolutions per second) along the central shaft. Each surface has a read/write head mounted on arm. The head can possibly move in radial direction. Usually, both sides of a platter are used for storing the data. Every part of plane is parted into concentric circles also called as tracks. Every platter has 300 tracks. Parted by spaces every track is parted in approx. 30 sects. For getting data in a particular place, arm has to be placed over a specific part. It takes almost 3-20 mins. Head would wait till the needed section is under it, which would take about six minutes. But once, the sector is established, data is visible at 4 MB/s. The disk controller is required for buffering the data to avoid bus contention and for bookkeeping.

**Diskette (Soft Disk, Floppy Disk):** It is a 3.5-inch diskette with a capacity of 1.44 MB The architecture is similar to hard disk, i.e., it is divided into concentric tracks, which are further divided into sectors.

**Magnetic Tape:** A magnetic tape consists of a plastic ribbon with a magnetic surface. The data is stored on the magnetic surface as a series of magnetic spots. It has a large storage capacity of 2 to 8 GB and slow transfer rate of 160 kB/s to

## NOTES

1 MB/s.

**Optical Disk:** A variety of optical disks are available in market, e.g., CD-ROM, DVD having storage capacities in the range of 128 MB–1GB, etc. These disks read the data by reflecting pulses of laser beams on the surface. It is usually written once with a high-power laser that burns spots in a dye layer and turns it dark that appears as pit on the surface. Such pits are read by a laser beam that reflects into a phototransistor. Due to variations in the thickness of the disk, vibrations, etc., a focusing lens is used to image the pits onto the phototransistor. Data access time of optical disk varies from 200 to 350 minutes with transfer rate of 150 KB/s to 600 KB/s.

**USB Flash Drives (Commonly Called Pen Drives):** These are typically small, lightweight, removable and rewritable. They are one of the most popular modes used for data transfer because they are more compact and generally faster, able to hold more data (commonly used capacity of 2 GB) and more reliable (due to their lack of moving parts and their more durable design) compared to the floppy disks. These are NAND-type flash memory data storage devices integrated with a universal serial bus (USB) interface.

**Magneto-Optical Disk:** A magneto-optical disk is based on the same principle as the optical disk. Both have capacities in the range of 128 MB, 230 MB, 1.3 GB. The only difference is that it uses a layer of magnetic grains that are reoriented by the magnetic write head so that they either block or allow light to reflect off of the backer. As in a floppy disk, the read-write media is stored in a self-sealing rigid case. The time required to access the data is 16–30 minutes, with transfer rate of 2 to 3 MB/s.

### Networking and Data Transmission Devices

With the advent of the Internet, networking has become an important component of today's computer system. Hence, each system should provide network facility. Networking is one of the most important parts of computer peripherals. Although the data transfer is possible through various media, the current computer network is usually built on top of the existing telephone network. Ethernet is commonly used as a media for data transfer. It can transfer data at the rate of 3 Mbit per second. It is an implementation of a one-wire bus. Networking can be done through Local Area Network (LAN) or Wide Area Network (WAN) or wireless communication (mobile communication). LAN connects computers within a few kilometers. Asynchronous Transfer Mode (ATM) is the new technology for high bandwidth network (30 Mbit to Gbit). WAN is designed for long-distance transfer. ARPANET is the first such network. For WAN, phone lines (28 kbits/s), copper and coaxial cables (30 Mbits/s), or optical fibers (1 Gbit /s) are used. Detail of networks is beyond the scope of this curriculum. However, it is another example of I/O peripheral designed specifically for the computer industry.

### Human-Interactive I/O Devices

The human-interactive devices can be further categorized as direct and indirect. Direct devices are those that interact with people. These devices respond to human action and display information in real-time at a rate that complements the capabilities

of people. The main job of these devices involves the translation of data between human-readable to machine-readable forms and vice versa. Direct I/O devices include the keyboard, mouse, trackball, screen, joystick, drawing tablet, musical instrument interface, speaker and microphone.

Indirect devices do not interact with users. These device are used where human are not directly involved in accepting the input or producing the output such as a scanner or a printer. These devices also perform the data translation in the format acceptable to machine. But they do not respond directly to a human in real-time.

### **Characteristics**

Individual devices are typically made up of many component parts which, in turn, affect the rate of the speed at which data can be transferred. Each device inevitably consists of both electronic and mechanical elements. An output event, obtained in some form of electrical signal (binary data obtained after processing from process), needs to be converted into a signal that allows the mechanical movement of the device to occur. As a result, this conversion of signals takes longer access times and subsequently results in the lower data transfer rates. Similar event may occur in an input event where mechanical signal is converted to electrical signal. In addition, regular intervention from the input/output (I/O) controller is required to facilitate this action. The actual intervention from the I/O controller further slows down the data transfer rates. Another characteristic of an input/output device is evidenced in the way the data are written or read from a storage device. These devices are broadly categorized into two types:

**Character Devices:** They transfer data as one character at a time. Here data are handled in a series of bytes (byte stream), that is, data are as one character at a time. There is little structure (if any) of a byte stream. Line printers are examples of character-oriented devices.

**Block Devices:** Here data are handled in blocks that are fixed in size. The data are transferred as one block at a time. This technique was originally designed to save space on a magnetic tape and to reduce the number of input/output requests needed.

The human-interactive devices can further be classified as input devices and output devices.

### **Input Devices**

Input devices collect the information from the end user or from a device and convert this information or data into a form which can be understood by the computer. We characterize an input device as good if it can provide useful data to the main memory or the processor directly and timely for processing. Some common input devices which allow to communicate with the computer are as follows:

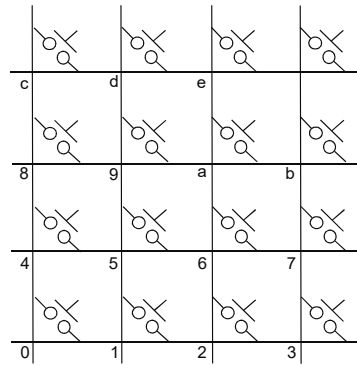
#### **Keyboard**

A keyboard is one of the most common input devices attached to all computers. This input device may be found as part of an on-line/interactive computer system used for entering the character. The layout of keyboard is similar to the traditional typewriter of the type QWERTY as it is designed basically for editing the data. In

## **NOTES**

## NOTES

other words the same function as that of a keyboard. However, the keyboards of a computer contain some extra command keys and function keys. They contain a total of 101 to 104 keys. One can input the data by pressing the correct combination of keys to input data. Other keyboard layouts that have been used include DVORAK, MALTRON, VELOTYPE, all of which have been developed in an attempt to increase data entry speed. If we consider the configuration of keyboard, it is just an assembly of switches that are logically arranged in the form of a matrix as shown in Figure 4.23. When a key is pressed (mechanical action), the switch connecting a row and column of wires is energized and the combination specifies a particular key.



**Fig. 4.23** Logical Arrangements of Keys

When a combination of keys is pressed, an electrical signal corresponding to it is generated. The computer can recognize the electrical signal and a byte stream for that key is generated which is forwarded to the system for processing. If we study the working of keyboards, the electronics of the keyboard has to perform two functions:

- It must 'Debounce' the key pressed, i.e., the keys must have a tendency to bounce back as soon as they are released so that switch comes back to open state.
- It must translate the row and column into a standard code and then send this as a sequence of pulses to the CPU. Corresponding to each combination of key, there is a unique code stored in table lookup. Thus translation is done through a table lookup, and the corresponding electrical signal is generated through a circuit called the Universal Asynchronous Receiver/Transmitter (UART). A typical keyboard produces bursts of character codes at a rate of up to 3 per second (120 words per minute) but the average data transfer rate is even lower.

### Pointing Devices

There are many pointing devices, such as light pen, joystick, mouse, etc.

**Mouse:** Of all the pointing devices, the mouse is the most popular device used with keyboard for accepting input. Its popularity is primarily due to the fact that it provides very fast cursor movement providing the user the freedom to work in any direction. It rolls on a small ball and has two or three buttons on the top. As the mouse is moved on a flat surface, the cursor on the screen moves in the direction

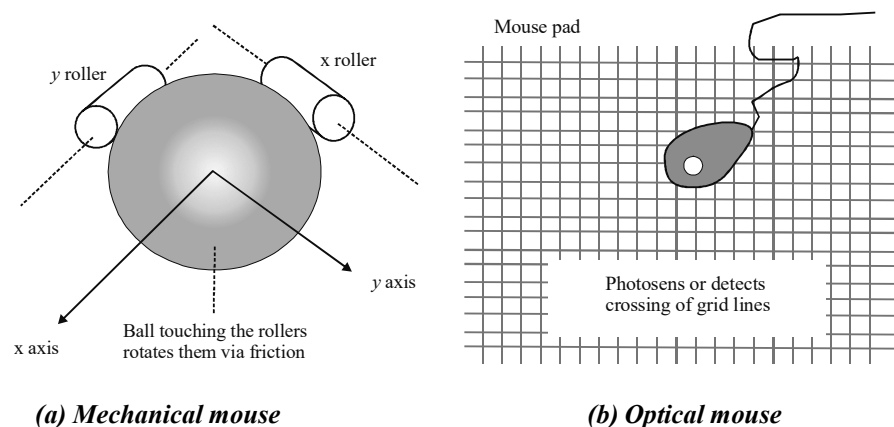


in which mouse moves. Two commonly used mouse are mechanical mouse and optical mouse.

**Mechanical Mouse:** As shown in Figure 4.24(a), in a mechanical mouse, there is a ball that protrudes under the housing. This ball is rolled across a flat surface. The ball movement turns a perpendicular pair of shafts inside the housing. The shafts drive encodes the distance travelled by using an encoder that consists of a clear plastic wheel with radial lines printed on it. This type of mouse uses LED and phototransistor to sense the ball movement and generate pulse corresponding to mouse movement.

Let us study how this translation exactly takes place. The basic principle behind the working of a mouse is there is an LED shines that through plastic wheel fall onto a phototransistor, and as mouse moves there is a variation in the light reaching the phototransistor is observed. This variation depends on the number of line it passes on the radial wheel and leads to generation of pulses. This pulse count will be in proportion to distance moved. The pulse generated can either be counted in the mouse itself or it can be sent to the computer for counting. Usually a pair of phototransistors is used so that it can be determined in which direction the shaft is rotating. Thus according to the distance the mouse covers on the flat surface, the corresponding cursor position moves on the screen.

**Optical Mouse:** In an optical mouse, as shown in Figure 4.24(b), we use a pair of LEDs. These LEDs shine on a special reflective pad which is printed with a grid of lines having two different colours; generally blue lines run horizontally and black lines run vertically. Two phototransistors are used to sense the reflected light. They determine direction in which mouse is moved across the pad. Each phototransistor is sensitive to one colour and is elongated in the particular direction. Like in mechanical mouse, the distance covered is measured by the count of the pulse that is resulted from the reflections of the dots. These pulses are either sent to the computer for counting or the pulses are counted in mouse only and the count result is sent to the computer.



**Fig. 4.24** Mechanical Mouse and Optical Mouse

The mouse is one of the devices designed solely for the computer industry. It can send data at the rate of 20 bytes per second. The information is sent to processor in serial manner, same as in the keyboard.

## NOTES

## NOTES

Working of the trackball is similar to a mechanical mouse. The only difference is that in trackball, the ball used is typically larger, and the user rolls it with his or her fingers or hands. The objective of trackball is again the cursor movement.

**Joystick:** A joystick is specially used in systems that are designed for gaming purposes. It is based on the principle of electricity, i.e., it is a resistive device. It consists of a stick that turns the two shaft potentiometers, one for *X* direction and the other for *Y* direction. The movement of stick is just like the volume knob on a radio. Different positions of potentiometer result in different voltage outputs. Using an Analog-to-Digital Converter (ADC), the output from the potentiometer's resistance at that particular position is converted into a corresponding number. Thus, in case of joystick also, the distance covered will give a particular output. This output of the ADC is then serialized and sent to the computer for further processing in similar manner as in a keyboard.

### Voice Input Systems

A system that enables a computer to recognize the human voice is called the voice-input system. The two commonly used voice input systems are: microphone and voice recognition software.

**Microphone:** The microphone turns acoustical pressure into a variation in voltage. The digital value of this voltage is obtained by dividing the analogue signal at regular intervals (the sampling rate) and average integer value of each sample is accepted as output. This digitized signal can be used for recording, as in audio CD, or can be converted into text by processing it by voice recognition software.

**Voice Recognition Software:** It is complex software. To extract phonemes and whole words from a voice message, we need software that is a combination of both signal processing and artificial intelligence techniques. Thus, we need a very powerful machine and a dedicated signal processing computer to implement it. But then also it may be limited to a single person for which it is trained or if there are multiple speakers we have to limit for just a small number of words and phrases.

### Source Data Automation (Scanner)

Often the data supplied is not in text format. For example, data may be in the form of pictures, etc. In such cases, the keyboard cannot be used as it can enter data through pressing a combination of keys. We need some devices that can collect data at their point of origin in digital form. Scanner is one such device.

Scanner is used to accept an input in any graphical format, store it in digital format and display it back if required. It is an optical device that can read text or illustrations printed on paper and translate this information into a form that a computer can use. In order to scan a black and white image, it divides the page into grid of boxes. If a box is to be filled, i.e., black in colour 1 is stored corresponding to that box else 0 is stored. In case of gray scaling and coloured images, the same principle is applied. However, the only difference is that instead of one bit for one box in case of black and white images, here there are 24 bits corresponding to each box. Thus, a scanner digitizes an image by storing a combination of bits corresponding to each box. The result is a bit matrix where row and column value represent the grid positions and value stored is value corresponding to the box. This is called a bit map. This bit map can be stored in a file. It can also be manipulated by programmers and displayed on a screen.

Optical scanners do not distinguish between text and illustrations. For them, all images are represented as bit maps. Hence, the editing of textual data is not possible in scanned image. To edit text read by an optical scanner, we need an optical character recognition system that decodes the bit image into ASCII characters. The common optical scanner devices are: Magnetic Ink Character Recognition (MICR), Optical Mark Reader (OMR) and Optical Character Reader (OCR).

**MICR:** It is a popularly used technique in the banking sector. All banks now issue cheques and drafts. As cheques enter an MICR machine, they pass through the magnetic field which causes the read head to recognize the character of the cheques. It has vastly helped the banking sector in authenticating the cheques.

**OMR:** This technique is vastly used in evaluating the objective answer-sheets. The students appearing in an objective test have to mark the answer by darkening a square or circular space in their answer-sheets by pencil. These answer-sheets are directly fed to OMR machine which by observing markings evaluates the sheets.

**OCR:** This device can read any printed character by comparing the pattern that is stored in computer. We keep a character of a hand-written image on a piece of paper and put it inside the scanner. The scanned pattern is compared with pattern information stored inside the computer. Only those patterns that are matched are read, this process is called a character read and the remaining unidentified patterns are rejected.

### Digital Camera (Video Camera and Tape)

A video camera records the image, converts it into digital format via an ADC and stores it on a frame buffer. A data rate of 28 MB/s can be achieved for a fully digitized system where there is no compression. But it can be improved to 80 kB/s, by using compression, which can lead to loss of some information. It can be further improved to 160 kB/s with broadcast quality. A dedicated digital signal processor is used for compression of data to be done between the ADC and the frame buffer.

### Sensor

Sensors are non-interactive type of devices, i.e., they are the devices which accept the non-online input and send this input data to computers. The inputs of sensors are the physical properties of devices, such as temperature, magnetic field, etc. Based on these properties, various types of sensors are designed, such as chemical sensors (that sense chemical combination), temperature sensors (that sense temperature), magnetic field sensors, etc. These sensors can have binary input, such as on and off, or multi-valued analog input, which is converted to digital value via ADC, and occasionally image as in case of ultrasound, CAT and MR scans, radar, IR, UV, radio, etc. Data transfer rates vary from bytes per hour to GB/s, depending on input.

## NOTES

## NOTES

### **Actuator**

Actuators are also non-interactive input devices widely used for accepting input from control devices, such as switches, valves, solenoids, motors, stepper motors, linear motors, lights, lasers, electron beams, X-rays, hydraulic pumps, and so on, that are controllable by computers. In these devices also the data transfer rates vary from B/s to kB/s.

### **Output Devices**

Output devices are those equipments that accept data and programs from the computer and provide them to users. Output devices are commonly referred to as terminals. Terminals can be classified into the following two types: (i) A *hard copy* terminal that provides a printout on paper and (ii) A soft copy terminal that provides visual copy on the monitor.

Terminals can also be classified as dumb terminals or intelligent terminals depending upon how they work.

Some important output devices are discussed as follows:

### **Visual Display Unit ( Monitor)**

Visual Display Unit (VDU), popularly known as monitor, is the most popular output device. It resembles a television screen. This device may form part of an interactive computer system that displays a response, message or request received from the computer to the user. No further processing will take place until the necessary action is taken by the user. The response time from the user is inevitably far slower than any action undertaken by the processor.

A monitor consists of a Cathode Ray Tube (CRT). It is attached separately to the main computer system via a cable. In a CRT tube, an existing electron beam (essentially a small linear accelerator) heads toward the screen. The position of beam is controlled by an orthogonal pair of charged plates. These plates appropriately deflect the charged electron beam. The intensity of electron beam scanning the screen can be varied to produce variations in brightness on the screen. The screen is coated with phosphorescent material which emits light when the electrons beam strike it. Thus, the kinetic energy of electron beam is converted into light. The material continues to emit light for a brief period after it is struck by the electron beam (called the decay period), so that the screen appears to remain evenly illuminated. In coloured screen, there are triads of phosphor dots where each dot emits one of the primary colours. The colour pattern is achieved by a mask dot pattern. The plates that control the beam movements are modulated so that it produces the desired colour pattern by illuminating the appropriate combination of signals in each triad (Refer Figure 4.25).

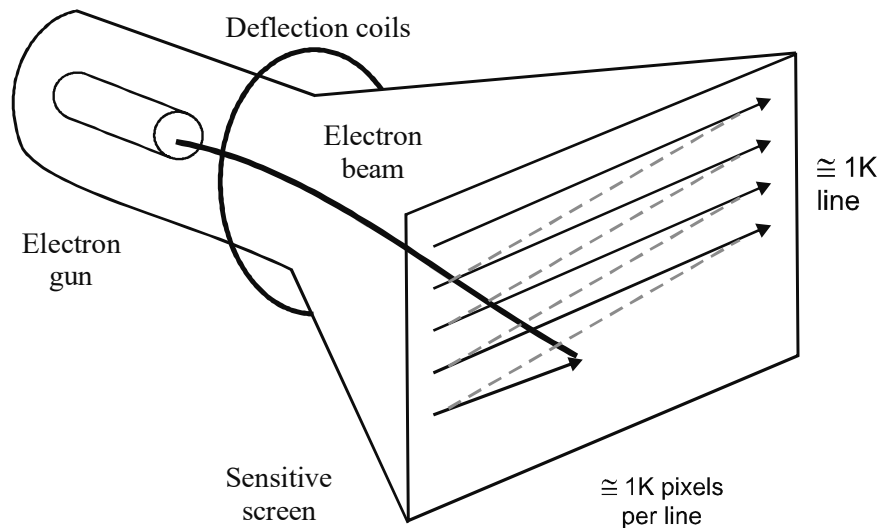


Fig. 4.25 Image Formations on CRT

The quality of screen is determined by the following factors:

- The *resolution* is an important parameter of screen. It defines the sharpness and clarity of an image. It depends on the number of pixel on the horizontal and vertical lines and it ranges from 320 by 200 to 1600 by 1200. The typical monitors have resolution of  $640 \times 480$ . High-resolution monitors are becoming common these days.
- The *range of colour* is also an important parameter of screen. It depends on the number of bits associated with each pixel and ranges from 8 to 24 bits.
- The *pitch*, i.e., the pixel per inch, which is usually 32 to 30, determines the quality of a screen.
- The *size of monitor*, which is measured diagonally including a portion of the tube, is an important parameter of determining the quality of a monitor. The size of monitors ranges from 12–21 inches.
- The *refresh rate*, i.e., the number of times the electron beam scans the screen in a second is another important parameter of a screen. It is usually 30 Hz per frame.

### Liquid Crystal Display (LCD)

Nowadays common monitors are replaced by LCDs as display devices. LCDs are specially used in laptops because they are compact, sleek and light weight. An LCD uses thin, flat sheet made up of liquid crystals, an organic substance that has both a liquid form and a crystal molecular structure. This thin sheet is placed in front of a light source. Under normal conditions, each molecule of liquid crystal acts as pixels get twisted from their natural state and allow the light to pass through them. When an electric field is set up across the cell, the molecules get stretched, blocking the light. How much light is blocked depends on the electric field. Pixels are completely darkened when there is no electricity. In a coloured LCD, there

### NOTES

are three sub-pixels, one for each primary colour, i.e., green, blue and red. Now depending on the electric field, each pixel will get different light and correspondingly produce the final image.

## NOTES

### Printer

Printer is a hardcopy terminal used to get a printed copy of the processed text or result on paper. A large variety of printers are available in the market, with each designed for different applications. Printers are typically categorized according to speed, the method of printing (e.g., impact or non-impact printing) and the quality of output (e.g. letter quality, high, low, etc.). Line printers are considered impact printers, where the letters themselves make contact with the paper surface. This contact involves a high degree of mechanical movements to produce output. As a result, impact printers are typically slower than non-impact printers. Laser printers are non-impact printers. No keys physically hit the paper. In laser printer, a beam of light writes an image onto the surface of the drum (which forms part of the printer). This, in turn, causes the toner (form of ink) to be deposited and transferred to the paper. Very fast laser printers with a high standard of output are now available. The data transfer rates from the computer to a printer are high as human intervention is not involved. Printer speed is 12 pages per minute (where each page is set at 80 characters wide  $\times$  66 lines)

$$= 12 \times 5280 \text{ bytes per minute} = 63,360 \text{ bytes per minute}$$

$$= 356 \text{ bytes per second}$$

Impact printing and non-impact printing are discussed as follows:

**Impact Printing:** In impact printing, each character is printed on the paper by striking a pin or hammer against an inked ribbon. According the striking pattern, the desired shape appears on the paper. Because hammering is the mechanical process, such printers have very slow speed. The most common printer based on this technology is Dot-Matrix printer, which can print typically 120 to 200 characters per second. These are again of two types:

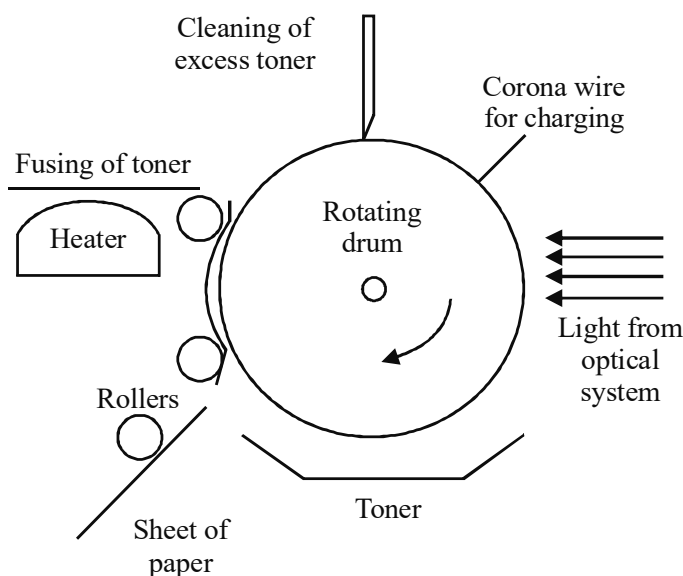
- (i) *Daisy wheels* which can print 40 characters/second. Here the bold characters are achieved by overprinting the text once.
- (ii) *High quality matrix printer*, which can also print 40 characters/second, but with a higher quality, as in such a printer hammer wires put impression on each character four times. If this printer is used, we can also have draught quality print with one pass for every character. In this case, the printing speed of 180 characters/second can be achieved. Bold characters are achieved by overprinting eight times with a horizontal displacement of typically 0.004 inches. Lists of characters that can be printed according to fonts are stored.

**Non-Impact Printing:** The non-impact printing technology prints characters and other images on the paper, or any surface by using principles of electrostatic chemical, heat, lasers, photography, or ink-jets. Ink-jet printers and laser-jet printers are prominent examples of non-impact printing.

*Ink-Jet Printers:* These printers spray tiny droplets of coloured inks on the paper. The pattern of printing depends on how nozzle sprays the ink, which has a quality to get dried within few seconds.

*Laser-Jet Printers:* The working of laser-jet printers is similar to photocopiers. Nowadays there is a tendency to design a device which is hybrid of photocopiers, scanners and printers. In laser-jet printers, there is rotating drum, as shown in Figure 4.26, on which the paper is rotated. Such printers use a low- power laser that charges the paper on the drum with a small electrical charge at the point where a black dot is required. This paper is then passed over a toner tray. The toner tray contains toner, a fine black powder which is attracted to the paper wherever it is charged. As shown in Figure 4.26, the toner is then fixed with a heater. The heater melts the toner onto the paper. The excess of toner is then removed.

## NOTES



**Fig. 4.26** Mechanism of a Laser Printer

## Plotters

Plotters are used for printing the big charts, drawings, maps and three-dimensional illustrations, specially used for architectural and designing purposes.

## Speaker and Sound Generation, Text to Speech Synthesis

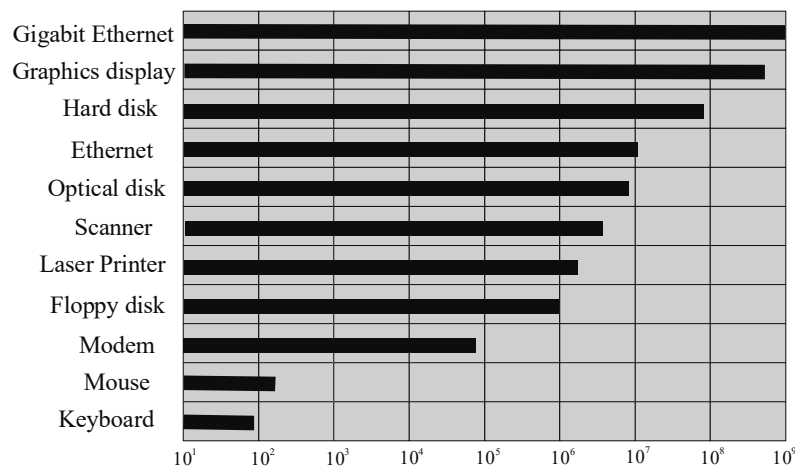
To produce sound signal as output is just opposite to the acceptance of sound input, as discussed earlier. The computer sends digital value to a Digital to Analog Converter (DAC). DAC translates it into an analog voltage which is further amplified and sent to the speaker. Speech is then synthesized using a lookup table of phonemes. These phonemes are clubbed together to form words.

We have studied a variety of input and output devices. As already discussed, all devices have different data transfer rates. The transfer rate of few popular devices and their usage are given in Table 4.9 and are also shown in Figure 4.27.

**Table 4.9** Transfer Rate and Behaviour of Popular Devices

Device	Behaviour	Data Rate
Keyboard	Input	3 bytes/s
Mouse	Input	20 bytes/s
Scanner	Input	0.2 MB/s
Laser printer	Output	0.1 MB/s
Graphics display	Output	30 MB/s
Floppy disk	Storage	50 kB/s
CD-ROM	Storage	0.5 MB/s
Magnetic disk	Storage	2 MB/s
Magnetic tape	Storage	2 MB/s

**NOTES**



**Fig. 4.27** Typical I/O Device Data Rates

**4.8.1 Input-Output Interface**

An Input/Output (I/O) interface is an entity that controls the data transfer from external device, main memory and/or Central Processing Unit (CPU) registers. We can say that it is an interface between computer and I/O devices (external devices) and is responsible for managing the use of all devices that are peripheral to a computer system. It attempts to make an efficient use of all available devices while retaining the integrity of data.

The major problems with the I/O device management are as follows:

- There are various peripherals working on different principles. For example, few of them work on electromechanical principle, few on electromagnetic principle and few on optical principle and so on. As each of them uses different methods of operation it is impractical for the processor to understand and interpret all. Thus, designing an instruction set that can convert the signals into corresponding input value for all devices is not possible.
- As a new I/O device is designed on some new technology, it is required to make the device compatible with the processor. Designing an instruction set for every new device is not at all feasible.



- The rate of data transfer is usually much slower than the processor and memory. Therefore, it is not logical to use the high-speed system bus that communicates directly between I/O device and processor. A synchronization mechanism is required for data transfer to be handled smoothly.
- Peripheral devices accept input in variety of formats. Thus, they may use different data formats and word lengths as used in processor and main memory.
- The operating mode of I/O devices is different for different devices. It must be controlled so that it may not disturb the operation of other devices connected to the processor.

## NOTES

To resolve these problems, there is a special hardware component between CPU and peripheral to supervise and synchronize all input and output transfers. Figure 4.28 illustrates the relationship between the CPU, the peripheral interface chip and the peripheral device. Although the peripheral interface chip may appear just like a memory location to the CPU, it contains specialized logic that allows it to communicate with the external devices. There are a number of such I/O controllers in a processor for controlling one or more peripheral devices.

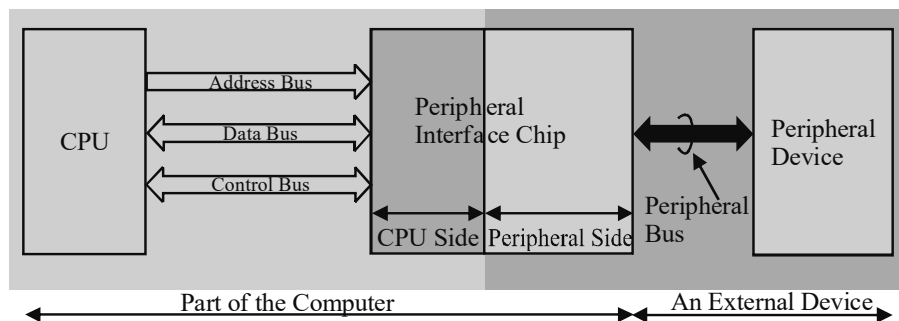


Fig. 4.28 Relationship Between CPU, Peripheral Interface Chip and Peripheral Device

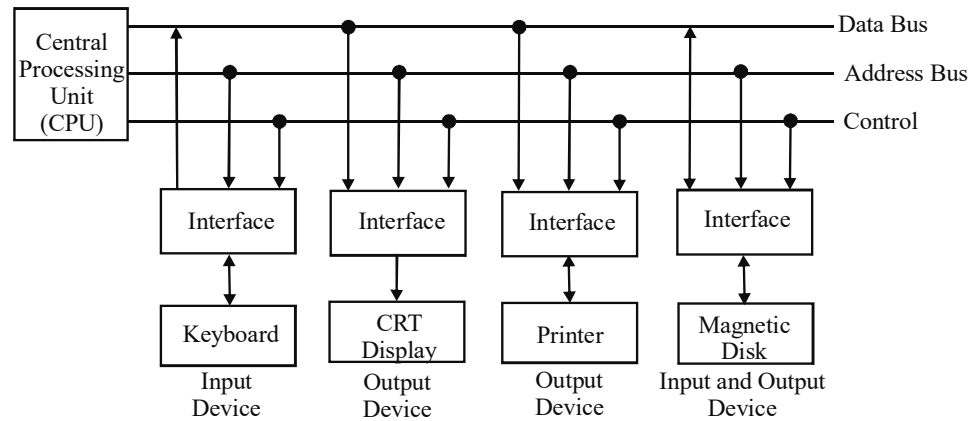
### Accessing I/O Devices

The I/O modules are designed with the aims to:

- **Achieve Device Independence:** It aims to facilitate more simplified software development. In other words, it removes the complexities of individual devices and provides a ‘Translator’ for the use of the device.
- **Handle Errors:** It should ensure that I/O data are correctly handled. It informs the users in event of detection of any error.
- **Speed up Transfer of Data:** As the I/O is typically the slowest part involved in a program’s execution, the direct memory access method will apply on a range of algorithms to enhance both the software and hardware speeds.
- **Handle Deadlocks:** It should monitor conditions that can ‘Lock up’ a system (e.g., resource holding) and should take steps to avoid these conditions.
- **Enable Multi-user Systems to use Dedicated Devices:** It should assign sensible printing instructions, while trying to prevent the erroneous output of data.

Each device may have its own controller that supervises the operations of that device. A typical communication bus system between processor and devices is shown in Figure 4.29.

## NOTES



*Fig. 4.29 Connections between I/O Device and Processor through I/O Bus*

There are three types of buses, namely data bus, address bus and control bus. Each device has an interface through which it is connected to a bus (Refer Figure 4.29). The interface decodes the signal received from the input device in the format that processor can understand, and also interprets the control signal received from processor for peripheral devices. It supervises and synchronizes the data flow between external device and processor. Many devices also have a controller which may or may not be physically integrated on the interface chip. The controller is often used for buffering the data, Integrated Development Environment (IDE) is used as a disk controller.

### Functions of I/O Interface

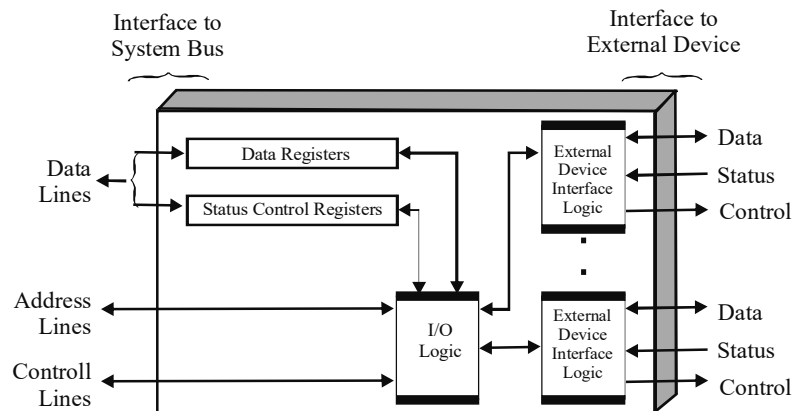
The main functions of the interface are as follows:

- **Control and Timing Signals:** Coordination in the flow of traffic between internal and external devices is done by control and timing signals.
- **Processor Communication:** As a bus is usually employed for data transfer, each interaction between the CPU and the I/O module involves bus arbitration. As the processor needs to communicate with the external device, I/O module must perform the following actions:
  - o **Command Decoding:** I/O module accepts commands, sent as signals on the control bus, from the processor.
  - o **Data:** Through data bus, the data is exchanged between the processor and I/O module over the data bus.
  - o **Status Reporting:** Different devices have different speeds. Few are very slow compared to processor. Hence, it is required for I/O module to know the status before the processor sends the data. Along with various error signals used to verify the data sent, the common status signals used are BUSY and READY.
  - o **Address Recognition:** I/O module must recognize a unique address for each peripheral it controls.

- **Device Communication:** In similar manner as I/O module communicate with processor it has to communicate with device to fetch status information, data transfer rate, etc.
- **Data Buffering :** Data comes from main memory in rapid burst and must be buffered by the I/O module and then sent to the device at the latter's rate.
- **Error Detection** I/O module not only detects error but also reports these errors to the CPU.

## NOTES

Figure 4.30 shows the block diagram of I/O Interface



*Fig. 4.30 Block Diagram of I/O Interface*

The various steps taken for I/O communication with peripheral devices are as follows:

- Processor sends device address of the device it wants to communicate with on the address bus.
- Interface attached to I/O bus contains address decoder. When an interface finds that the device address is on the address line, it activates its path between the bus line and the devices that it controls.
- Processor interacts with I/O module to check the status of external device.
- I/O module returns status.
- The processor provides the operation code on the control line.
- If device is ready, processor gives I/O module command to request data transfer.
- I/O module gets a unit of data from device.
- Data is transferred from the I/O module to the processor.
- Interface interprets the opcode and proceeds accordingly.

There are four types of commands that an interface may receive.

- **Control Command:** This activates the device and informs the device what action to be performed. A particular control command depends on a particular device.

## NOTES

- **Status Command:** Before the peripheral device performs the action required by the processor, it should first check the status of device and interface. In other words, the printer should not get new data until it has printed the pervious data. If there is an error in the device, the same may be responded back to the processor.
- **Data Output Command:** This transfers data from the bus into one of the interface registers.
- **Data Input Command:** The interface receives data from the device and places it in its registers which can be forwarded to the processor by putting these data on the data line of the bus.

### I/O vs Memory Bus

The I/O devices need to communicate with not only the processor but also with the memory unit. There are two ways in which computer buses can communicate with memory.

- (i) **Using Separate Buses for Memory and I/O Device:** We have two separate sets of data, addresses and control buses, one for accessing memory and other for I/O. It is especially useful in those systems that use another I/O processor along with CPU. Memory bus communicates both with I/O processor and with CPU. Figure 4.31 shows independent I/O bus.

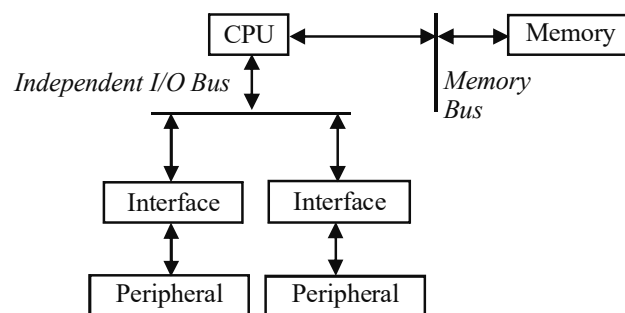


Fig. 4.31 Independent I/O Bus

- (ii) **Using common bus for I/O device and memory:** Separate read and write control line for memory and I/O device is used. Thus enabling the read or write control lines the CPU specifies whether the address bus contains the address of memory location or the interface address. Figure 4.32 shows the diagram for common bus architecture.

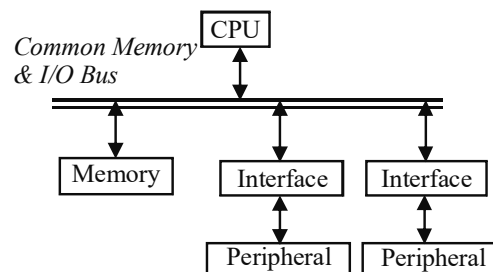


Fig. 4.32 Common Bus Architecture

## Addressing

There are the following two ways in which addressing can be done:

- (i) **Isolated I/O:** Here a separate address space is used for both memory and I/O devices and we need separate memory and I/O select lines. The CPU has distinct input and output instructions. Each of these instructions is associated with address of an interface register. The steps taken for communication are:
- The CPU decodes the operation for input or output.
  - The address of associated device is placed on the common address bus.
  - It enables I/O read (for input) or I/O write (for output) control line.
  - External device now knows that the address on the bus is meant for the interface and not for memory.

Same happens for memory read operation, in which case I/O interface do not participate.

The advantages of isolated I/O are as follows:

- Small number of I/O instructions
- Commonly usable

- (ii) **Memory-Mapped I/O:** It is a single address space for storing both memory and I/O devices. The processor treats the status and data registers of I/O module as memory location. Thus, computer in this case can use memory type instructions to access I/O data. For example, load and store instructions can be used for reading input and writing output for registers, respectively.

The advantages of the memory-mapped I/O are as follows:

- More efficient programming allowed
- Single read line and single write lines needed
- Commonly usable

The disadvantages of the memory-mapped I/O are as follows:

- Valuable memory address space used up
- I/O module registers treated as memory addresses
- Same machine instructions used to access both memory and I/O devices

### 4.8.2 Asynchronous Data Transfer

All the operations in a digital system are synchronized by a clock that is generated by a pulse generator. The CPU and I/O interface can be designed independently or they can share common bus. If CPU and I/O interface share a common bus, the transfer of data between two units is said to be synchronous. There are some disadvantages of synchronous data transfer, such as:

- It is not flexible as all bus devices run on the same clock rate.
- Execution times are the multiples of clock cycles (if any operation needs 3.1 clock cycles, it will take 4 cycles).

## NOTES

## NOTES

- Bus frequency has to be adapted to slower devices. Thus, one cannot take full advantage of the faster ones.
- It is particularly not suitable for an I/O system in which the devices are comparatively much slower than processor.

In order to overcome all these problems, an asynchronous data transfer is used for input/output system.

The word 'Asynchronous' means 'Not in step with the elapse of time'. In case of asynchronous data transfer, the CPU and I/O interface are independent of each other. Each uses its own internal clock to control its registers. There are many techniques used for such data transfer.

### Strobe Control and Handshaking

In strobe control, a control signal, called strobe pulse, which is supplied from one unit to other, indicates that data transfer has to take place. Thus, for each data transfer, a strobe is activated either by source or destination unit. A strobe is a single control line that informs the destination unit that a valid data is available on the bus. The data bus carries the binary information from source unit to destination unit.

**Data Transfer from Source to Destination:** The steps involved in data transfer from source to destination are as follows:

- (i) The source unit places data on the data bus.
- (ii) A source activates the strobe after a brief delay in order to ensure that data values are steadily placed on the data bus.
- (iii) The information on data bus and strobe signal remain active for some time that is sufficient for the destination to receive it.
- (iv) After this time the sources remove the data and disable the strobe pulse, indicating that data bus does not contain the valid data.
- (v) Once new data is available, the strobe is enabled again.

**Data Transfer from Destination to Source:** The steps involved in data transfer from destination to source are as follows:

- (i) The destination unit activates the strobe pulse informing the source to provide the data.
- (ii) The source provides the data by placing the data on the data bus.
- (iii) The data remains valid for some time so that the destination can receive it.
- (iv) The falling edge of strobe triggers the destination register.
- (v) The destination register removes the data from the data bus and disables the strobe.

The disadvantage of this scheme is that there is no surety that destination has received the data before source removes the data. Also, destination unit initiates the transfer without knowing whether source has placed data on the data bus.

Thus, another technique, known as handshaking, is designed to overcome these drawbacks.

## Handshaking

The handshaking technique has one more control signal for acknowledgement that is used for intimation. As in strobe control, in this technique also, one control line is in the same direction as data flow, telling about the validity of data. Other control line is in reverse direction telling whether destination has accepted the data.

**Data Transfer from Source to Destination:** In this case, there are two control lines request and reply. The sequence of actions taken is as follows:

- (i) Source initiates the data transfer by placing the data on data bus and enable request signal.
- (ii) Destination accepts the data from the bus and enables the reply signal.
- (iii) As soon as source receives the reply, it disables the request signal. This also invalidates the data on the bus.
- (iv) Source cannot send new data until destination disables the reply signal.
- (v) Once destination disables the reply signal, it is ready to accept new signal.

**Data Transfer from Destination to Source:** The steps taken for data transfer from destination to source are as follows:

- (i) Destination initiates the data transfer sending a request to source to send data telling the latter that it is ready to accept data.
- (ii) Source on receiving request places data on data bus.
- (iii) Also, source sends a reply to destination telling that it has placed the requisite data on the data bus and has disabled the request signal so that destination does not have new request until it has accepted the data.
- (iv) After accepting the data, destination disables the reply signal so that it can issue a fresh request for data.

## Asynchronous Serial and Parallel Transfers

The data transfer can be serial or parallel. Thus, to transfer a 16-bit data in parallel format, we require 16 transmission lines, one line for each bit. In serial transfer, each bit is sent one after another in a sequence of event. Serial transmission is slow. However, at the same time, it requires just one line, hence, simple to implement. Parallel transmission, on the other hand, requires multiple paths and is a faster mode of transmission.

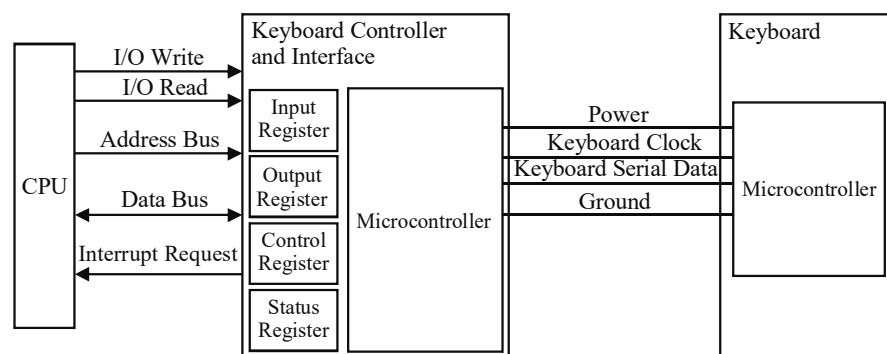


Fig. 4.33 Keyboard Controller and Interface

## NOTES

## NOTES

The keyboard has a serial asynchronous transfer mode. In this technique, the interactive terminal inserts special bits at both ends of the character code. Thus, each character transmission has three types of bits: a start bit, the character bits and stop bits. Usually the transmitter rest at 1 state, it happens when no transmission is done. The first bit, which is 0, is first sent indicating that the character transmission has begun. The last bit is always 1.

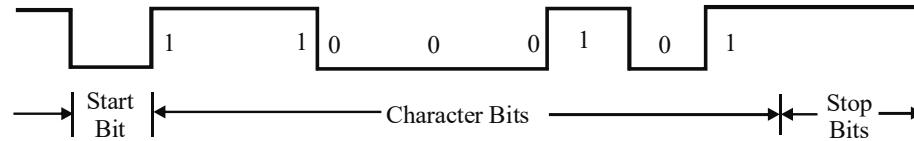


Fig. 4.34 Format of Asynchronous Serial Data Transfer

The various stages in an asynchronous data transmission are as follows:

1. When no transmission is done, the line is kept at 1 state.
2. The character transmission initiates with a start bit which is always 0.
3. The receiver can detect the start bit when line goes from the 1 to 0.
4. The character follows the start bit.
5. The receiver knows the transfer rate and the number of bits to be transferred.
6. After the last bit of character is sent, one or two stop bits of 1 are sent.
7. The stop bit is detected when the line returns to the 1 state for at least one bit.

### 4.8.3 Mode of Transfer

Let us summarize the steps taken to write a block of memory to an output port such that one byte is transferred at a time.

- (i) Firstly, we have to initialize memory as well as the output port addresses.
- (ii) The following steps are repeated until all bytes are transferred:
  - (a) Read one byte from memory.
  - (b) Write that byte to output port.
  - (c) Increment memory address so that next byte can be transferred during the next clock pulse.
  - (d) Verify if all bytes are transferred:  
If **yes**, go to the end of Step (ii).  
If **no**, wait until output port is ready for transferring the next byte. Go to Step (ii)a

Using this approach, we can transfer the data with a speed which is much less than the maximum rate at which they can be read from the memory. Practically, there are various transfer modes through which the data transfer between computer and I/O device takes place with a much faster rate. These modes are as follows:

1. Programmed I/O
2. Interrupt-Initiated I/O



3. Direct Memory Access (DMA)
4. Dedicated processor, such as Input-Output Processor (IOP)
5. Dedicated processor like Data Communication Processor (DCP)

**NOTES**

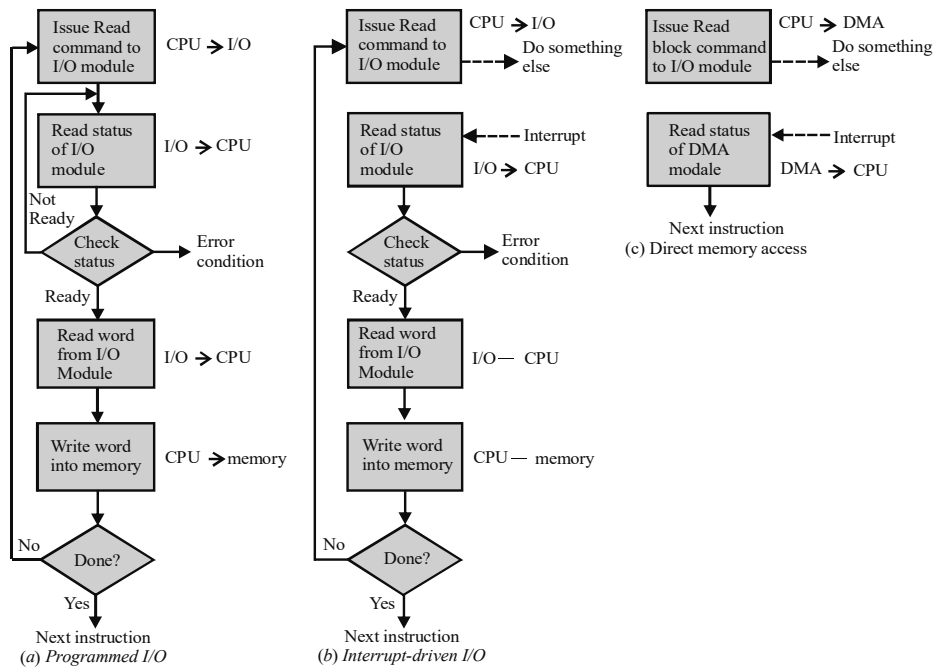


Fig. 4.35 Flow Chart of Various Data Transfer Modes

**1. Programmed I/O**

Programmed I/O operations are the results of I/O operations that are written in the computer program. Each data transfer is controlled by an instruction set stored in the program. When the processor has to perform any input or output instruction, it issues a command for the appropriate I/O module that executes the given instruction as shown in Figure 4.35(a). Processor has to continuously monitor the status of I/O device to see whether it is ready for data transfer. Once it is ready, I/O module performs the requested action and then setting the appropriate bits in the I/O status register alerts the processor for further action.

**2. Interrupt-Initiated I/O**

In programmed I/O, the processor has to check continuously till the device becomes ready for transferring the data. It uses the interrupt facility and issues a command that requests the interface to issue an interrupt when the device is ready for data transfer. Here the interrupt is generated only when device is ready, and hence, till device becomes ready, the processor can execute another program instead of checking the device as it has to do in programmed I/O. Once processor receives an interrupt signal [Refer Figure 4.35(b)], it stops the current processing task and starts I/O processing. After the completion of I/O task, it returns back to original task.

## NOTES

### 3. Direct Memory Access

In direct memory access, the interface transfers the data directly to memory unit via memory bus. The processor just initiates the data transfer by sending the starting address and the number of bits to be transferred and proceeds with the previous task. When the request is granted by the memory controller, the DMA transfers the data directly into memory [Refer Figure 4.35(c)]. It is the fastest mode of data transfer.

### 4. Input–Output Processor

IOP is a special dedicated processor that combines interface unit and DMA as one unit. It can handle many peripherals through DMA and interrupt facility.

### 5. Data Communication Processor

DCP is also a special-purpose dedicated processor that is designed specially for data transfer in network.

#### 4.8.4 Priority Interrupt

In the interrupt driven I/O techniques, the processor starts data transfer when it detects an interrupt signal which is issued when device is ready. This helps processor to run a program concurrently with I/O operations.

Interrupt driven I/O data transfer technique is based on the *on-demand processing* concept. In this, each I/O device generates an interrupt only when an I/O event has to take place like action to be taken if the user presses a keyboard key. The transfer is done by the service routine that processes the required data. The interrupt handler transfers the control to this routine. After the I/O interrupt is serviced, the processor returns the control to the program which was interrupted and is waiting to be executed.

Its main advantages are as follows:

- The processor does not have to wait for long for I/O modules.
- The processor does not have to repeatedly check the I/O module status.

#### Types of Exceptions

Interrupts are nothing but just a type of *exception*. As far as software is considered, there are three types of exceptions:

- Interrupts:** These are raised by hardware at anytime (*asynchronous*).
- Traps:** These are raised as a result of the execution of the program, such as division by zero. As the traps are reproduced at the same spot if the program parameters are the same as before, they are considered as *synchronous*.
- System Calls:** Also called *software interrupts*, system calls are raised by the operating system to provide services for performing certain common I/O tasks, such as printing a character, opening a file, etc.

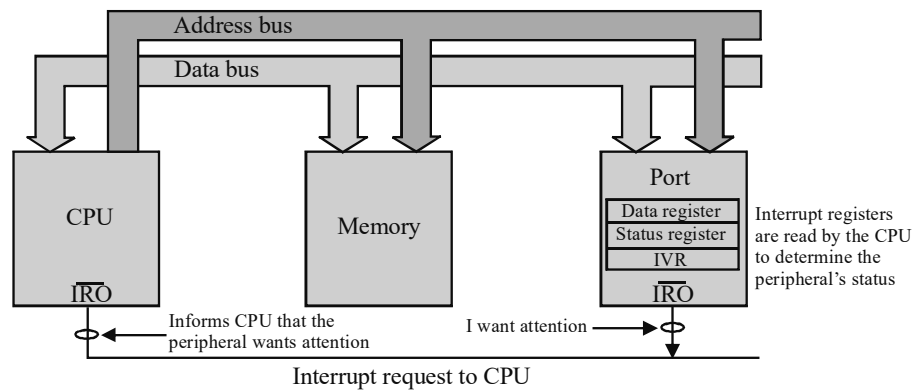


Fig. 4.36 Interrupt Driven I/O

Figure 4.36 illustrates the organization of a system with a simple interrupt-driven I/O mechanism. In most microprocessors, during I/O operation interrupt request, IRQ, is asserted by a peripheral device requesting attention. This *request* may or may not be granted.

### Techniques of Priority Interrupt

A priority interrupt establishes a priority over the various sources to determine which request should be entertained first if several requests arrive simultaneously. The system may allocate a priority. Usually, a high speed system, such as magnetic disk, has a high priority, and one with slow speed, such as keyboard, has a low priority. There are various techniques employed to decide which device to entertain first if two devices interrupt the computer at the same time.

### Polling

Polling is the technique that identifies the highest priority resource by means of software. The program that takes care of interrupt begins at the branch address and polls the interrupt source in sequence. The priority is determined in the order in which each interrupt is entered. Thus, the highest priority resource is first tested if interrupt signal is on the control branch to the service routine. Otherwise, the source having next lower-priority will be tested and so on.

The disadvantage of polling is that if there are many interrupts, the time required to poll exceeds the time available to serve the I/O device. To overcome this problem, hardware interrupt unit can be used to speed up the operation. The hardware unit accepts the interrupt request and issues the interrupt grant to the device having highest priority. As no polling is required, all decisions are done by hardware unit. Each interrupt source has its own interrupt vector to access its own service routine. This hardware unit can establish priority either by a serial or a parallel connection of interrupt lines.

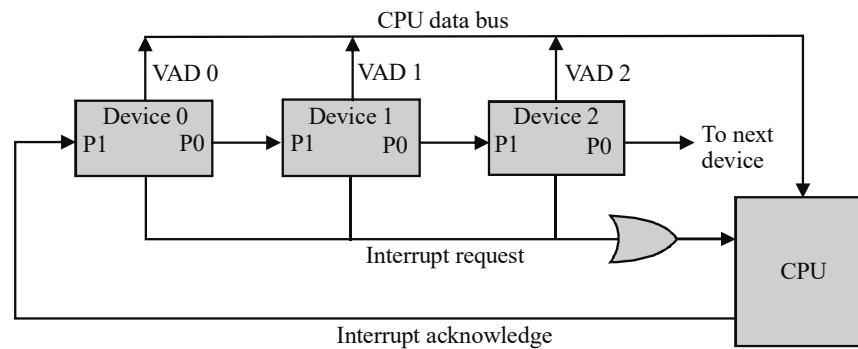
### Daisy Chaining

This method is used to establish priority by serially connecting all devices that request an interrupt. The priority is located according to physical position of device in the serial connection. As in this technique all devices are attached serially, the

## NOTES

**NOTES**

CPU issues grant signal to the closest device requesting it, i.e., the one closest to the processor will have the highest priority. Devices request the interrupt by passing a signal to their neighbours who are closer to the processor as shown in Figure 4.37. This technique is a hybrid of central and distributed arbitration. Here, if device has an interrupt signal, it sets interrupt line into a low state and enables interrupt input to CPU. If no interrupt is pending, the interrupt line stays in high state. The input signal is received by the device 0 at PI (Priority Input). The acknowledge signal passes on the next device through Priority Out (PO). Device 0 will pass the signal only if it has no interrupt request. If there is some pending interrupt, the device 0 will block the acknowledge signal by placing 0 in the PO output. It then proceeds to handle interrupt by placing Vector Address (VAD) into the data bus.



*Fig. 4.37 Daisy Chain Priority Interrupts*

To understand how it actually works, let us study one stage of daisy-chain priority: A device with a 0 in its PI input will have a 0 in its PO output to inform the next-lower priority device that acknowledges that signal is blocked. If the device is requesting for an interrupt has a 1 in its PI input, it will intercept the signal by placing 0 in its PO output. If device has no pending request and PI is 1, then it transmits the acknowledge signal to the next device by placing 1 in its output. Thus, a device having PI=1 and PO = 0 and one with highest priority among the devices requesting interrupt (according to proximity with processor) places its VAD on the data bus for processor. The circuit diagram of one stage of daisy chain priority is given in Figure 4.38. Here Enable is one only if PI = 1, i.e., the device has open interrupt acknowledge and a request for interrupt, i.e., RF = 1 else in all other condition the VAD is Disabled to be passed to data bus. The PO will be 1 when there is no interrupt request from the device to processor, i.e., RF = 0 and PI=1, i.e., interrupt acknowledge is not blocked by pervious device. Only in this condition, the request of the device of next lower priority can be entrained. The truth table for one stage of the daisy chain priority is also given (Refer Table 4.10).

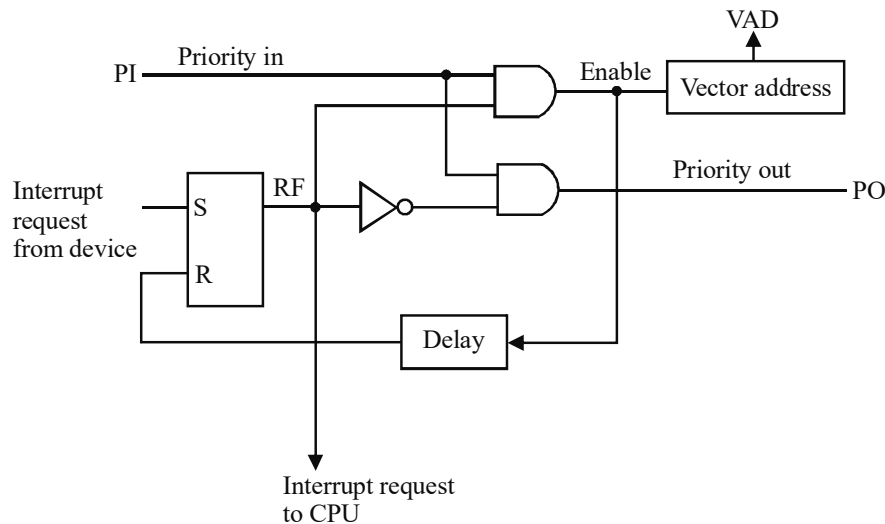


Fig. 4.38 One Stage of the Daisy Chain Priority Arrangement

Table 4.10 Truth Table for One Stage of the Daisy Chain Priority Arrangement

PI	RF	PO	Enable
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

### Disadvantages

The disadvantages of the priority scheme are as follows:

- The priority scheme is fixed by the device’s physical position on the bus and cannot be changed in software.
- If a closer device also requests the bus, then the request from the more distant device is blocked. It may lead to starvation for distant devices if a high priority devices (one nearest to arbitrator) frequently request for the bus
- Daisy chaining is a low cost technique and also susceptible to faults.

However to overcome the disadvantage of assigning priority according to physical position, sometimes multiple request and grant lines are used with daisy chaining to enable requests from devices to bypass a closer device and thereby implement a restricted software controllable priority scheme.

### Parallel Priority Interrupt

The parallel priority interrupt method uses a register whose bits are set separately by interrupt signal for each device. Priority is assigned according to the bit value in the interrupt register and a mask register is used whose purpose is to control the status of each interrupt request. The mask register disables a lower priority interrupt while a higher priority device is being serviced. Suppose there are four devices, as shown in Figure 4.39. The interrupt register bits are set by the external condition and cleared by the program instruction. The mask register has same number of

### NOTES

## NOTES

bits as the interrupt register. It is possible to set or reset any bit in the mask register. The ANDed output of bits of interrupt register and mask register are set as inputs of priority encoder. The priority encoder applies the logic that if more than one input arrives at the same time in the input, the request of one with highest priority will be granted. The two output bits A0 and A1 of priority encoder are the part of the vector address for each interrupt source. The two output bits tell the subroutine which device is to be entertained and stored in VAD.

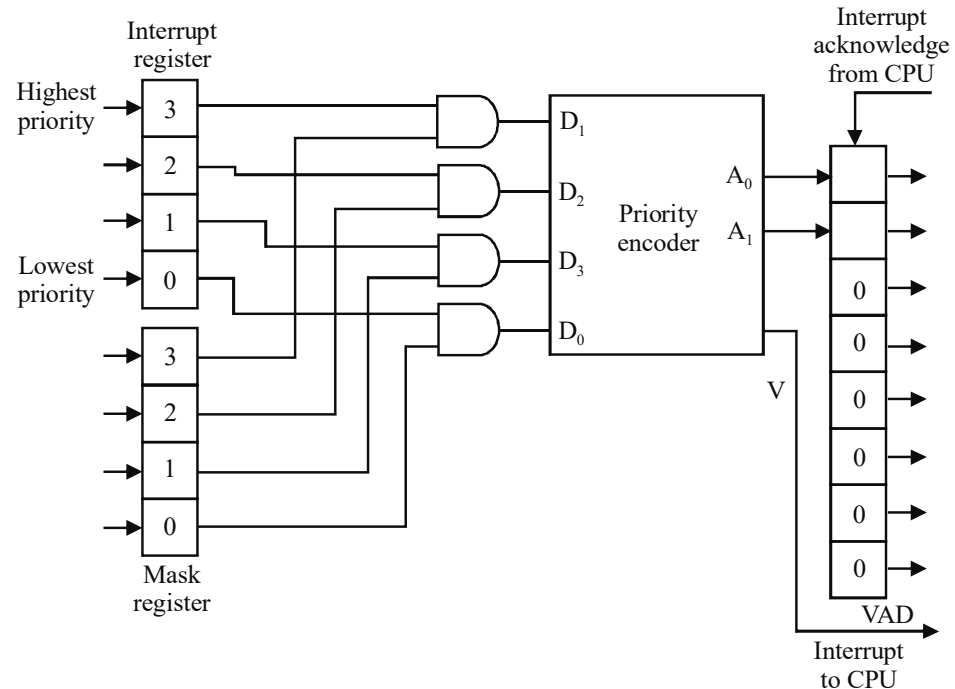


Fig. 4.39 Parallel Priority Interrupt Hardware

### Interrupt Handling using Software Subroutine

Subroutine is an important concept used by programming language. A subroutine is a self contained program (piece of instruction code) that may be invoked or called by main program. After the subroutine has been executed, a return is made to the point immediately after the subroutine call. The subroutine offers economic advantage, i.e., the same piece of code can be reused making efficient use of storage space. As it allows breaking a large program into smaller units, it also provides advantage of modularity. The subroutine mechanism involves two instructions, both of them basically involving branching.

- **Call Instruction:** Branching from current location to the location where subroutine is stored.
- **Return Instruction:** Return from the subroutine to the place from where it was called.

In order to implement the subroutine, we require a place to store the return address. It can be:

- Dedicated register
- Start of subroutine

- Top of stack, where the stack is the ordered set of element which can be accessed only from the top.

Among them, the most powerful technique to handle subroutine is the use of stack. When a CPU invokes a subroutine, it performs the following two functions:

- Pushes updated PC content (return address) on a stack.
- Loads PC with the starting address of subroutine, i.e., the address of the first instruction of subroutine.

After execution of subroutine instruction, pop the top item on stack, i.e., the return address of main program which has called it, to PC.

Let us understand how it works by using a simple example where the main program starting at location 100 has a subroutine (Sub) call at address 120. The program places its return address at the top of the stack, i.e., 121 (next instruction to be executed after subroutine is over). Let us consider that the subroutine Sub starts from 200. Once the address is stored on top of the stack, a branch microoperation will be called and 200 will be loaded to PC so that on the next clock cycle, instructions specified in the subroutine are executed. The execution of subroutine using stack is shown in Figure 4.40.

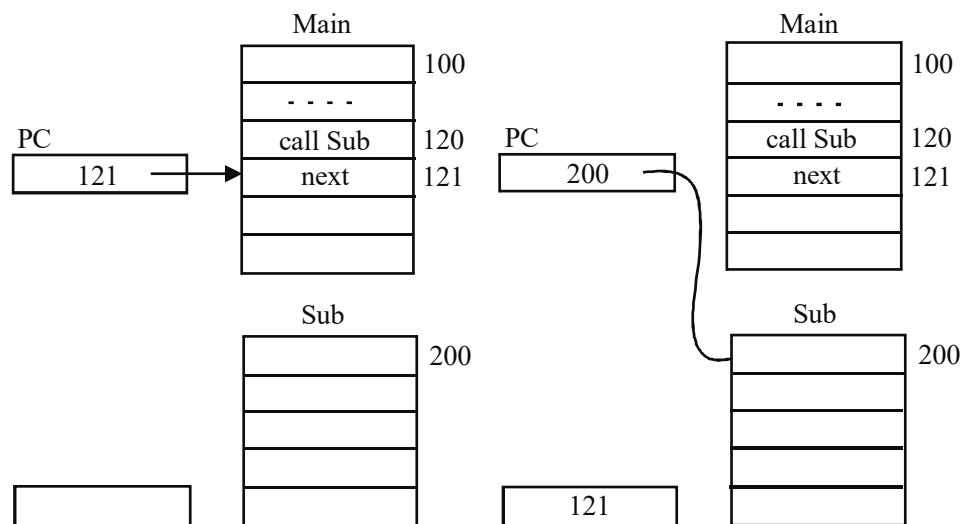


Fig. 4.40 Execution of Subroutine Using Stack

If a subroutine uses a data or address register, it will overwrite any 'old' data in it. If you do not want this to happen, you must save the contents of the register on entry to the subroutine and restore its contents on exit.

Following are certain properties of a subroutine:

- The subroutine call and return mechanism is automatic; therefore programmer will not execute it explicitly.
- A subroutine can be called from more than one program. The top of stack will store the return address of the program calling it.
- A subroutine call can appear in a subroutine, i.e., it can be nested.

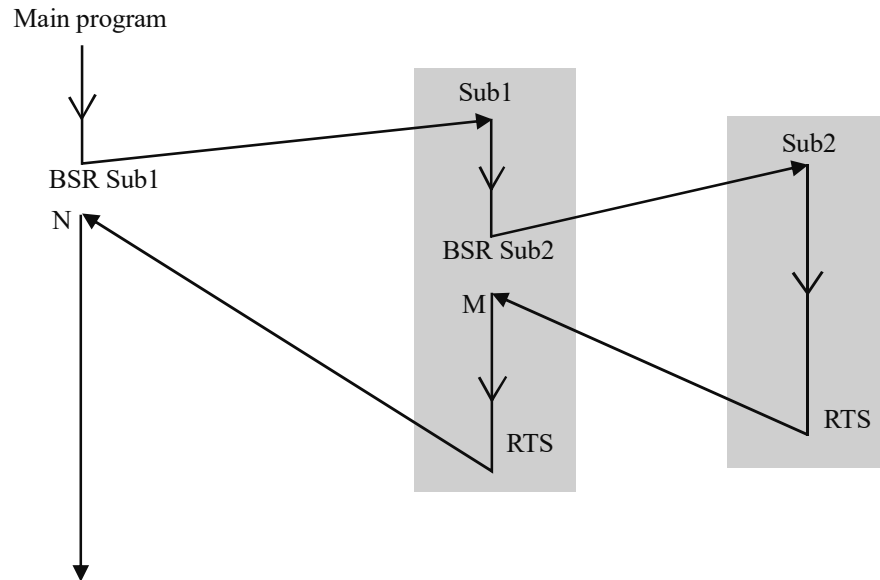
## NOTES

**NOTES**

**Understanding Nesting of a Subroutine Call**

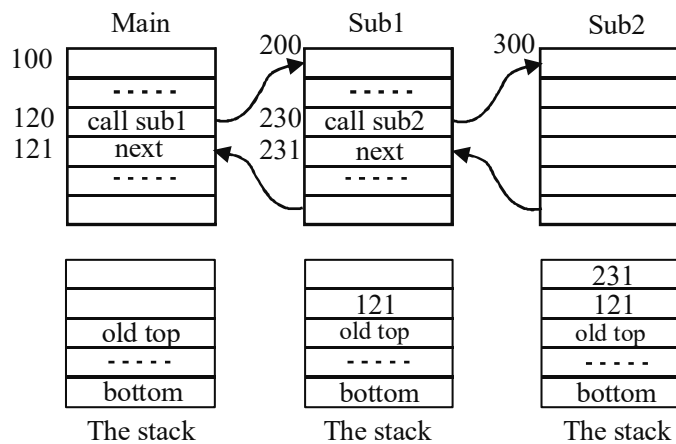
We can understand the nesting of a subroutine call with the help of an example. The main program calls a subroutine Sub1 which, in turn, calls another subroutine Sub2. When a called subroutine is finished, go back to the calling routine.

Figure 4.41 demonstrates how subroutine Sub1 is called from the main program and how Sub1 itself can call a second subroutine Sub2.



**Fig. 4.41** Nested Subroutine

Suppose Sub1 starts at address 200 and Sub2 starts at 300. When Sub1 is called, the stack pointer will store 121 (the instruction in main program to be executed once Sub1 is over). When Call Sub2 is made in Sub1 at address 230, the top of stack will store 231 (next address). Once Sub2 is completed, the program counter will have 231 and Sub1 will continue. On completion of Sub1, main program will continue from location 121. The implementation of nested subroutine using stack is shown in Figure 4.42.



**Fig. 4.42** Implementation of Nested Subroutine using Stack



## Features of Subroutine Nesting

Following are the features of subroutine nesting:

- Nesting Subroutines can be recursive, i.e., a subroutine can call itself.
- Subroutine may involve Parameter/Result Passing: If function Sub need some CPU registers whose contents will be used by Main program again after Sub is executed. Hence, it is required to save the content of these registers somewhere before Sub and then restore after Sub. This passing of data can also be implemented using some registers or main memory location. But this approach may prevent the use of reentrant subroutine. Hence, an easy solution is to use stack. Let us see how stack implements it. There are two parameters as follows:
  - o Pass parameters (variables) to be used in the function or their addresses from Main to Sub.
  - o Return the results from Sub to Main.

When the parameters have to pass, the stack pushes the return address. Also, the parameters are pushed in the stack. The subroutine can access these parameters from stack. Similarly, on return, the result can also be placed in the stack which will be used by main program using 'Pop' operation. The entire set of parameters, including return address and parameters (registers/memory location holding these parameters) passed, is referred to as stack frame (Refer Figure 4.43).

## NOTES

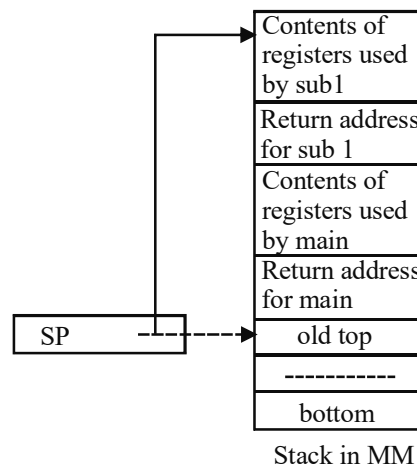


Fig. 4.43 Stack Frame

## 4.8.5 Direct Memory Access (DMA)

Direct Memory Access (DMA) is an important data transfer technique. In DMA, the data is moved between a peripheral device and the main memory without any direct intervention of the processor. Although DMA requires a relatively large amount of hardware and is complex to implement, it is the fastest possible means of transferring the data between peripheral device and memory. It reduces the CPU overhead as it requires no CPU involvement for continuous checking the

## NOTES

device status, leaving the CPU free to do other useful work. It grabs the data buses and address buses from the CPU and uses them for transferring the data directly between the peripheral device and memory. The CPU provides an address on the address bus specifying the memory location from where data is to be fetched or location where data available on data bus is to be written on memory. DMA uses a dedicated data transfer device that reads data coming from a device and stores it in buffer memory that can be retrieved later by the processor. The DMA technique is particularly useful for transferring the large amount of data, for example images, disk transfer, etc., to memory. The data transfer in such cases through programmed I/O is impractical. The transfer of small data packet through DMA is not considered very effective as there is lot of overhead for establishing a DMA connection. DMA requires additional hardware, such as a DMA controller, DMA memory partition(s) and a fast bus.

The major advantages of DMA over other the programmed and interrupt-driven I/O technique are as follows:

- Processor is not involved in I/O transfers in DMA. In other two techniques, on the other hand, each I/O transfer is performed by a set of instructions that are executed by CPU. So, with the DMA data transfer technique, the processor is available for other processing activities as it is not used for handling the data transfer activity. In the systems where the processor primarily uses cache, data transfer can take place in parallel, increasing overall system utilization.
- Only one or two bus read/write cycles are required per piece of data transferred in DMA as compared to other two methods where the rate with which I/O transfer can take place is limited by the speed by which processor tests the device and provides service.
- As there are dedicated hardware to respond more quickly than interrupts, DMA is able to minimize the latency in servicing a data acquisition from the device, which further reduces the amount of temporary storage (memory) required for an I/O device.
- If we compare the DMA data transfer to the interrupt-driven approach, there is no wastage of time in issuing a read command, waiting for device, generating interrupts by device and reading data, etc., is no longer required in DMA data transfer as it no longer has to set up the device and read from it. This results in a reduced overhead and increased throughput.

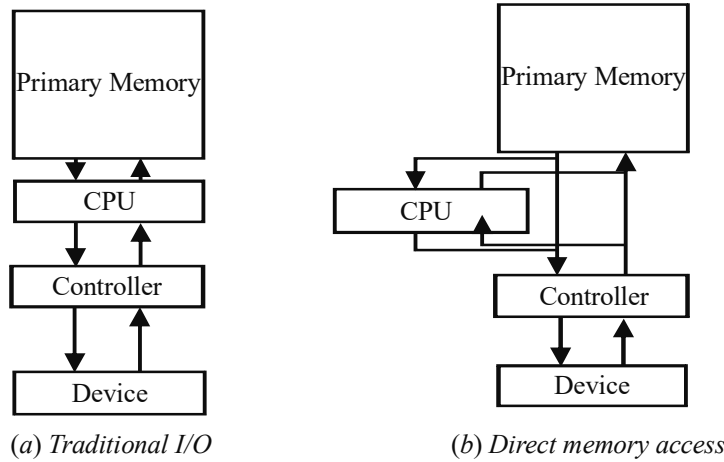


Fig. 4.44 Comparing the Functioning of Traditional I/O and DMA

Thus major part of CPU overhead is the time the CPU spends in reading operation as traditional case now overcome as shown in Figure 4.44. It is allowed to use system bus when processor does not need it or to temporarily force processor to suspend operations. This suspension of the process is called cycle stealing.

Figure 4.45 shows the structure of CPU bus control signals.

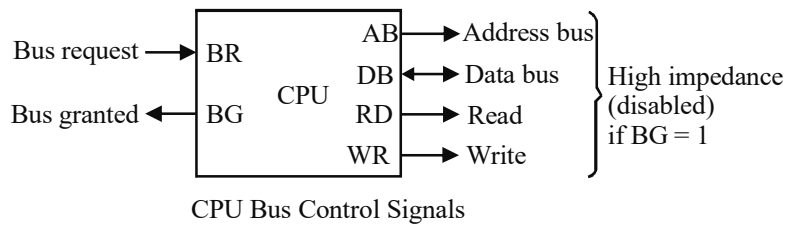


Fig. 4.45 Structure of CPU Bus Control Signals

To initiate a DMA transfer, the host writes a DMA command block. The block contains a pointer to the source and destination of the transfer and the number of the bytes to be transferred. The address of this command block is written to the DMA controller by the CPU. Once the CPU requests, the 'Request' bit will be set for that specific block. After DMA controller detects a request, it starts data transfers, which gives the CPU an opportunity to perform other tasks. Once the DMA reads all the data, only one interrupt is generated per block and CPU is notified that the data is available at the buffer.

On comparing DMA with programmed I/O we find that overhead is negligible. As CPU is no longer responsible for setting up the device, checking if the device is ready after the read operation and processing the read operation itself, we have 0 overhead. By using DMA, the bottleneck of the read operation will no longer be CPU. Now the bottleneck is transferred to the PCI BUS. Decrease in overhead results in much higher throughput, approximately 3–5 times higher than programmed I/O.

NOTES

Following are three possible ways of organizing DMA module using detached bus or integrated bus or separate I/O bus:

## NOTES

### (i) Single Bus: Detached DMA Module

- Each transfer uses bus twice: one from I/O to DMA and the other from DMA to memory.
- Processor is suspended twice.

### (ii) Single Bus: Integrated DMA Module

- Module may support more than one device.
- Each transfer uses bus only once, from DMA to memory.
- Processor is suspended once.

### (iii) Separate I/O Bus

- Bus supports all DMA enabled devices.
- Each transfer uses bus only once, from DMA to memory

## 4.8.6 Input/Output Processor

Till now we have studied the various modes for data transfer which involve the CPU. As the I/O Processor (IOP) is slow and wastes maximum of processor's time we can deploy one or more external processors and assign them the task of communicating directly with I/O devices without intervention of CPU. An IOP may be classified as a processor with the direct memory access capability that communicates with I/O device. As shown in Figure 4.46, such a processor has one memory unit and a number of processor which include CPU and one or more IOPs. IOP's responsibility is to handle all input/output related operations and relieve the CPU for other operations. The processor that communicates with remote terminals like telephone or any other serial communication media in serial fashion is called Data Communication Processor (DCP).

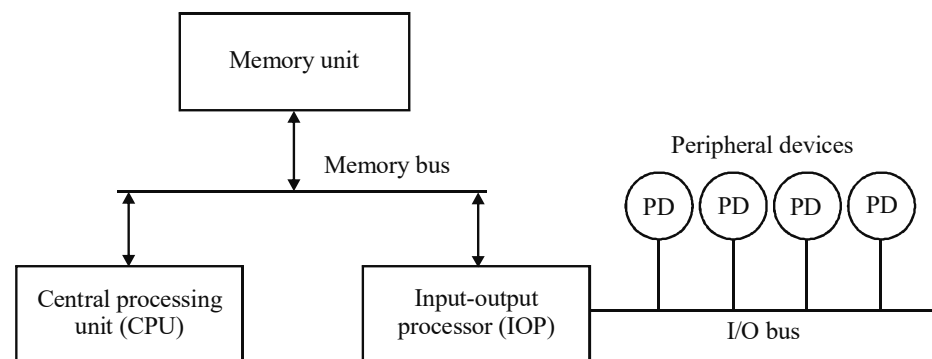


Fig. 4.46 Block Diagram of IOP

Figure 4.46 shows the block diagram of computer having an IOP. An IOP is just like a CPU. It can fetch and execute its own instruction. It is designed to handle all details of I/O processing. IOP can perform other processing tasks, such as arithmetic, logic branching and code translations. It provides the path for data transfer between various peripheral devices and memory unit. The CPU assigns

the task of initiating the I/O operation by testing the status of IOP. If status is fine, the processor continues its other works and IOP handles the I/O operation. After the input is completed, IOP transfers its content to memory by stealing one memory cycle from CPU. Similarly, an output is directly transferred from memory to IOP, stealing a memory cycle and from IOP to the output device at a rate the device accepts the output (Refer Figure 4.47).

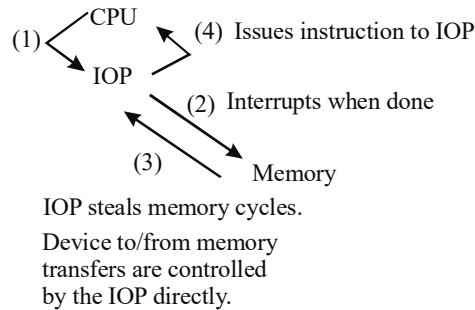


Fig. 4.47 Data Transfer between IOP and CPU

Instructions that are used for reading from memory by an IOP are called commands (instructions words are used as CPU instructions). The CPU informs IOP where the command is in memory and when it is to be executed (Refer Figure 4.48).

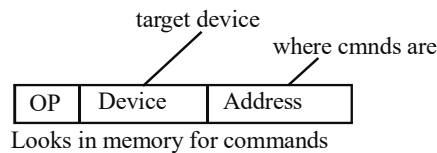


Fig. 4.48 CPU Command for Memory

The command word constitutes the program for the IOP. It informs IOP what to do, where to store data in memory, how much data transfer to take place and any other special request (Refer Figure 4.49).

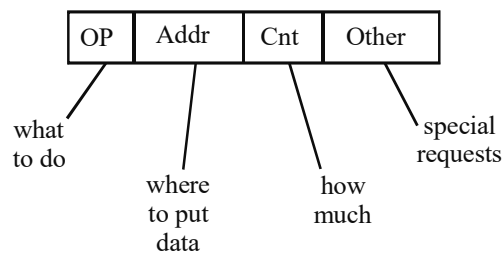


Fig. 4.49 IOP Instruction

In most computers, a CPU acts as a master and IOP as slave. The I/O operations are started by CPU but are executed by IOP. CPU gives the start command to start the I/O operation after testing the status. The status words indicate the conditions of the IOP and I/O devices, such as overload condition, device busy or device ready status, etc. Once it finds that status bit is O.K, the CPUs send the instruction to IOP to start the I/O transfer. The memory address received from the instruction tells the IOP where to find the program. The CPU continues with another program, while IOP is busy with the I/O program. Both programs refer to memory by means of DMA transfer. The IOP interacts with CPU by means of interrupt.

## NOTES

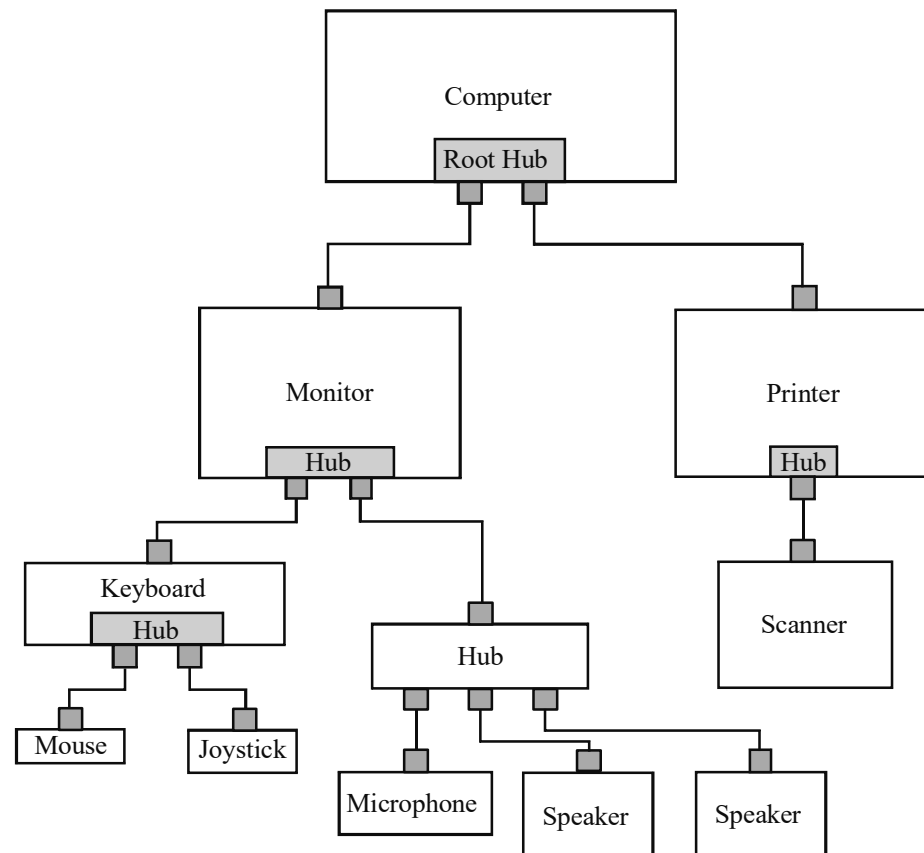
Also, for ending the instruction IOP, an interrupt is sent to CPU. The CPU responds to the interrupt by checking the IOP status to find whether the complete transfer operation takes place with or without error.

## NOTES

### 4.8.7 Serial Communication

For data communication with a remote device, a special data communication processor is used.

The data communication processor is an IOP that distributes and collects data from the remote terminals through telephone or other connection lines. It is a specialized I/O processor designed to communicate directly with data communication network. A communication network may consist of wide range of devices, such as printer, display device, sensors, etc. Using data processor communication, the computer can serve fragments of each network demand in an interspersed manner. Thus it appears it is serving many users at once. The main difference between the IOP and DCP is that the IOP communicates with peripherals through a common bus that is consisted of many data and control lines, while in DCP each terminal is attached with a pair of wire. Thus, in IOP all peripherals use a common bus and to transfer information to and from the processor. The DCP communicates with each terminal through a pair of single wires. Both data and control information are transferred in serial fashion that results in a much slower transfer. It is DCP's task to collect and transfer data to and from each terminal and also to ensure that all requests are taken care of according to the predetermined procedure. Figure 4.50 shows an example serial transmission.



**Fig. 4.50** An Example of Serial Transmission

One common example of DCP is modem. It is used for establishing connection between the computer and telephone line. As telephone lines are designed for analog signal transfer, a modem should convert the audio signal of telephone line to digital format for computer use and also convert the digital signal to audio signal that is to be transmitted through communication line.

The transmission can be synchronous or asynchronous depending upon the transmission mode of the remote terminal. The synchronous transmission does not use start and stop bits. This is commonly used in high-speed device to realize full efficiency of communication link. The synchronous message is sent as a continuous message for maintaining a synchronism. In modems, internal clocks are set to the frequency of communication line. In this case the receiver clock has to be maintained continuously for adjusting any frequency shift. In asynchronous transmission, on the other hand, each character can be separately with own start and stop bit. The message is sent as group of bits as block of data. The entire block is transmitted with a special control characters at the beginning and end of the block as shown in Figure 4.51. SYNC is used for synchronous data, PID is process ID, followed by message (packet), CRC code and EOP indicating end of block. One function of the data communication processor is to check the transmission errors. CRC cyclic redundancy check a polynomial code algorithm is used to check the error occur during transmission.

SYNC	PID	Packet Specific Data	CRC	EOP
------	-----	----------------------	-----	-----

*Fig. 4.51 Data Format*

### Check Your Progress

14. Define a magnetic tape.
15. What does a sensor mean?
16. What is MICR?
17. State a problem with the I/O device management.
18. Write some properties of subroutines.
19. Write an advantage of DMA over programmed and interrupt driven I/O.
20. Define DCP.

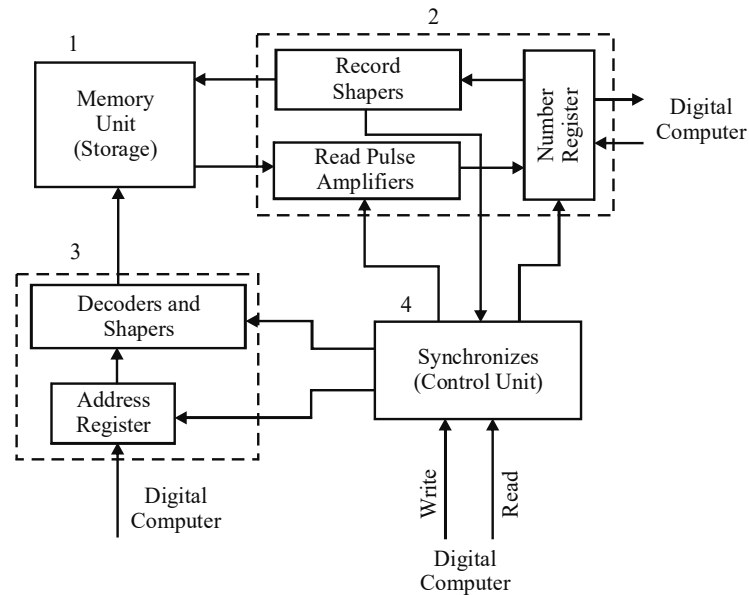
## 4.9 MEMORY UNIT

Many computer systems have a memory hierarchy consisting of CPU registers, Static Random Access Memory (SRAM) caches, external caches, Dynamic Random Access Memory (DRAM), paging systems and virtual memory or swap space on a hard drive which collectively refers to the structure of large memories. This entire pool of memory is referred to as RAM by many developers. Within a hierarchy level, such as DRAM, the various interleave organization of the components make the access time fast. The goal of structuring the large memory is

### NOTES

**NOTES**

to obtain the higher possible average access performance while minimizing the total cost of the entire memory system. Memory units are used to store data whose composition does not change during the processing of a specific type of information or the solution of one class of problem, such as tabular data, standard subprograms and the unchanged programs of control computers.



**Fig. 4.52** Block Diagram of Memory Unit

Figure 4.52 illustrates addressed memory units which include large storage capacity (1), a numerical section (2), an address section (3) and a local control block or synchronizer (4). The storage consists of memory cells and performs the functions of data storage. The numerical section or write read circuit is the intermediate link in which data exchange takes place between the storage and devices which are external with respect to the memory unit. It consists of a number register for temporary storage of words or numbers in terms of **readin** or **readout** shapers which convert the numerical code into a series of signals recorded by the storage cells. The readout pulse are used for amplifying **readout** signals cancelling and shaping the noise. In the address part of the memory unit, the assigned address code is converted into a set of signals that unambiguously define the required storage location. The synchronization block forms internal commands that effect control of the sequence of operation of all assemblies of the memory unit in accordance with incoming commands. The aggregate of all blocks of the memory unit except the storage is called the electronic control circuit or the peripheral equipment or the electronic framework of the memory unit. The structure of memory unit can be enlarged by:

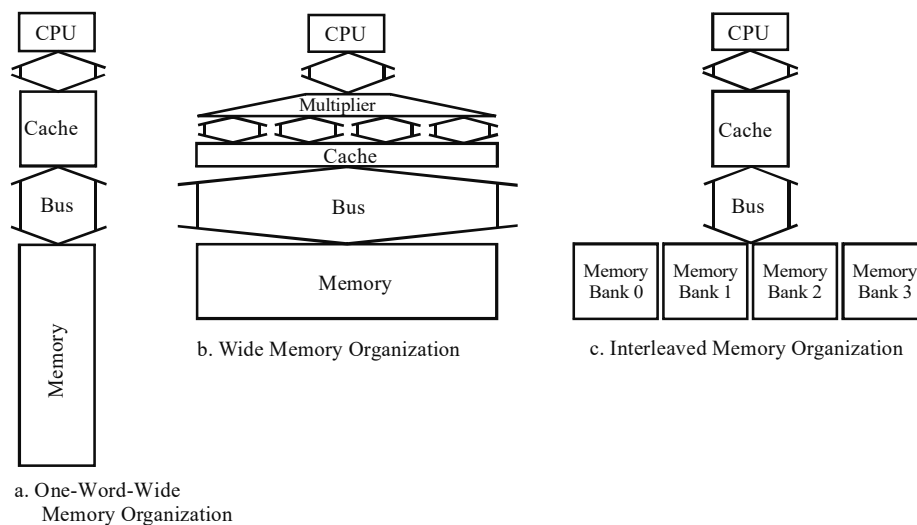
- Reserving, committing and freeing virtual memory.
- Changing protection on pages of virtual memory.
- Locking pages of virtual memory.
- Querying a process's virtual memory.

In many modern Personal Computers (PCs), the RAM comes in an easily upgraded form of modules called memory modules or DRAM modules. These can quickly



be replaced when changing needs demand more storage capacity. Caches are memory circuits which work with RAM by accelerating the task of moving data to the CPU. Technology works behind virtual memory refer to enlarge the memory capacity of computer. Virtual memory system came into existence when large memory is required in data transmission and of development computer programs. At times programs are so complex and big that they are not able to fit in the specified memory. The better option is to divide the programs into subprograms. These subprograms are known as overlays. The overlays are swapped dynamically in the memory processed by operating system. Virtual Memory System (VMS) was first devised by Fotheringham in 1961. This system was developed because stack, data and programs together exceed from required physical memory and hence excess memory is required. Virtual memory use cache in physical DRAM in the system unit. The virtual memory address space may exceed physical memory size. The VMS simplifies the complete memory management which resides in the main memory. The involved process has own address space. Virtual memory system provides protection because one process can not intervene with other process because both processes operate in different address spaces.

## NOTES



**Fig. 4.53** Structure of Memory System in Large Memory

Figure 4.53 shows how memory system is organized collectively in the disk, processor, physical memory, virtual memory, MMU and operating system which can store huge data, such as multimedia streaming files. Alphabet 'b' represents the wide memory organizations which include multiplexer, cache, CPU, bus, etc., to work with interleaved memory organization which is referred by alphabet 'c'. Alphabet 'a' refers to one word wide memory organization. In the VMS, the page table plays an important role to manage the memory. The disk keeps all the system files and the programs to run successfully the OS. Virtual memory is used to manage if there is shortage of memory. If processes are very complex and big then VMS is needed for efficient protection scheme. It is also used to maintain paging system and sharing the processes. The piece of virtual memory is generally loaded from hard disk drive and referenced as stack. The rarely used piece of virtual memory can be discarded out to disk. Basically, VMS is defined as an abstract space instead physical device and is comprised of virtual address space including all

## NOTES

processes. Some system use virtual address space as a set of virtual addresses or some might use a single virtual address space to all processes. Having large memory in CPU includes the features, such as swapping, protection, Translation Lookaside Buffer (TLB), etc., which are used in operating system to share the large memory system.

### Swapping

The VMS use swapping that is also called paging for Windows OS (Operating System). The WIN386.SWP on Windows 3.11 is created by Windows 3.x. It is in fact a hidden file and used as virtual memory swap file. It resides on root directory and its size depends on setting of control panel's, such as how much virtual memory is used for OS. In Windows 95 this swap file is located at 'C:\pagefile.sys' on all Windows NT versions of Windows. In Linux OS, the whole partition in HDD can be used for VMS. A swap area is created by issuing command 'mkswap filename/device' which is accompanied by swap file name or swap partition. It is suggested that swap partition is done at the very beginning of the hard disk drive because it increases the transfer speed of the data. Swapping is used on the video cards that have sometimes 128 or even 256 MB of RAM. Pages are defined as the division units of virtual address space. The virtual address space is divided into units known as pages. Sometimes, virtual page field is replaced by a page frame field due to the following reasons:

- If the page table would be extremely large.
- If memory mapping would be very fast.

### Protection

Virtual addresses give memory protection by using multiprogramming technique. Bugs in any program can cause other program to crash and even OS. VMS keeps user programs from crashing one another and the OS. It uses to protect OS by using address translation and dual mode operation. The information of address translation is accessed by page table entry. Hardware enforces this protection trap into OS if violation occurs. The two processes namely, 'Process i' and 'Process j' are allotted in page tables that shows the preemptive multitasking. The translation tables offer protection if they are not altered by applications. Sometimes, malicious programs construct a new instruction every time when the programs load into memory. The programs which are written to control the malicious programs reside into the memory and they trap the code in memory. It makes a difference between original program and infected programs from the protection code using Program Status Word (PSW) key. The protection code is an interruption that is written in main memory's region which includes subsidiary memory. It is connected with CPU and the main memory. The two registers, base and limit registers are used to implement the techniques. The PSW refers the programs to the process registers, program counter, accumulator, etc., which are involved in the execution of each computer program. The protection code checks subsidiary memory number and the produced instruction. Both are accessed directly from CPU. Protection code can be written in the stack region and information execution in the main memory and are transmitted beyond the CPU.

## Translation Lookaside Buffer

Memory Management Unit (MMU) contains TLB that translates virtual page number to physical page number. In fact, TLB is hardware cache. It contains page table entries. Each virtual memory reference can cause two way of accessing the physical memory and they are known as fetching the page table and fetching the data. The TLB lookup contains recently used page table entries. The processor at this stage investigates TLB that contains a given virtual address. The present page table entry is known as TLB hit containing retrieved frame number and real address of pages. The missing page table entry is known as TLB miss containing index of page table. The page fault occurs pages are not available in the main memory. Basically, a page fault is an interrupt or exception to the software raised by the hardware, when a program accesses a page that is mapped in address space but not loaded in physical memory. TLB can be updated for new page entry.

Virtual memory amount is considered as abstract design. Virtual RAM is a progression used by the operating windows system to optimize in addition to increase the storage size in addition to electric of its running RAM without the need of adding different equipment. Virtual memory system management follows some policies to maintain the large memory system for communicating with various processes. The policies are discussed below:

- **Allocation Policy:** The allocation policy follows variable allocation along with global scope is easy to be adopted by many operating systems. The list of free frames is followed by the allocation policy. If page fault occurs the set of process works accordingly. For this, free frames are replaced for other processes.
- **Fetch Policy:** Fetch policy is used to bring the page into memory. If reference is indexed for location of page then demand paging is used to bring into main memory. When process starts first time many page faults occur.
- **Replacement Policy:** The replacement policy follows the frame locking mechanism. The frame locking policy involves locked frames that can not be replaced in the kernel of OS. The I/O buffers and control structures are associated with each frame containing a lock bit. The used bit is set as 0 if it is loaded into the memory the first time. The used bit is set as 1 if the very first frame gets used bit 0. This is called replacement algorithm where each used bit 1 is set as used bit 0.
- **Placement Policy:** Placement policy involves the real memory process. It is important in segmentation system. The address translation is involved in paging with hardware segmentation. Basically, it determines the memory space where incoming new pages are loaded into the physical memory. Placement policy checks the tasks, such as which page can be replaced and which page can be kept for future reference. This policy can predict the page reference on the basis of segmentation system.
- **Cleaning Policy:** The cleaning policy follows demand cleaning and pre-cleaning for managing the VMS where a page is selected for replacement. The cleaning policy is used for page buffering where modified and unmodified replaced pages are involved. The modified pages are periodically written in batches and unmodified pages are written if the pages are to be reclaimed or referenced. The referenced pages can be accessed if frames are assigned to other pages.

## NOTES

## NOTES

### 4.9.1 Types of Memory

Memory is an indispensable part of computer and microprocessor based systems. The data used in a program as well as the instructions for executing the program are stored in the memory. Hence, digital systems require memory facilities for temporary as well as permanent storage of data to perform their functions. A flip-flop stores one bit of information, a register is able to hold a word and a register file holds a modest number of words of information.

The very first computer memory consisted of a minute magnetic toroid, called *core memory*, which required large and bulky circuit boards stored in large cabinets. Semiconductor memory, on the other hand, is very compact and can be accessed at very high speeds and is capable of storing data in extremely high densities. All modern computers and microprocessor systems have been made possible by the development of inexpensive and reliable Very Large Scale Integration (VLSI) semiconductor memory chips utilizing Negative-channel Metal Oxide Semiconductor (NMOS), Complementary Metal Oxide Semiconductor (CMOS), Bipolar Junction Transistor (BJT) and Bipolar CMOS (BiCMOS) technologies.

Information from magnetic and optical storage devices, such as hard disk, floppy disk, CD ROM and digital tape must be accessed sequentially, starting at the beginning of a data file or track. In contrast, data stored in an electronic memory cell can be accessed at random and on demand using direct addressing. Direct addressing eliminates the need to process a large stream of irrelevant data in order to find the desired data word.

The evolution of Programmable Logic Device (PLD) began with Programmable Read Only Memory (PROM). A ROM is a memory device that consists of AND and OR arrays which can be programmed by the user to implement combinational and sequential functions. For reprogrammability, PLDs use EPROM or EEPROM like cells. Generally, PLDs may be classified depending upon the programmability of the AND and OR arrays. PLDs with programmable AND and fixed OR arrays are called Programmable Array Logic (PAL) devices. When both the AND and OR arrays are programmable, such PLDs are known as Programmable Logic Arrays (PLA).

The programmability and high density of PLDs make them useful in the design of Application Specific Integrated Circuits (ASICs) where design changes can be made rapidly and inexpensively.

A Field Programmable Gate Array (FPGA) is a reprogrammable gate array that uses antifuse. The differences between FPGA and PLD is that FPGA incorporates logic blocks instead of fixed AND – OR gates and is faster with low power dissipation. Following are the types of memory system which work collectively to perform system operations.

#### ROM

A Read Only Memory (ROM) is a semiconductor memory device used to store the information permanently. It performs only read operation and does not have a write capability. A ROM is programmed for a particular purpose during the manufacturing process and the user cannot alter its function. ROM circuits are

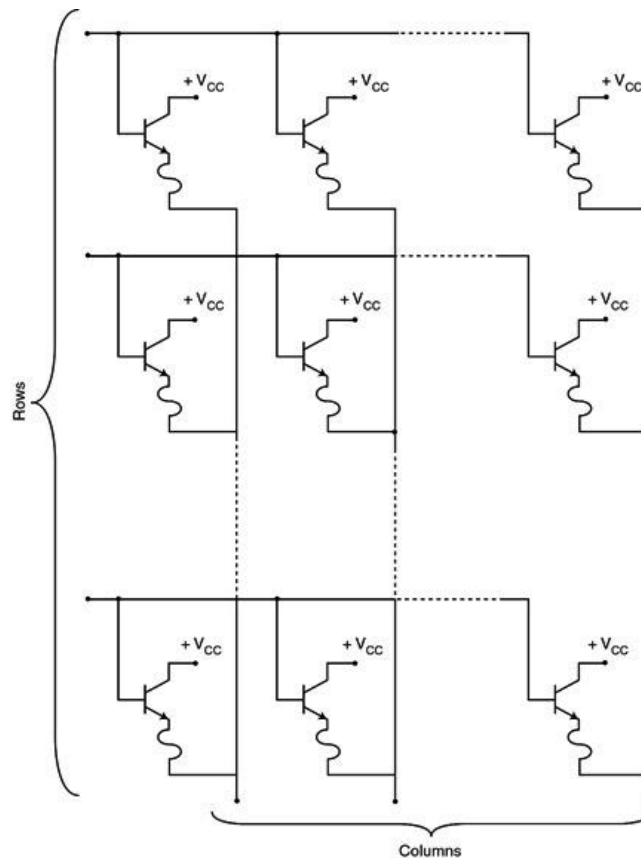
typically used to provide the computer with resident programs and key operating functions needed to boot the operating system of the computer.

The ROM is a combinational logic circuit. It includes both the decoder and the OR gates within a single IC package. In order to minimize the number of address lines, decoders are used. The address of the desired line is given in binary. The connections between the output of the decoder and the input of the OR gates can be specified for each particular configuration. The ROM is used to implement complex combinational circuits within one IC package or as permanent storage package for binary information. The binary information must be specified by the designer and is then embedded in the unit to form the required interconnection pattern.

**NOTES**

**PROM**

In order to provide some flexibility in the possible applications of ROM Programmable ROMs (PROMs) have been introduced. The PROM can be programmed electrically by the user but cannot be reprogrammed. In a PROM chip the manufacturer includes a connection at every intersection of the grid of address and data lines. PROMs are widely used in the control of electrical equipment, such as washing machines and electric ovens.



*Fig. 4.54 Bipolar PROM Array with Fusible Links*

PROMs are available in both bipolar and MOS technologies. Figure 4.54 shows bipolar PROM array with fusible links. A fusible link is a tiny fuse that can

## NOTES

be manufactured as per requirement. PROMs are fabricated using bipolar transistors. Vcc pins are tied together in bipolar PROM array. PROMs have 4-bit or 8-bit output word formats with capacities ranging in excess of 2,50,000 bits. A PROM is manufactured as a generalized integrated circuit with all the matrix intersections linked by fusible diodes or transistors. In series with each connection, the manufacturer includes a fusible link which can be melted and thereby opened, by passing a large current through it. A memory link is fused open or left intact to represent a binary number 0 or 1. The user can selectively burnout some of the fuses by passing enough current through them and can thereby program the PROM according to the required truth table. Once a PROM is programmed, it cannot be changed and therefore it has to be done carefully and correctly in the first time itself. Hence, the fusing process is irreversible.

### EPROM

A PROM device that can be erased and reprogrammed is called EPROM. It uses an array of n-channel enhancement type Metal Oxide Semiconductor Field Effect Transistors (MOSFETs) with an insulated gate structure. Figure 4.55 shows the basic structure and symbol of a typical EPROM cell. Here, an additional floating gate is formed within the silicon dioxide ( $\text{SiO}_2$ ) layer. The floating gate is left unconnected while the normal control gate is connected to the row decoder output of EPROM. The data bits are represented by the presence or absence of a stored charge. The initial values of unprogrammed EPROM cells may be all 0s or all 1s.

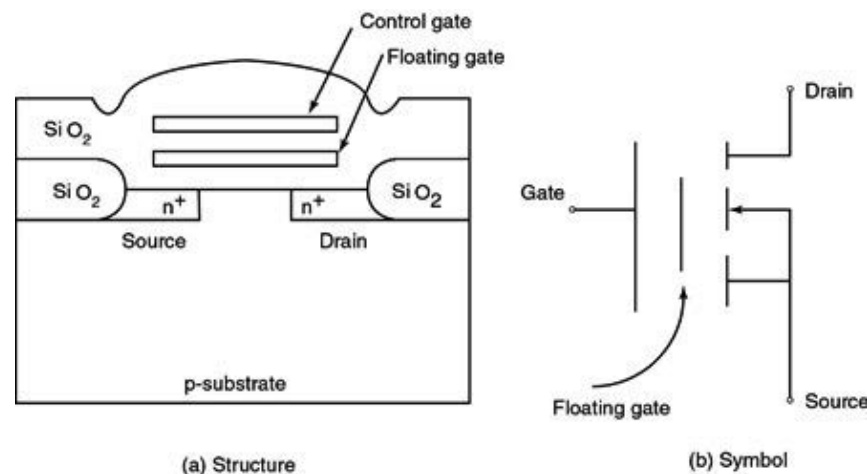


Fig. 4.55 EPROM Cell

### EEPROM

Another type of reprogrammable ROM device is Electrically Erasable Programmable ROM (EEPROM) which is also known as Electrically Alterable Programmable ROM (EAPROM). The EEPROM overcomes the disadvantages of EPROM. It EEPROM can be erased and programmed by the application of controlled electric pulses to the IC in the circuit and thereby changes can be made in the selected memory locations without disturbing the correct data in other memory locations. EEPROM is non-volatile, for example EPROM but does not require ultraviolet light to be erased. The non-volatility of EEPROM permits a system to be immune to power interruptions.

EEPROM is a rugged and low power semiconductor device and it occupies less space. It has the advantages of program flexibility, small size and semiconductor memory ruggedness, i.e., low voltages and no mechanical parts. The requirement of low power supports field programming in portable devices for communication encoding, data formatting and conversion and program storage. With EEPROM, the programs can be altered remotely and possibly by telephone.

### 4.9.2 Flash Memory

Flash memory is a non-volatile computer storage chip that can be electrically erased and reprogrammed. It was developed from EEPROM (Electrically Erasable Programmable Read Only Memory) and must be erased in fairly large blocks before these can be rewritten with new data. The two types are NAND and NOR. The high density NAND type must also be programmed and read in (smaller) blocks or pages while the NOR type allows a single machine word (byte) to be written and/or read independently. The NAND type is primarily used in memory cards, USB flash drives, solid state drives and similar products, for general storage and transfer of data.

Thus, a flash memory is an electronic flash memory data storage device which is used in digital cameras, mobile computers, telephones, music players, video game consoles and other electronics. It is designed in a form of digital device with excellent speed performance and capacity. It keeps a write protection switch for holding all the written contents. The card used in flash memory generally contain a grid of columns and rows having two transistors at each intersection. These two transistors are also known as gates. One transistor is known as floating gate and other is known as control gate. The flash memory cards come in various designs and configurations ranging from 32 Megabytes to 16 Gigabytes. The most common flash memory card is Secure Digital (SD) flash memory card which is used frequently as devices in various electronic gadgets. The SD flash memory offers high storage capacity, fast data transfer, great flexibility in data transfer, excellent security and very small size about the size of a postage stamp. They are used in digital cameras, digital camcorders, Internet music players and recorders, audio players, Personal Digital Assistant (PDA), digital voice recorders, cellular phones, digital projectors, photo printers, e-books, etc. Figure 4.56 shows a flash memory card.

### NOTES



**Fig.4.56** Flash Memory Card

Table 4.11 lists the types of flash memory cards and their functions.

*Table 4.11 Types of Flash Memory Cards and their Uses*

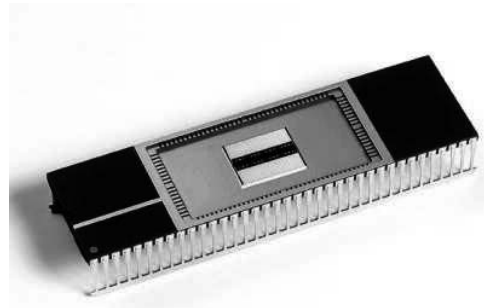
**NOTES**

Types of Flash Memory Card	Size/Function	Use
Compact Flash Card	It works as Advanced Technology Attachment (ATA) drive.	In digital cameras.
Data Flash Card	It is a type of removable flash card.	In digital cameras and laptops.
Memory Flash Stick	Its size is 50 mm × 21 mm × 2.8 mm.	In video game consoles.

Following types of flash memory cards are available for use:

**The NOR Flash Memory Card**

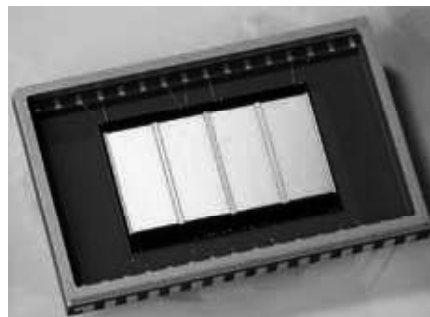
The NOR flash memory card is used for programming code execution containing cells. Each cell stacked vertically resembling a standard MOSFET and connected with bitline. Figure 4.57 shows a NOR flash memory card.



*Fig. 4.57 The NOR Flash Memory Card*

**The NAND Flash Memory Card**

The NAND memory card is assembled with host in interface mode. This type of card includes the NAND flash memory to convert the interface mode of the host. It is connected with bit line in a series with 16 or 32 memory cells providing low cost media for picture storage. Figure 4.58 shows a NAND flash memory card.



*Fig. 4.58 NAND Flash Memory Card*

In flash memory cards, two architectures NOR setup and NAND setup are very significant. The NAND setup is being used in cellular phones.



### 4.9.3 Associative Memory

Associative memory can be defined as memory unit accessed by content. It is also known as Content Addressable Memory (CAM).

When a word is written on an associative memory, no address is given. The memory is capable of finding an empty location to store the word. If a word is to be read from an associative memory, the content of the word or part of the word is specified. The memory locates all the words that are identical to the identified content and marks them for reading.

This type of memory is accessed simultaneously and is equivalent to the information instead of the specific address or location.

Associative memory is more costly than random access memory as every cell should have storage capacity as well as logic circuits for matching its content with an external argument.

Thus, associative memories are utilized in applications where the search time is very important and must be very short.

#### Hardware Organization of Associative Memory

The block diagram of associative memory is shown in Figure 4.59. It consists of a memory array and logic for  $m$  words with  $n$  bits per word. The argument register  $A$  and the key register  $K$  each have  $n$  bits, one for each bit of a word. The match register  $M$  has  $m$  bits, one for each memory word. Each word in the memory is compared in corresponding with the content of the argument register. The words that match the bits of the argument register set the corresponding bit in the match register. After the matching process, those bits in the match register that have been set indicate the fact that their corresponding words have been matched.

Reading is accomplished by a sequential access to memory for those words whose corresponding bits in the match register have been set.

For selecting a particular field or key in the argument register, the key register offers a mask. The entire argument is compared with each memory word if the key register contains all 1s. Otherwise, comparisons of just those bits in the argument that have 1s in the parallel position of the key register are done.

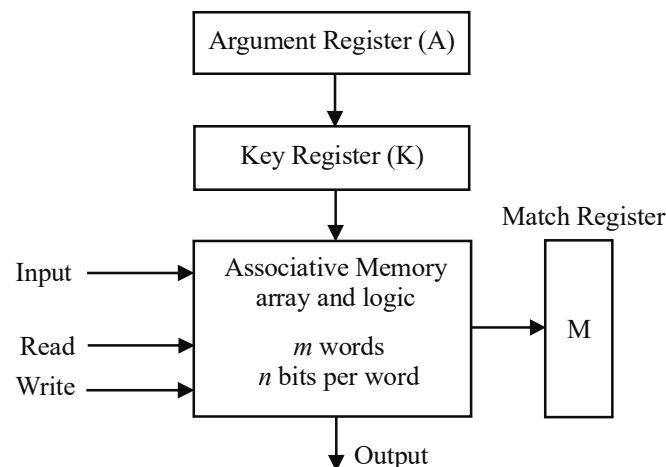


Fig. 4.59 Block Diagram of Associative Memory

#### NOTES

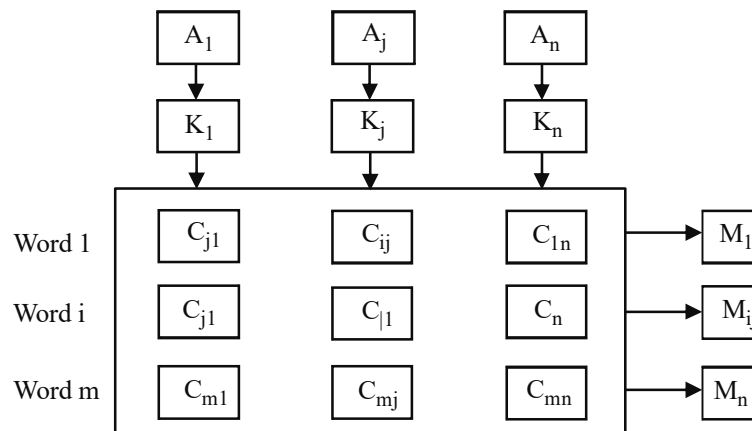
**NOTES**

Suppose, A	110 001110	
K	111 000000	
Word 1	101 001110	no match
Word 2	110 000001	match

Only the three leftmost bits of  $A$  are compared with memory words because  $K$  has 1s in these positions. The three leftmost bits of argument and Word 2 are equal. Hence, Word 2 is matched with the unmasked argument field.

Figure 4.60 shows how external registers are associated with the memory array of associative memory. The letter  $C$  with two subscripts marks the cells in the array. The first subscript gives the word number and the second specifies the bit positions in the word. Thus, cell  $C_{ij}$  is the cell for bit  $j$  in the word  $i$ . Bit  $A_j$  in the argument register is compared with all the bits in column  $j$  of the array, provided that  $K_j = 1$ .

If a match occurs between all the unmasked bits of the argument register and the bits in word  $i$ , the corresponding bit  $M_i$  in the match register is set to 1. If any unmasked bit of the argument does not match with the word, the corresponding bit in the match register is cleared to 0.



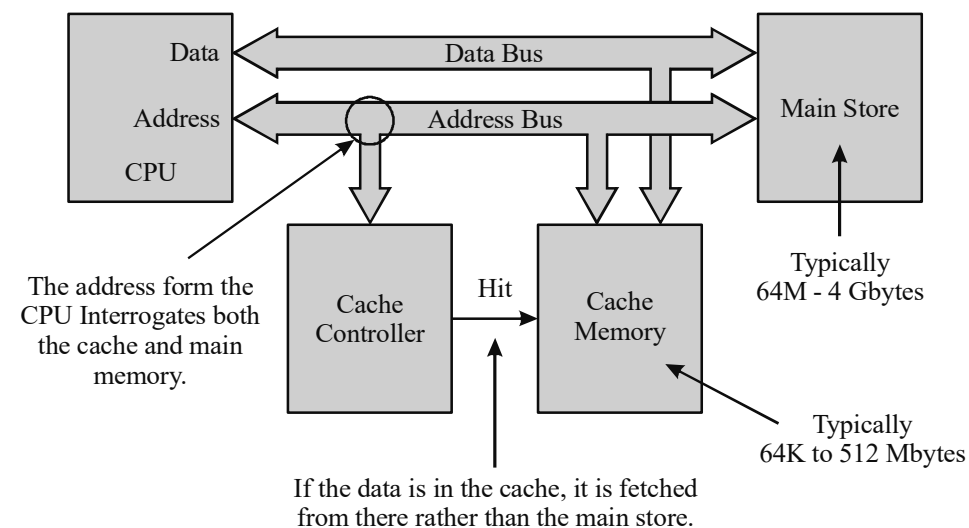
**Fig. 4.60** Associative Memory of  $m$  Words,  $n$  Cells per Word

### 4.9.4 Cache Memory

Cache memory is used to store the data and information temporarily. It is implemented for Internet content by distributing it to multiple servers which are periodically refreshed. In fact, cache is a small and fast memory placed between the CPU and the main memory as shown in Figure 4.62. The system performance can improve dramatically by using cache memory at a relatively lower cost. The word cache is derived from the French word that means hidden. It is named so because the cache memory is hidden from the programmer and appears as if it is a part of the system's memory space. It improves the speed because of its very high speed and rapidly been accessed by the processor with a fetch cycle time comparable to speed of CPU. The whole concept of using cache memory is based on the principle of hierarchy and locality of reference which you have already

studied. This results in an overall increase in the speed of the system. A system that uses a tiny 512 MB cache memory and RAMs of 2 GB, it is observed that the processor accesses to the cache 95 per cent more than RAM. The initial microprocessors had truly tiny cache memories, for example, 32 bytes. But in the early 1990s, the cache sizes of 8 KB to 32 KB became common. By the end of the 1990s, multilevel cache configuration became common. The multilevel chip has one cache of capacity up to 128 KB internal on the chip and other is external to chip and form second level caches having capacity up to 1 MB.

In Figure 4.61, it can be seen that the cache memory is attached to both the processor as well as main memory in parallel via address and data buses. This is done so that data consistency is maintained in both cache and the main memories.



**Fig. 4.61** Cache Memory Organization

According to the principle of memory hierarchy, the complete program resides on the hard disk and a few active pages of the current process (in case of large programs) reside in the main memory. A small part of the main memory is copied to the cache. It is the role of cache controller to determine whether the data desired by the processor resides in the cache memory or it is to be obtained from the main memory. The processor generates the address of a word to be read and sends it to address bus. The cache controller fetches the address and matches it with the content of cache. If the desired data is found in the cache, a Hit signal is generated and the word is delivered to the processor. However, if the data does not exist in cache then a Miss signal is generated and the data is searched in main memory. If data is found in main memory, it is delivered to the processor and is also simultaneously loaded into the cache. If data is not found in main memory, it is fetched from hard disk, as in case of virtual memory technique discussed earlier.

The amount of information which is replaced at one time in the cache is called the line size/block for the cache. It is usually a wider word than the CPU requires and is often equal to the width of the data bus connecting the cache and the main memory. A wide block size means that several data words are loaded from main memory into the cache at one time instead of single word this. It results

## NOTES

in prefetching of data.

NOTES

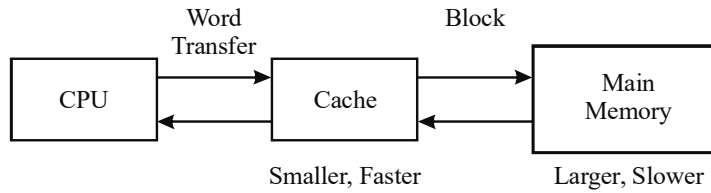


Fig. 4.62 Transfer of Information between CPU, Cache and Memory

Figure 4.62 shows a cache and main memory structure. A cache consists of  $C$  slots and each slot in the cache can hold  $K$  memory words. Thus, the cache stores  $C \cdot K$  words where  $K$  is the block size and  $C$  is the number of lines. Suppose the main memory stores  $2^n - 1$  words ( $M = 2^n - 1$ ) with each word having a unique  $n$ -bit address. Each word that resides in the cache is a subset of main memory. As only some portion of main memory can reside in the cache, as it is not possible to store the complete program in the cache, it is required to replace the existing word from cache and bring the new one whenever a Miss signal is reported. Thus no line can occupy permanently in the cache. To identify which particular block of main memory is currently residing in the cache, a tag is used for each line of cache. This tag is usually a portion of the main memory address. The cache memory is accessed by mapping the physical address with the tag stored in the cache.

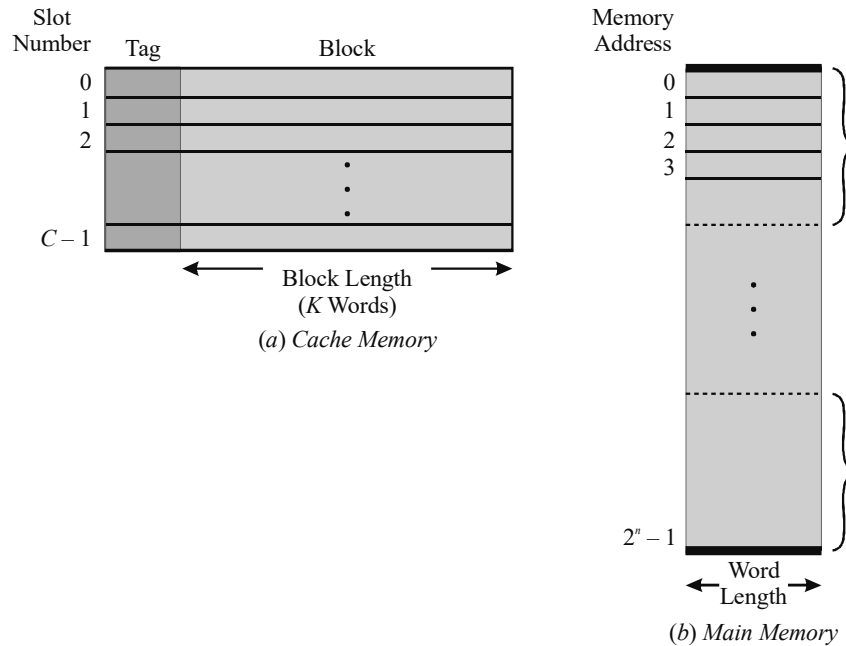


Fig. 4.63 Structure of Cache and Main Memory

For accessing the data for cache, it required to transform the main memory address to corresponding address in cache. This process is referred to as mapping. For mapping the address generated by CPU and the memory controller needs some algorithm. The result of mapping lets us know whether the data required by processor is available in the cache or not. As shown in Figure 4.63, cache memory is a linear array of some entries and each entry stores following information:

- **Data:** The block of data from main memory that is stored in a specific line in the cache.
- **Tag:** A small field of length K bits which is usually some portion of main memory address and is used as input for mapping process.
- **Valid Bit:** A 1-bit field that indicates the status of data stored in the cache.

**NOTES**

The address generated processor is divided into the following three fields to access the cache memory:

- **Tag:** A K-bit field that is compared with the K-bit tag field stored in each entry of cache.
- **Index:** An M-bit field that points to particular entry of cache.
- **Byte Offset:** L bits that find particular data in a line if valid cache is found.

Thus, the size of address generated by processor is given by  $N = K + M + L$  bits.

How this cache address translation take place is shown in Figure 4.64. Here the cache address generated by processor is of 32 bits in which Tag field occupies 12–31 bits, Index field occupies bits 2–11 and bits at 0,1 position contain the offset information. The index field tells us about the line of the cache which contains the data requested by the processor. Tag field of that line is retrieved from the cache and is compared with the tag field stored in the cache address generated by processor by using the comparator circuit. If the tag matches, we get the output of the comparator as one. The valid bit in the cache row pointed to by the index field of the cache address is tested to check the validity of data. The output of comparator and valid bit is ANDed if a Hit signal is generated and the data stored in the cached block corresponding to the byte offset is sent to the processor. If the output of comparator is zero, a cache Miss is reported.

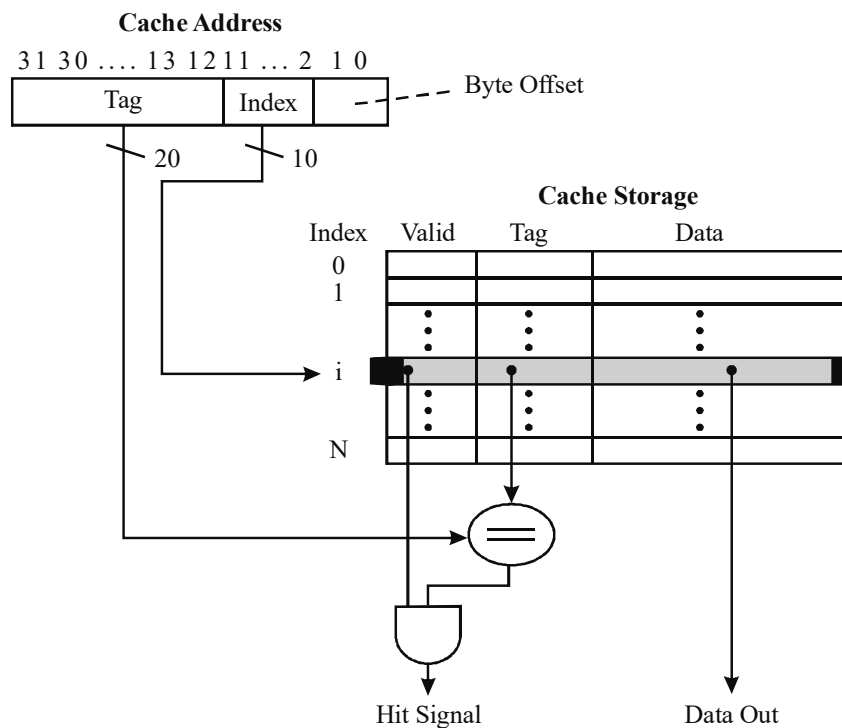


Fig. 4.64 Schematic Diagram of a Cache

## NOTES

### Performance Considerations

The performance of the cache is measured in terms of hit ratio. The hit ratio is number of hits divided by the total number of references a processor makes to cache memory (hits plus misses). The hit ratios of 0.9 and above have been reported. In a system, the hit ratio is determined by statistical observations. Apart from hardware technology, such as size of cache and the hit ratio is also software dependent, i.e., it depends on the nature of the program under execution. The value of hit ratio depends, on how effectively a program implements the principle of locality of reference. Hence, it is possible that some programs have very high hit ratios while others have low hit ratios. Let us calculate the effect of cache on the performance of the system.

Let us consider that a given system has:

$$\text{Access time of main memory} = t_m$$

$$\text{Access time of cache memory} = t_c$$

$$\text{Hit ratio of cache memory} = h$$

$$\text{Miss ratio of cache memory} = m$$

$$\text{Speedup ratio} = S$$

The average time to access a data is given by:

$$= \text{Hit ratio} * \text{Cache access time} + ((1 - \text{Hit ratio}) * (\text{Memory access time}))$$

The access time depends on the program and how data reference pattern is made. Hence, we calculate the average of all access time and the mean access time is used in our calculation. A cache miss can happen for both data and instruction. A cache miss on a data read may be less serious than instructions, as in principle, it continues execution until the data to be fetched. On an instruction, fetch requires that the processor to 'install' and wait until the instruction is available from main memory. Hence, it is advised to use two different caches in the system, one for data and the other for instruction.

### 4.9.5 Interleaving

Interleaving is an advanced technique used by high end motherboards/chipsets to improve memory performance. Memory interleaving increases bandwidth by allowing simultaneous access to more than one stack of memory. This improves performance because the processor can transfer more information to and from memory in the same amount of time. It helps alleviate the processor memory bottleneck that is a major limiting factor in overall performance. Interleaving works by dividing the system memory into multiple blocks. The most common numbers are two or four called two-way or four-way interleaving respectively. Each block of memory is accessed using different sets of control lines, which are merged together on the memory bus. When a read or write is begun to one block, a read or write to other blocks can be overlapped with the first one. The more blocks, the more overlapping can be done. Interleaving is an advanced technique that is not generally supported by most PC motherboards because of cost. It is most helpful on high end systems, especially servers that have to process a great deal of information quickly. Interleaving infrastructure is arranged with memory. To speed

NOTES

up the memory operations (read and write), the main memory of  $2^n = N$  words can be organized as a set of  $2^m = M$  independent memory modules (where  $m < n$ ) each containing  $2^{n-m}$  words. If these  $M$  modules can work in parallel (or in a pipeline fashion), then ideally an  $M$  fold speed improvement can be expected. The  $n$ -bit address is divided into an  $m$ -bit field to specify the module and another  $(n-m)$ -bit field to specify the word in the addressed module. The field for specifying the modules can be either the most or least significant  $m$  bits of the address. For example, these are the two arrangements of  $M = 2^m = 2^2 = 4$  modules ( $m = 2$ ) of a memory of  $2^n = 2^4 = 16$  words ( $n = 4$ ). Before the data signal is modulated and spread, an interleaving process scatters the bit order of each frame so that if some data is lost during transmission due to a deep fade of the channel, for example the missing bits can possibly be recovered during decoding. This provides effective protection against rapidly changing channels, but is not effective in slow changing environments.

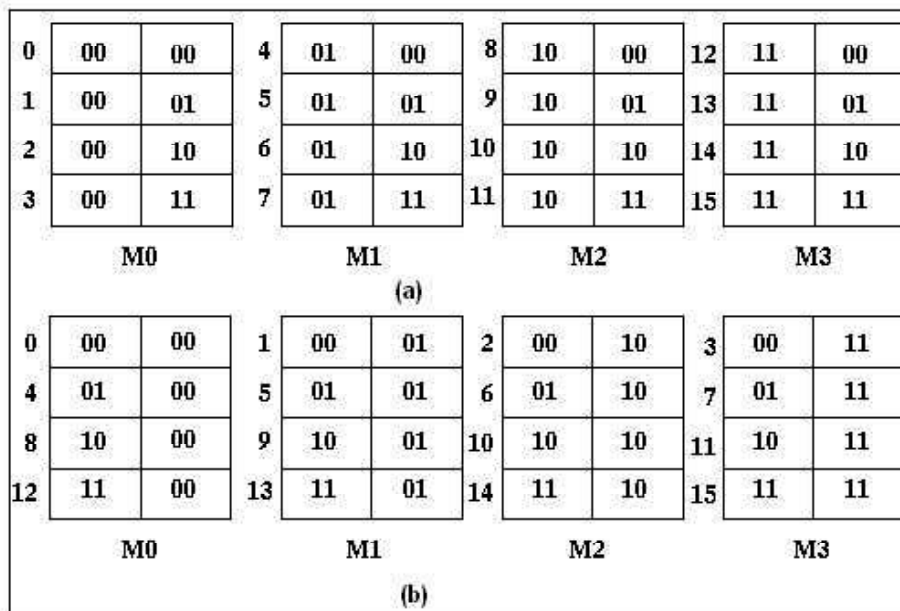


Fig. 4.65 (a) High Order Arrangement and (b) Low Order Arrangement of Interleaving

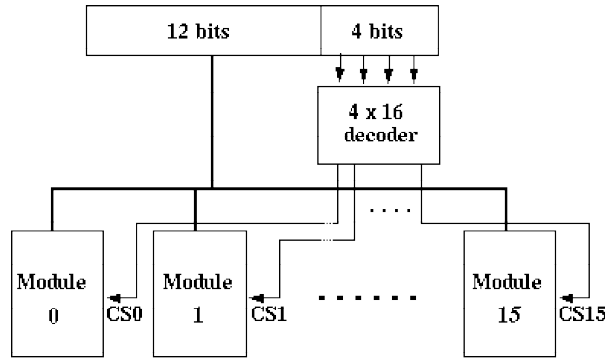
In general, the CPU is more likely to need to access the memory for a set of consecutive words either a segment of consecutive instructions in a program or the components of a data structure, such as an array, the interleaved (low order) arrangement is preferable as consecutive words are in different modules and can be fetched simultaneously (Refer Figure 4.65 (b)). In case of high order arrangement, the consecutive words are usually in one module, having multiple modules is not helpful if consecutive words are needed (Refer Figure 4.65 (a)). The following on example interleave infrastructure will make the concept clear.

**Example 4.3:** A memory of  $2^{16} = 64K$  words ( $n = 16$ ) with  $2^4 = 16$  modules ( $m = 4$ ) each containing  $2^{n-m} = 2^{12} = 4K$  words.

**Solution:**

In interleaving process you can find that a memory of  $2^{16} = 64K$  words are utilized if ( $n = 16$ ), i.e.,  $2^4 = 16$  modules ( $m = 4$ ) are given. Each contains  $2^{n-m} = 2^{12} = 4K$  words.

**NOTES**



**Memory Interleaving**  
( $2^{16}$ =64K words, 16 modules)

**4.9.6 Hit Rate and Miss Penalty**

Let us consider a system which has a cache ‘hit rate’ of 95 per cent, cache memory of 20 nanoseconds (ns) cycle time and main memory of 150 ns cycle time. The average cycle time of the system will be calculated as follows:

$$(0.95) * 20 \text{ ns} + 0.05 * 150 \text{ ns} = 26.5 \text{ ns}$$

The effective memory cycle time as a function of cache hit rate for the above system is summarized in Table 4.12.

*Table 4.12 Cache Hit and Effective Cycle Time*

Cache Hit Per Cent	Effective Cycle Time (ns)
80	46
85	39.5
90	33
95	26.5
100	20

From the above example, it can be easily seen that the effective access time is greatly reduced with the increase in cache hit.

The speed ratio may be defined as the ratio of the memory system’s access time without cache memory to its access time with cache memory. It helps us to understand how much acceleration is observed in access time by using cache memory along with main memory. The speedup ratio is defined as: ratio of time taken for N access without cache and with cache memory.

The speedup ratio is, therefore, given by:

$$S = \frac{Nt_m}{N(ht_c + (1-h)t_m)} = \frac{t_m}{ht_c + (1-h)t_m}$$

Where, N accesses to a system without cache memory require  $Nt_m$  seconds.

N accesses to a system with cache require  $N(ht_c + mt_m)$  seconds.

Usually memory designers consider the relative speed of the main memory and cache memories more than the absolute speed. Hence, the ratio of the access time of cache memory to main memory  $k = t_c/t_m$ , is used in the above formula than



taking typical values for  $t_m$  and  $t_c$ . In present system, this ratio is of the order 0.02. Therefore,

$$S = \frac{\frac{t_m}{t_m}}{\frac{ht_c}{t_m} + (1-h)\frac{t_m}{t_m}} = \frac{1}{hk + (1-h)}$$

## NOTES

If  $h = 0$ , all accesses are made to the main memory and the speedup ratio is 1, as there is no gain by using cache.

When  $h = 1$ , all accesses are made to the cache and a speedup ratio of  $1/k$  is achieved.

**Level 2 Caches:** The hierarchy cache memory, main memory, hard disk can be further expanded by dividing the cache into a level 1 and a level 2 cache. A level 1 cache normally lives on the same chip as the CPU itself, i.e., integrated with the processor. Level 1 caches grow in size as semiconductor technology advances and more memory devices can be integrated on a chip. A level 2 cache lives off the processor chip and is larger than a level 1 cache. Level 2 caches are typically of  $\frac{1}{2}$  MB. When the processor makes a memory access, the level 1 cache is first searched. If the data is not there, the level 2 cache is searched. If it is also not in the level 2 cache, the main store is accessed. The average access time is given by:

$$t_{\text{cache}} = hL_1 t_{c1} + tL_2 t_{c2} + (1 - hL_1 - hL_2) t_{\text{memory}}$$

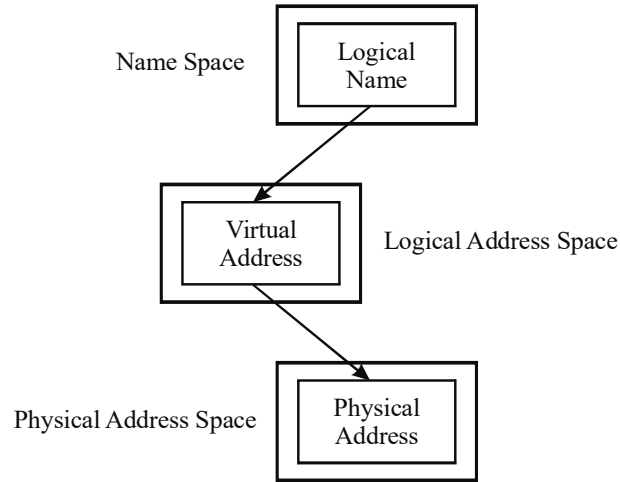
Where,  $hL_1$  and  $hL_2$  are the hit rates of the level 1 and level 2 caches and  $t_{c1}$  and  $t_{c2}$  are the access times of the L1 and L2 caches respectively.

### 4.9.7 Virtual Memory

As you know, all data is stored in the hard disk and the program that is under the execution resides in the main memory. The virtual memory is a concept that permits the user to construct a program with size more than the total memory space available to it. This technique allows user to use the hard disk as if it is a part of main memory. Hence with this technique, a program with size even larger than the actual physical memory available can execute. Here the only thing required is an address mapping from virtual address to physical address in main memory. An address generated by CPU during execution of program is called virtual address and the set of such addresses is address space. An address in the main memory is called physical address and the set of these addresses is called memory space. A virtual memory system provides a mechanism for translating a program generated address by the processor into main memory location. A program uses the virtual memory addresses space which stores data and instruction. In usual case, the address space is more than memory space, where actually manipulation has to be done. If there is a main memory of capacity of 32K words, 15 bits are required to specify the physical address of memory. Let the system have auxiliary memory of 1MB size and it will require 20 bits, i.e., address bit to access the data. As said earlier, in the virtual memory system, a mapping from a virtual address space to a physical address space is required. System uses a table that maps a virtual address of 20

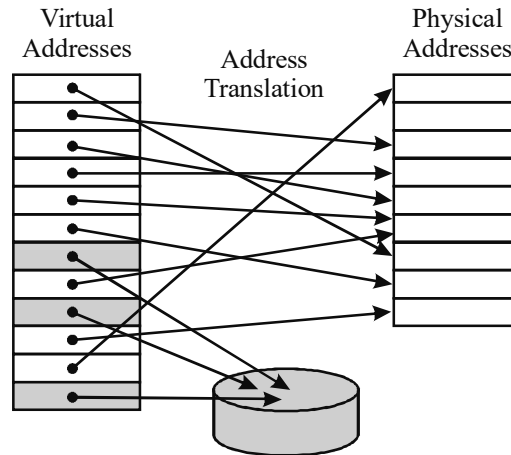
bit to physical address of 15 bits. This process is required for translation of every word (Refer Figure 4.66).

**NOTES**



**Fig. 4.66** Translation of Logical Address to Physical Address

In Figure 4.67, a relationship between virtual address and physical address is shown. By using address translation, we calculate the physical location of data in main memory. It can be seen from the figure that the virtual address space is more than the physical address space.



**Fig. 4.67** Mapping from Virtual Address to Physical Address

This technique is especially useful for a multiprogramming system where more than one program resides in main memory. Such a system is managed efficiently with the help of operating system. The objective of virtual memory is to have maximum possible portion of program in the main memory and the remaining portion of program to reside on the hard disk. The operating system, with some hardware support, swaps data between memory and disk such that it interferes minimum with the running of the program. The operating system manages the whole memory. If it is required to refer to hard disk very frequently such that swapping in and out of data between the main memory and hard disk becomes the dominant

activity. Such condition is referred to as *thrashing* and it reduces the efficiencies of the system greatly. Virtual memory can be thought as a way to provide an illusion to the user that disk is an extension of main memory.

Any program under execution should reside in the main memory as CPU cannot directly access hard disk. The main memory usually starts at physical address 0. Certain memory locations are reserved for the special purposes program, such as operating system. It can be either at the low addresses or other high end. However, it is usually at the low end. The rest of main memory is divided into pieces where different programs reside. Now a days, most operating systems have a multiprogramming environment, i.e, there are more than one programs that reside in main memory. Different processes are mapped to different physical locations in the main memory.

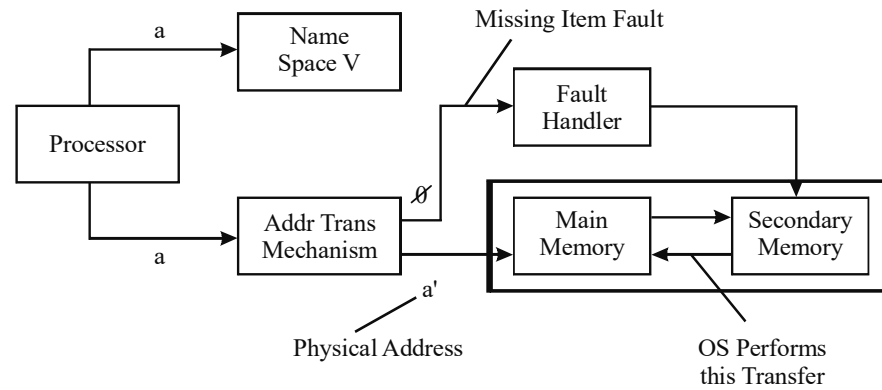
## NOTES

### Principles of Implementation of Virtual Memory Concept

Let us study how the virtual memory concept is actually implemented by an operating system.

- The maximum portion of program should be in the main memory. Also, maximum number of possible processes should be loaded into the main memory.
- A copy of the complete process, which is also called its image, should be maintained in a hard disk, i.e., the inclusion principle of memory hierarchy. This file is called the *swap* file.
- The main memory can be divided into number of pages of equal sizes, which are nothing but chunks of memory. Pages are commonly of sizes 512, 1024, 2048 or 4096 bytes or it can be of unequal size called segments. The virtual memory manager organizes these pages in the memory and also takes care of their protection.
- Similar to memory, programs are also divided into pages of the same size in which the memory is partitioned. Only some pages of a complete program reside in the main memory others do not. But a complete program is always there in the swap file.
- If the portion of the process which is loaded into main memory is sufficient to complete the process, then the only job of the memory manager is to perform address translation with support from the hardware
- However, if the process needs the data that is not present in the main memory, a page fault is reported. Now it is the job of the memory manager to handle the fault by bringing the page that contains the required data from the swap file. As the swap file resides in secondary memory, the overall speed of system reduces as speed of reading a disk file takes a minimum of 10 milliseconds as compared to reading the main memory may take only 10 nanoseconds. To create a space for storing this page, we require removing some pages from the memory. The page which is to be replaced is decided by the *page replacement policy* adopted by the operating system.

**NOTES**



**Fig. 4.68** Paged Memory Management Address Translation

Address generated by CPU refer to the process of address translation in which fault header retains missing item fault which can be fetched either through main memory or through secondary memory (Refer Figure 4.68). The use of the mapping function between the processor and the memory provides flexibility by allowing data to be stored at different locations in the memory. The programmers perspective of logical arrangement is different from the actual data arrangement. The advantages of using the virtual memory technique are as follows:

- With this scheme, programs are compiled for a standard address space and then loaded into the available memory and executed without any modification. This provides a big flexibility.
- More than one program can fit into memory. Hence, it is very adjustable to an operating system that supports multitasking in which an arbitrary number of programs reside in the main memory.
- It is very useful for running large program on machine having comparatively smaller memory. The operating system which works on the concept of virtual memory make it possible to run a program having a size larger than the memory size by explicitly moving the portions of the program or data in and out from disk.

**Techniques of Implementing of Virtual Memory**

The concept of virtual memory is implemented by either of these three techniques: paging, segmentation and a technique which is combination of the two is known as paged segmentation.

**Paging Memory Management**

In paging technique, the memory is divided into fixed size blocks called pages. As main memory is smaller than virtual memory, the total number of pages in main memory is lesser than the virtual memory. For example, let us consider a physical memory of 1GB size. Suppose the virtual memory is of size of 4GB. Thus, there is 1 to 32 mapping. A page map table is used for implementing a mapping. There is one entry per virtual page, and the address of corresponding physical address is stored.

The presence bit is used for letting the system know whether the page has been transferred form the auxiliary memory into main memory. A 0 bit represents

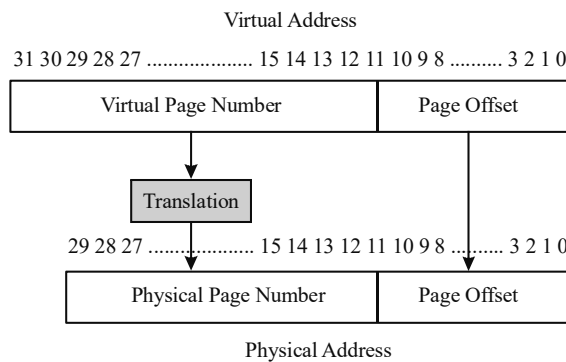
that this page is not available in the main memory. When the presence bit indicates that the page is not available in main memory, a page fault exception is triggered and the operating system brings the page into memory. However, once the transfer is complete, the operating system stores the new physical page number into the page table and continues with the instruction that has caused the page fault to complete that program.

The main memory and secondary memory are divided into pages. Let the page be of size 4KB. The system will have 1MB pages in virtual memory. In case of main memory for the pages size of 4KB, there will be 256KB pages. Thus, there is 1 to 4 mapping. The secondary storage address is the address of the disk where data is stored. This address is split into the offset and page number. Here, 12 bits are used for offset and 20 bits for virtual page number.

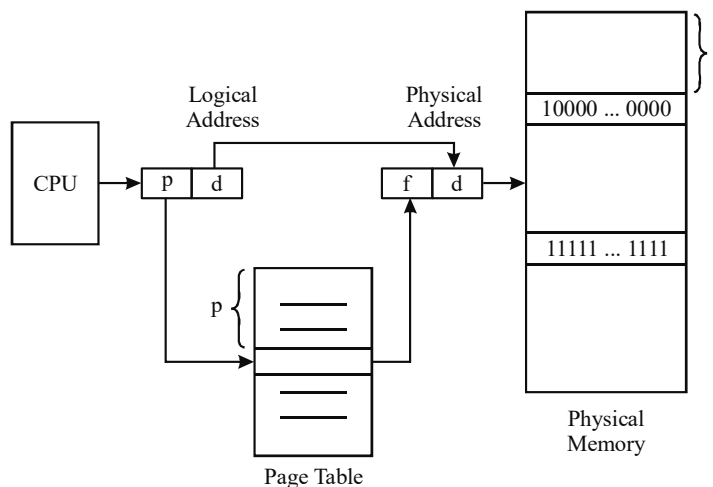
Physical page address is also divided into page number and offset. Here 18 bits are used for page number and 12 bits are used as offset.

The translation from virtual page number to physical page number is done through a page map table as shown in Figure 4.69 (a). The page number is the index of this page map table. The mapping is implemented by looking up the virtual page address as index of this page map table to find the location from where data is to be fetched.

**NOTES**



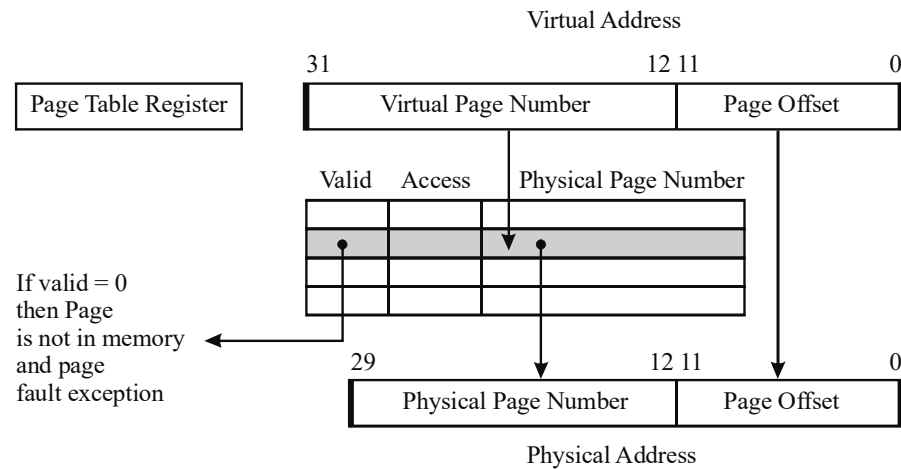
**Fig. 4.69 (a)** Address Translation from Virtual Address to Physical Address



**Fig. 4.69(b)** Hardware Implementation of Address Translation

## NOTES

There is one entry per virtual page in the page map table and the data stored is the address of corresponding physical address (Refer Figure 4.69 (b)). The entry of a page table is shown in Figure 4.70. Usually this 'page translation table' is maintained using associative memory that allows a fast determination of the physical address in main memory corresponding to a particular virtual address. A total of 20 entries will be there in the page map table. The page offset (low order 12 bits) is appended in physical page number to get desired location where that word is found in the particular page. Thus, the virtual address translation hardware appears as follows:



**Fig. 4.70** Virtual Mapping Technique

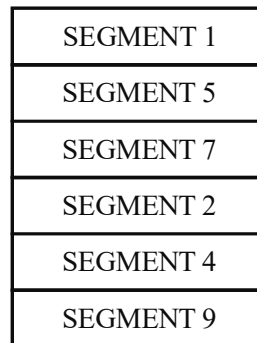
Along with page number and offset few other attributes are also stored in a page translation table, such as the one shown in Figure 4.70 is the access right. The access right determines what type of rights are provided, such as read only, read write, etc. These are stored as extra fields to the table. The common information that are stored in page map table are as follows:

- The access privileges, so that no program can corrupt data of another program
- The 'Dirty' bit a bit which indicates whether or not a page has been modified. If page content are changed it must be written back onto the disk while page replacement.
- How recently the page is used.

The page size is an important issue while designing the virtual page memory technique. If we reduce the page size, the Page map Translation Table (PTT) will increase and vice versa. For example, if a processor has a virtual memory address of 32 bits and it is divided into pages of size 4096 bytes, i.e., the offset bit are 12 and there will be a 1024 K pages thus total possible entry in page map table will be  $2^{20}$ . However, if page size is 10 K ( $2^{20}$  bytes), then there will  $2^{12}$  pages and 4 K page table entries. These large page tables will normally not be full, since the number of entries is limited to the amount of physical memories available. These bigger pages may result in the problem of internal fragmentation.

## Segmented Memory Management

In a segmented memory management, the main memory is divided into blocks of unequal size. Here, the division is based on logical concept and each segment corresponds to a logical block of code or data, for example, a subroutine or procedure. Thus, we can divide the user program into segments (logical division) and these segments may be placed anywhere in main memory but the instructions or data in one segment should be contiguous, as shown in Figure 4.71.



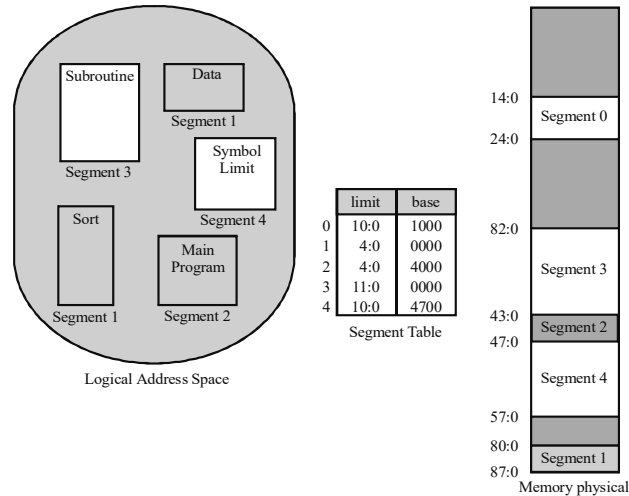
*Fig. 4.71 A Segmented Memory Organization*

Segmentation is implemented in a similar manner as paging using a lookup table. There is one entry corresponding to each segment called segment descriptor. As the segments are of unequal sizes, each segment descriptor in the table should contain the base address (start address) of the segment and its length. Each process can be assigned a different segment descriptor in the lookup table, completely unaware of how many processes are sharing that segment. The address translation from virtual memory to physical memory is done effectively at run time by the virtual memory mapping mechanism. In order to keep the size of the segment tables small, the maximum number of segments in which memory is divided is typically small compared to the virtual address range, for example, 254. Similar to paging technique, in this case also the virtual address is divided into segment number and a segment offset. Some other information like protection information, such as the read/write permission of the segment and the process ID, are stored in the segment descriptor. It also stores other housekeeping information, such as a presence bit and dirty bit. In this scheme, unlike paging scheme where offset is of fixed size, the segment offset field grows and shrinks depending on the logical length of the segment. Segmentation is able to remove the internal fragmentation problem which exists in paging technique.

Segment is very useful in the multiprogramming environment, especially if two processes share some procedures, such as library functions. In such circumstances, only one copy of the code of that procedure resides in memory and both processes use it, i.e., reference to the same code is made in the segment descriptor table of both programs. Figure 4.72 shows the memory allocation process using segmentation technique. The logical address space keeps Segment 1 for data, Segment 2 for main program and Segment 3 for subroutine.

## NOTES

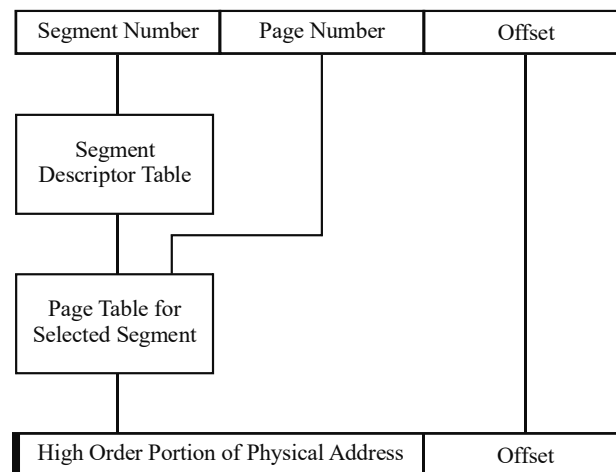
**NOTES**



**Fig. 4.72** Memory Allocation using Segmentation Technique

**Paged Segmentation**

The main advantage of the paged memory management system over the segmented memory management system is that it is simpler to implement and it does not suffer from external fragmentation which is observed in the segmented memory management technique. However, the major disadvantage of paged memory management is that it suffers from internal fragmentation and also the division is done physically instead of logical division as done in the segmentation technique. Hence, the pages do not necessarily correspond to complete functional blocks or data elements, as in the case of the segmented memory management. So, swapping is more frequent in a paging scheme as compared to a segmentation scheme, i.e., the operating system has to swapped in or out page of memory, if full of data or instructions that is required for execution is not present in a page. The paged memory management can be considered as a special case of segmented memory management such that all segments are of equal size. And in advance system, we combine the two techniques and the memory is divided into segments, with each segment further divided into pages. The address generated by the system consists of segment number followed by page number and then fixed size offset, as shown in Figure 4.73.



**Fig. 4.73** Page Segmented Translation



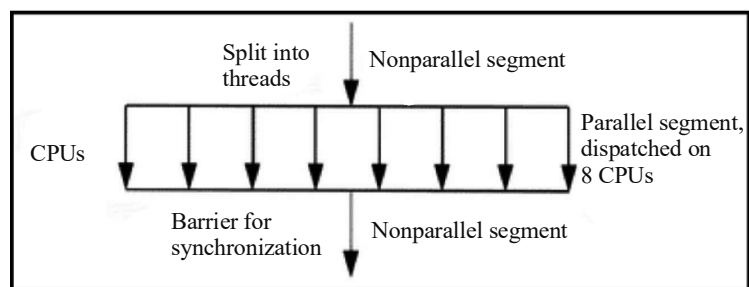
## 4.10 PARALLEL PROCESSING

Parallel processing is the architecture of working in a single computer that uses multiple processors to execute the various parts of the same programs simultaneously. The main objective of parallel processing is to reduce the processing time. The computer resources can include a single computer with multiple processors or networks or a combination of both. Parallel processing is arranged in a uniprocessor system using the pipelining technique.

The area of parallel processing is the approach of creating effective parallel computers that also includes parallel Linux clusters. In parallel processing technique, many applications use parallelism maintained by various types of compilers, such as Silicon Graphics' MIPSpro Power C and MIPSpro Power FORTRAN 77, etc. In essence, a normal machine is run as a parallel machine after changing an alternating sequence of serial and parallel sections.

The parallel processing takes place if the applications take inherent serial actions, such as opening files and initializing data areas. For example, loops, subroutine calls, etc. Parallel processing machines are much faster than single processing machines because more processors are installed in them. The applications of parallel processing computers use parallelism concept to simulate parallelism. They use it in such well-known applications as neural networks, cellular automata, simulations, video games and Very High Speed Integrated Circuits Hardware Description Language (VHSIC VHDL).

If a machine runs under parallel segment, the applications are distributed into a number of threads. The logic behind this is to execute various portions of parallel segment at the same time. At the end of parallel segment, a barrier exists where all the threads are collected together until and unless they are finished executing and the next serial segment starts. This type of parallel processing is referred to as 1 to N that means one application runs on N CPUs. This is the basic fundamental concept behind the parallel processing machines. To see the architecture under Parallel Processing see Figure 4.74.



**Fig. 4.74** Architecture under Parallel Processing

There are some limitations in parallelism that includes the communication overhead between the threads of application. The parallelism supports underlying hardware. Most of the parallel processing techniques speed up parallel processors, such as four processors or eight processors. The number of processors varies

### NOTES

because parallelism depends on the application and the coding in which it is installed. To see parallel applications speedup see Figure 4.75.

## NOTES

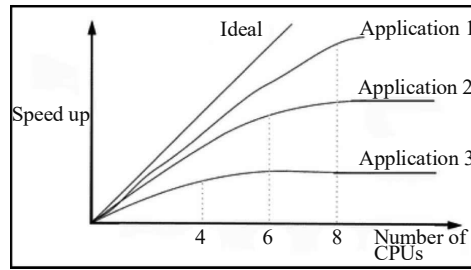


Fig 4.75 Parallel Applications Speed Up

For example, the IRIX/IRIS (Integrated Raster Imaging System) operating system has many features to manage these types of threaded applications that include shared memory, thread manipulations, high performance user-level locks and semaphores. It is assembled by a shared memory programming model that provides the high performance and fast speed. Multiple parallel applications on parallel processing are characterized by one or more of the following workloads:

- In parallel processing, more than one user could be on the same parallel application or different parallel applications.
- More parallel applications achieve a small number of parallel threads like four threads to eight threads. These threads can be increased in power challenge multiprocessors to enhance the performance.
- In some loosely-coupled message passing applications, a typical multiprocessor workload is resembled to speed up the multiprocessing area.
- In parallel execution, workload associated with high multiprocessor workload satisfy the general purpose and non-parallel applications needs power challenge servers with up to 18 R8000 CPUs in combination with SGI parallelizing tools and IRIX operating system. Parallel processing includes parallel throughput in Figure 4.76 that refers to the performance for multiple parallel machines at the same time on a multiprocessor.

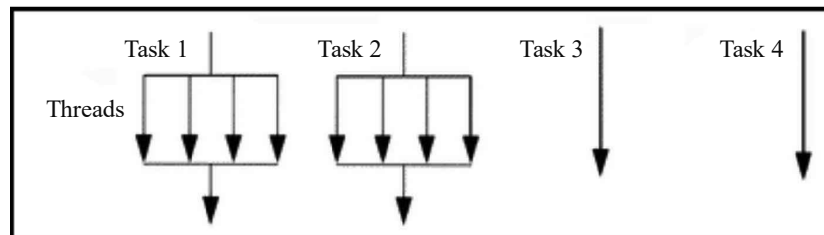


Fig 4.76 Parallel Processing Includes Parallel Throughput

These parallel processing underlies the operating system in a parallel computer. The suitable operating system is able to start up a number of threads quickly. The highest performance for parallel computers is achieved by greatest possible system throughput. The efficiency of shared memory depends on the amount of memory sharing in parallel processing. The workstation clusters is an extreme example of a multiple processing model but not suitable for codes in computation to communication

ratio processing. In such clusters, multiple threads work with network links. They both collectively provide high and efficient bandwidth network connections.

Parallelism has no advantages but this concept has different levels of effectiveness by using some methods. These methods are as follows:

- Symmetric Multiprocessor (SMP)
- Non-Uniform Memory Access (NUMA)
- Uniform Memory Access (UMA)
- Single Instruction Multiple Data (SIMD)
- Multiple Instruction Multiple Data (MIMD)

## NOTES

### Symmetric Multiprocessor (SMP)

The SMP is easy to program but does not scale beyond 16 or 32 processors. The SMP systems range from two to as many as 32 or more processors. However, if one CPU fails, the entire SMP system is down. Clusters of two or more SMP systems can be used to provide high availability fault tolerance. If one SMP system fails, the others continue to operate. See SMP in Figure 4.77.

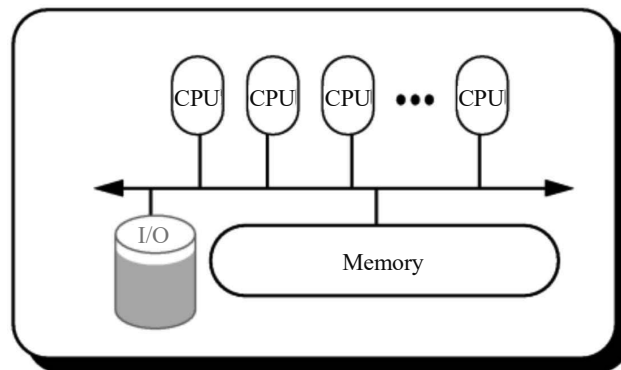


Fig 4.77 Symmetric Multiprocessor

### Non-Uniform Memory Access (NUMA)

In NUMA configurations, accessing specific locations in main memory is different for some of the CPUs relative to the others. It typically consists of several smaller SMPs wired together. It is easier to program than an MPP.

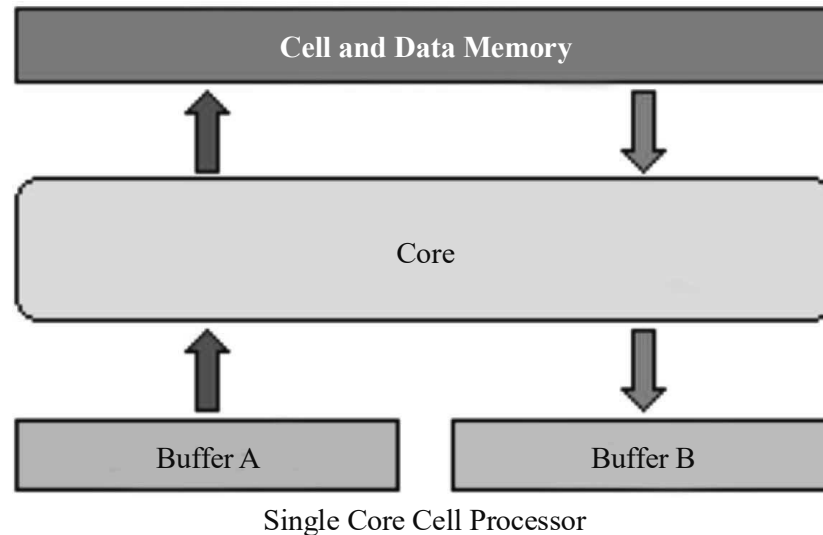
### Uniform Memory Access (UMA)

This architecture in UMA is referred to as UMA architecture in the sense that each socket can get to any memory location in a uniform way, primarily in terms of latency. The memory access is uniform in this processor.

### Single Instruction Multiple Data (SIMD)

The processor core is designed as a pure vector core, which is not to be confused with a GPU core, which uses an SIMD configuration (Refer Figure 4.78). In SMP, CPUs are assigned to the next available task or thread that can run concurrently. In parallel processing operation, the problem is broken up into separate pieces, which are processed simultaneously. The processor is either single core or multicore.

NOTES



*Fig. 4.78 SIMD Configuration*

**Multiple Instruction Multiple Data (MIMD)**

In a multicore processor, the cores would divide the instruction load in the buffers among themselves in a way that is completely transparent to the programmer. Adding more cores would simply increase the processing power without having to modify the programs. In an MIMD configuration, the processor is universal, meaning that it can handle all types of applications. There is no need to have a separate processor for graphics and another for general purpose computing. A single homogeneous processor can do it all.

**Parallel Computing**

Parallel computing means doing several tasks simultaneously, thus improving the performance of the computer system. Parallel processing is a type of computing where numerous instructions are implemented at the same time on the principle that larger problems may be divided into smaller ones that are solved in parallel. The different types of parallel computing are as follows:

- Bit-level parallelism
- Instruction-level parallelism
- Data parallelism
- Task parallelism

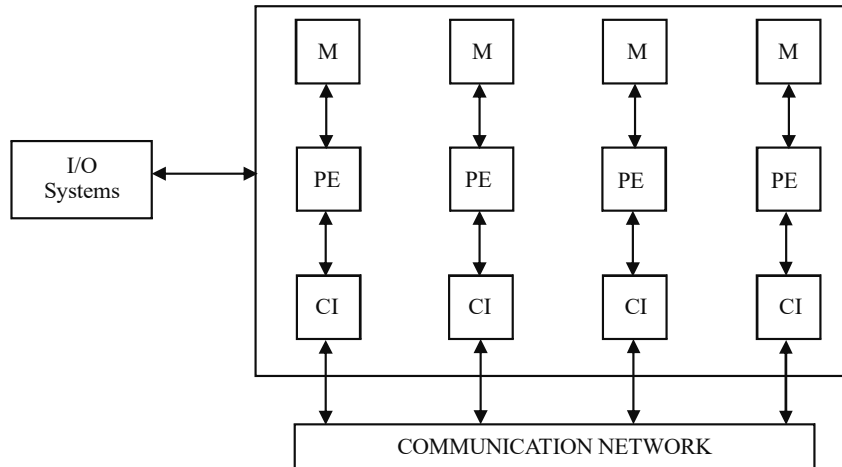
If a computer system provides the technique for simultaneous processing of different sets of data, then the computer system is said to be using a parallel processing technique. The parallel processing technique increases the computational speed of the processing system. Thus, it enhances the performance and throughput of a computer. The various techniques by which parallel processing can be achieved are as follows:

- Pipelining
- Vector processing
- Parallel processors

A parallel computer is defined as an interconnected set of Processing Elements (PEs), which cooperate by communicating with one another to solve large problems fast.

Thus, from this definition, the keywords which define the structure of a parallel computer are processing elements, communication and cooperation. The generalized structure of a parallel computer is shown in Figure 4.79.

## NOTES



*Fig. 4.79 Parallel Computer*

PE: Processing elements

CI: Communication interface

I/O: Input–Output

M: PE's local memory

A PE may have a private memory in which case it is called a Computing Element (CE). The heart of the parallel computer is a set of PEs interconnected by a communication network. This general structure can have many variations based on the following:

- Types of PEs used
- Memory available to PEs to store programs and data
- PEs-memory connection
- Type of communication network used
- Technique of allocating tasks to PEs
- Their mutual communication and cooperation

The variations in each of these lead to a rich variety of parallel computers.

### Types of PEs

- PEs may be arithmetic and logic units only. The ALU may use 64-bit operands or may be quite tiny using 4-bit operands.
- A PE may be a microprocessor with only a private cache memory or a full-fledged microprocessor with its own cache and main memory. A PE with its own private memory is called a computing element.
- A large powerful computer such as a mainframe or a vector processor.

## NOTES

### Number of PEs

- A small number of powerful PEs (10 to 100)
- A medium number of microprocessors (100 to 1000)
- A large number of tiny PEs (e.g., ALU > 1000)

### Memory system

- Each PE has its own private cache and main memory.
- Each PE has its own private cache but all PEs share one main memory, which is uniformly addressed and accessed by all PEs.
- Each PE has its own private cache and main memory and all of them also share one large memory.

### Mode of Cooperation

- Each CE has a set of processes assigned to it. Each CE works independently and CEs cooperate by exchanging intermediate results.
- A host CE stores a pool of tasks to be executed and schedules tasks to free CEs dynamically.
- All processes and data to be processed are stored in the memory shared by all PEs. A free PE selects a process to execute and deposits the result in the memory for use by other PEs.

### Communication Networks

- A fixed interconnection network.
- A programmable interconnection network where switches are used for interconnection, which can be programmed using bit patterns to change the connections.
- A single bus or a set of buses as the communication network.
- A memory shared by all the PEs, which is used to communicate among the PEs.
- A combination of two or more of the above.

#### 4.10.1 Overcoming Pipelining Conflicts

There are several problems associated with controlling smooth, efficient execution of instructions on the pipeline. These problems are generally called hazards. There are four possible techniques for resolving a hazard.

- Forward:** If the data is available somewhere, create extra data paths to 'forward' the data to where it is needed. This is the best solution, as it neither slows down the machine nor does it change the semantics of the instruction set.
- Add Hardware:** This is the most appropriate structural hazard where if a piece of hardware has to be used twice in an instruction, then if possible you should duplicate the hardware.
- Stall:** Let the instruction wait until the hazard resolves itself. It is not a good solution as it slows down the machine, but may be used if no other alternative is possible and it is necessary.

- (iv) **Document (AKA punt):** Define instruction sequences that are forbidden, or change the semantics of instructions, to account for the hazard. Some such examples are delayed loads and delayed branches.

These hazards can be broadly classified in many types. These types are as follows:

1. **Structural hazards:** Structural hazards occur when different instructions try to access the same piece of hardware at the same time, especially resource conflicts, which arise when memory is accessed by two segments at the same time. For proper functioning of pipelining the resources required by operands for execution must be independent. This type of hazard can be removed by having more than one piece of that hardware for the segments wherein the collision can occur, i.e., you should duplicate the resources such that all possible combinations of instructions in the pipeline can execute simultaneously. You can use separate data and instruction memory if there is some conflict in memory access. These hazards can overcome if required by inserting stalls or reordering the instructions. Let us study a few examples where structural hazards have been observed and a solution has been found for them.

Lets see how the structural hazard can be overcame by using two separate memory units, one for data and other for instruction, instead of using a single memory unit. In case of pure sequential program, i.e., where there is a simple non-pipelined implementation of any program it would work equally well with either approach.

In effect, the pipeline design you are starting from has anticipated and resolved this hazard by adding extra hardware.

Also, the first Sparc implementations (remember, Sparc is almost exactly the RISC machine defined by one of the authors) did have exactly this hazard, with the result that load instructions took an extra cycle and store instructions took two extra cycles.

**Example 4.4:** Let us take a floating number addition problem pipeline. Here is the program which contains following two instructions

$$R3 = R0 + R2$$

$$R4 = R1 + R2$$

**Solution:** Here, both the instructions have R2 as one of their inputs. This can result a potential conflict in stages two and three of the adder pipeline. Let us take an example that R2 has to shift its mantissa in order to equalize the exponent with R1. Now, this changed value of R2 will be added to R0 in order to obtain value of R3 in third stage which will generate a wrong result. A possible solution to the above problem is to make multiple copies of the operands, and pass the individual copies through the pipeline. In the addition of first pair, the CPU provides the adder circuit a copy of R0 and R2 when it starts the pipeline for the first pair, and the second stage will obtain the result from these copies only similarly the second pair will get separate copies.

Let us take an example, where the first instruction involves a register file write operation and fourth instruction involves register file read operation. Instr i will be completed at 5 clock pulse when write operation is done. Instr i+3 starting at 4th

## NOTES

clock pulse will require this register. As required data is not present, hence the normal flow of data in pipeline does not occur. To overcome this hazard, the pipeline stalls for one clock cycle.

**NOTES**

Table implementation of stalls during clock cycle

Instr	1	2	3	4	5	6	7	8	9	10
Instr i	IF	ID	OD	EX	WB					
Instr i+1		IF	ID	OD	EX	WB				
Instr i+2			IF	ID	OD	EX	WB			
Stall				bubble	bubble	bubble	bubble	bubble		
Instr i+3					IF	ID	OD	EX	WB	
Instr i+4						IF	ID	OD	EX	WB

(a)

For simplification, refer to the following table:

Instr	1	2	3	4	5	6	7	8	9	10
Instr i	IF	ID	OD	EX	WB					
Instr i+1		IF	ID	OD	EX	WB				
Instr i+2			IF	ID	OD	EX	WB			
Instr i+3				stall	IF	ID	OD	EX	WB	
Instr i+4						IF	ID	OD	EX	WB

(b)

The above problem can be resolved either by (a) duplicating hardware or (b) modifying the existing hardware to support concurrent operations. If you duplicate the hardware as in this case if you use two register files instead of one, then you could perform concurrent read and write operations. However, this could lead to the consistency problem because if you are duplicating the instruction it is always possible that at a given clock cycle, registered in one register file could have different values than the corresponding registers in the other register file. This inconsistency is clearly unacceptable if accurate computation is to be maintained.

Another way of handling it could be if you modify the register file architecture such that it performs register write on the first half of the clock cycle and register read operation on the second half of the cycle. In earlier hardware, designers sometimes inserted a delay between write and read that was very small in relation to the clock cycle time, in order to ensure convergence of the register file write.

Some other conditions when structural hazards are observed are as follows:

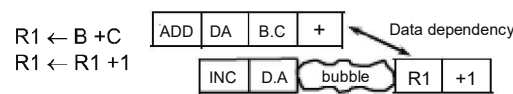
- Structural hazards could also be observed during the branch instruction, if there were not two ALUs. Thus, with only one ALU, you cannot perform the simultaneous computation of any variable (arithmetic operation) and at the same time determine whether or not the branch condition was fulfilled (logical operation).
- A further structural hazard is also observed if you are using only single memory for storing both instructions as well as data. This problem is very similar to using single file for the concurrent reads and writes on the register file. The possible solution could be:
  - (i) Using two different caches, one for instructions, and other for data. It is better than using two types of main memory, one for data and other for instruction as cache is more frequently used than the main memory.



- (ii) Another possible solution could be design a special-purpose memory module that permits writing (reading) on the first half of the clock cycle. In order to implement it however you would require special hardware which is quite expensive.

**2. Data Hazards:** Data hazards occur when the input required for execution of an instruction depends on the output of the previous instruction still in the pipeline, i.e., not completed yet as a result output will be not available in the beginning of a new segment. This hazard causes a degradation of performance in instruction pipeline as there will be collision of address or data that are needed for the execution of instruction. All the data hazards discussed here involve registers within the CPU. By convention, the hazards are named by the ordering in the program that must be preserved by the pipeline (Refer Figure 4.80).

**NOTES**



*Fig. 4.80 Data Hazards*

Consider two instructions *i* and *j*, such that the instruction *i* occurs before *j*. There are four possible data hazards:

- (i) **RAW (Read After Write)** – It happens when *j* tries to read a source before *i* has modified (written) the value, so *j* gets the old value which is not correct. Hence, here an instruction tries to read an operand before a previous instruction is changed (modified contents of operand). This is caused by a true dependence. For example:

```
Sub R1, R2, R3
Add R4, R5, R1
```

Here the 2nd operation involves the addition operations. It reads the output of 1st instruction, i.e., subtract operation. Thus, for the correct execution add operation should not be executed till the requisite modified data is not available. This is the most common type of hazard and can be overcome by using forwarding.

- (ii) **WAW (Write After Write)** – It happens when *j* tries to write an operand before it is written by the first instruction, i.e., *i* instruction. This happens when the writing process is performed in the wrong order, leaving the value written by *i* rather than the value written by *j* in the destination. This occurs when there are two instructions and both of them have to write on the same register.

This hazard is observed in the case when in a pipeline there are two segments that have write operations or allow an instruction to proceed even when a previous instruction is stalled.

- (iii) **WAR (Write After Read)** – *j* tries to write a destination before it is read by *i*, so *i* incorrectly gets the new value. This is caused by an anti-dependence that occurs when one instruction writes a register that will be read by another instruction. For example:

**NOTES**

```
add  r1, r2, r3    // r1 := r2 + r3
add  r3, r4, r5    // r3 := r1 + r4
```

There is an anti-dependence between the two instructions because the second instruction writes a register r3 that is used by the first instruction. This occurs as processor have finite number of registers and registers are reused. If there are infinite number of registers available, i.e., for each output there is a separate register then WAR and WAW dependencies will never happen.

- (iv) **RAR (Read After Read)** – This occurs when two instructions read from same registers. As reading does not change the value of registers RAR will not create a problem hence it is not considered a hazard.

The possible solutions for this problem are following:

- (a) **Stall Insertion:** The simplest remedy to all data hazards is to insert stalls in the execution sequence. These stalls are no operation instructions that are added (one or two) into the pipeline so that the requisite delays of the current instruction until the required operand is written on the register file. Thus, in order to ensure the correct the use of stall. Decrease the efficiency and throughput of the pipeline, which is against the goals of pipeline processor design. Stalls are thus an expedient method and should be used as the last resort under the conditions when other techniques like compiler action or forwarding fails or the hardware support required to execute these instructions or the required software design to implement them are not available.
- (b) **Forwarding:** Another technique to resolve this dependency is to add a special circuitry in the pipeline. This circuitry comprises a set of wires and switches with the help of which one can forward or transmit the desired value to the pipeline segment that is needed for the computation. Although this increases the cost and complexity of the system as you need additional hardware and control circuitry but this technique is very efficient and has improved throughput because it takes far less time for the required data to travel through this circuit than it does for a pipeline segment to compute its result (Refer Table 4.13).

*Table 4.13 Forwarding*

DD	R1, R2, R3	IF	ID	OD	EX	WB		
SUB	R4, R5, R1		IF	ID <sub>sub</sub>	OD	EX	WB	
AND	R6, R1, R7			IF	ID <sub>and</sub>	OD	EX	WB

In Table 4.13 result ADD operation is not needed by SUB operation till the time it produced it, i.e., the output of EX ADD operation is needed by OF Segment of SUB operation. If you have a circuitry through which one can move the result from where the ADD produces it (EX register), to where the SUB needs it (ALU input latch), then the stall which is introduced can be avoided. In above case forwarding

is implemented by forwarding the ALU result from the EX register to the ALU input latches (through feedback). If the forwarding hardware detects that the previous ALU operation has to write on the register, then the control logic selects the forwarded result as the ALU input rather than the value read from the register file.

**NOTES**

- (c) **Code Re-Ordering:** Code re-ordering is another technique where, the compiler reorders the source code or one can do the assembler by reordering the object code, thus by placing one or more statements between the current instruction and the instruction in which the required operand was computed as a result. In order to perform this operation one needs an ‘intelligent’ compiler or assembler, which must have in advance the detailed information about the structure and timing of the pipeline, that is when and where any possible data hazard could occur in the pipeline. This type of arrangement is called a hardware-dependent compiler. These compilers not only detect a data conflict but also reorder the instructions by inserting no-operation instructions or inserting any step which does not involve conflicting data so that the required delay in the loading of conflicting data happen by this technique called delayed load. For example let us consider the following case:

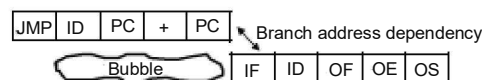
```
a = b + c ;
d = a - f ;
e = g - h ;
```

The stall can be easily avoided if step 2 is performed after step 3, i.e., if following sequence is adopted there will be no stall required

```
a = b + c ;
e = g - h ;
d = a - f
```

**3. Control Hazards:** They are the most difficult as well as most common types of hazards that arise with normal operation of a program. The most common among the control hazard is the branching instruction. Branching can be conditional or unconditional.

In case of the unconditional branching the PC is always loaded by new PC (Refer Figure 4.81).



**Fig. 4.81** Control Hazard

However, in case of conditional branching, there are always two possible alternative results:

- If condition meets, then jump to the branch target address.
- If condition does not meet, execute the instruction after the branch.

As in conditional branching the value of PC is dependent on the condition

## NOTES

and in case the condition is tested true, the contents of pervious pipeline are flushed out and the address of the instruction where one has to jump is loaded in PC so that the pipeline will continue with new address. But, in case when the condition tested is not satisfied there will be sequential execution, hence pipeline contents are not affected. The simplest method of dealing with branches is *to stall* /to insert bubbles in the pipeline as soon as the branch is detected until you reach the EX stage, which determines there is branching or interrupt to be handled and new PC is to be loaded and you should start with first IF instruction. The problem arise when the branch target address is not ready in time for the branch to be taken, which results in stalls (dead segments) in the pipeline that have to be inserted as local wait events, until processing can resume after the branch target is executed. The pipeline behaviour looks like as shown in Table 4.14.

**Table 4.14** Pipeline Behaviour

Branch	IF	ID	OF	EX	WB					
Branch successor		IF(stall)	<i>stall</i>	<i>stall</i>	IF	ID	OF	EX	WB	
Branch successor+1						IF	ID	OF	EX	WB

The stall does not occur until after ID stage.

This control hazards stall must be implemented differently from a data hazard, since the IF cycle of the instruction following the branch must be repeated as soon as you know the branch outcome. Thus, the first IF cycle is essentially a stall (because it never performs useful work), which comes to total three stalls. Three clock cycles wasted for every branch is a significant loss. With a 30 per cent branch frequency and an ideal CPI of 1, the machine with branch stalls achieves only half the ideal speedup from pipelining.

Control hazards can be mitigated through accurate branch prediction (which is difficult), and by delayed branch strategies. The problem with the branch instruction is that you usually do not know which result will occur (i.e., whether or not the branch will be taken) until the branch condition is computed. Often, the branch condition depends on the result of the preceding instruction, so you cannot precompute the branch condition to find out whether or not the branch will be taken. The following strategies are employed in resolving control dependencies due to branch instructions.

- **Assume Branch Not Taken:** As you saw, you can insert stalls until you find out whether or not the branch is taken. However, this slows pipeline execution unacceptably. A common alternative to stalling is to continue execution of the instruction stream as though there is no conditional branch. The intervening instructions between the branch and its target are then executed. If the branch is not taken, this is not a harmful or disruptive technique. However, if the condition satisfies and branching has taken place, then you must discard the results of the instructions executed after the branch statement. This is done by flushing the IF, ID and EX stages of the pipeline for the discarded instructions. Execution continues uninterrupted after the branch target. The cost of this technique is approximately equal to the cost

of discarding instructions. For example, if branches are not taken 50 per cent of the time, and the cost of discarding results is negligible, then this technique reduces the cost of control hazards by 50 percent.

- **Reducing Branch Delay:** In this technique the compiler detects the branch instructions and rearranges the machine language code sequence by inserting useful instructions that keep the pipeline operating without the interruption. It is similar to delayed load. The advantage of this technique is that lesser numbers instructions are needed to be discarded as compared to previous one. The most advantageous situation is the one where the branch condition does not depend on instructions immediately preceding it then you can proceed with these step till the result of branch condition is computed.
- **Dynamic Branch Prediction:** It would be useful to be able to predict whether or not a majority of the branches are taken or not taken. Thus, you require some additional logic to guess the outcome of a conditional branch instruction before it is executed. The pipeline then begins prefetching the instruction stream from the predicted path. This can be done in software, using intelligent compilers and can also be done at runtime. The most advantageous situation is one where the branch condition does not depend on instructions immediately preceding it, as shown in the following code fragment:

```
Loop : ADD R5, R5, R6 # One of the registers for branching condition
      is modified
```

```
      SUB R4, R3, R6 # Nothing important to the branch here
```

```
      Equal beq R5, R6, Loop
```

Here, the branch compares registers 5 and 6, which are last modified in the add instruction. You can therefore precompute the branch condition as sub r R5, R6, where r denotes a destination register. If r = 0, then you know the branch will be taken, and the runtime module (pipeline loader) can schedule the jump to the branch target address with full confidence that the branch will be taken.

- **Branch Target Buffer:** Another approach is to keep a history of branch statements, and to record what addresses these statements branch. This is with use of Branch Target Buffer (BTB). BTB is an associative memory which contains the address of previously executed branch instruction and the target instruction for the branch. It also stores next few instructions. When the pipeline found a branch instruction it views the BTB associative memory. If its address is available it prefetch from BTB and continue the available path after instruction in BTB. If the instruction is not in BTB, the pipeline shifts to new instruction and stores the target instruction in BTB. Thus, if branch instruction is previously executed then you can directly jump to next steps instead of inserting extra delays slots.
- (v) **Loop Buffer:** Although all the data dependencies are important to identify when designing parallel programs, loop carrying dependencies are particularly important since loops are possibly the most common target of parallelization efforts. Since, the vast majority of branches are used as tests of loop

## NOTES

## NOTES

condition, then you know that the branch will almost always jump to the loop back point. If the branch fails, then you know the loop is finished, and this happens only once per loop. Since, most loops are designed to have many iterations, branch failure occurs less frequently in loops than does taking the branch. Hence, you take a register file maintained by the instruction fetch segment of the pipeline. When a program loop is detected in the program, all the instructions (i.e., instructions inside the loop) are stored in loop buffer. Thus, program loop can be executed directly without the access to memory until mode is removed by final branching out. Thus, it makes good sense to assume that a branch will jump to the place that it jumped to before. However, in dense decision structures (e.g., nested or cascaded if statements), this situation does not always occur. In such cases, one might not be able to tell from the preceding branch whether or not the branching behaviour will be repeated. It is then reasonable to use a multi-branch lookahead.

- (vi) **Branch Delay Slot:** Another clever technique of making branches more efficient is the branch delay slot.

The concept of efficient branching has two parts. First, the branch target address is computed in the ID stage of the pipeline, to determine as early as possible the instruction to fetch if the branch succeeds. Since, this is done in the second stage of the pipeline, there is an instruction I following this (in the first or IF stage). After I moves to the ID stage, then the branch target (pointed to by either PC+4 or the BTA) is loaded into the IF stage. It is this instruction (I) that is called the Branch Delay Slot (BDS). In BDS, an instruction that does not have data dependencies on be safely placed with respect to (i) the branch condition, (ii) the instruction following the branch condition or (iii) the branch target. This ensures that, when the instruction J is executed (J is the instruction to which control is transferred after the branch condition is evaluated, whether J is pointed to by PC+4 or BTA), then the instruction I will have been executed previously, and the pipe will not have a stall where I would have been. As a result, the pipe will remain full throughout the branch evaluation and execution, unless an exception occurs.

---

## 4.11 FLYNN'S CLASSIFICATION

---

The classification based on the multiplicity of instruction streams and data streams in a computer system is referred to as Flynn's classification.

Parallel processing can be classified in a variety of ways. It can be classified on the basis of the internal organization of processors, the interlinked structure between processors or the flow of information throughout the system.

M.J. Flynn introduced another classification on the basis of instructions and data flow in a system; this classification is known as the Flynn's classification.

Flynn classified parallel computers into four categories based on how instructions process data. These categories are as follows:

- Single Instruction Stream, Single Data Stream (SISD) Computer
- Single Instruction Stream, Multiple Data Stream (SIMD) Computer
- Multiple Instruction Stream, Single Data Stream (MISD) Computer
- Multiple Instruction Stream, Multiple Data Stream (MMD) Computer

The normal operation of a computer is to fetch instructions from memory and execute them in a processor. The series of instructions read from the memory constitutes an instruction stream. The operations performed on the data in the processor constitute a data stream. Parallel processing may occur in the instruction stream, in the data stream or in both.

### Single Instruction Stream, Single Data Stream (SISD)

A computer with a single processor is called a Single Instruction Stream, Single Data Stream (SISD) computer. It represents the organization of a single computer containing a control unit, a processor unit and a memory unit. Instructions are executed sequentially and the system may or may not have internal parallel processing. Parallel processing can be accomplished by means of pipeline processing.

In such a computer, a single stream of instructions and a single stream of data are accessed by the processing elements from the main memory, processed and the results are stored back in the main memory. The SISD computer organization is shown in Figure 4.82.

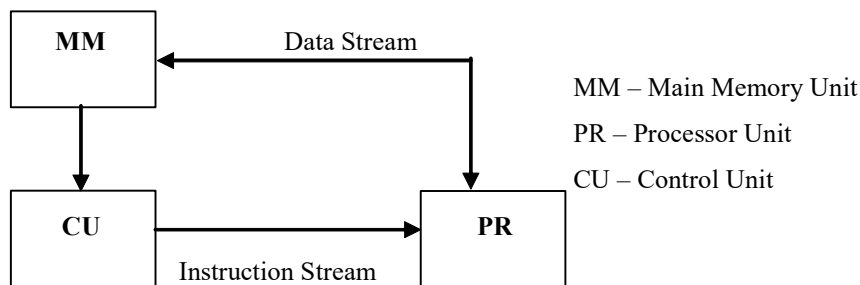


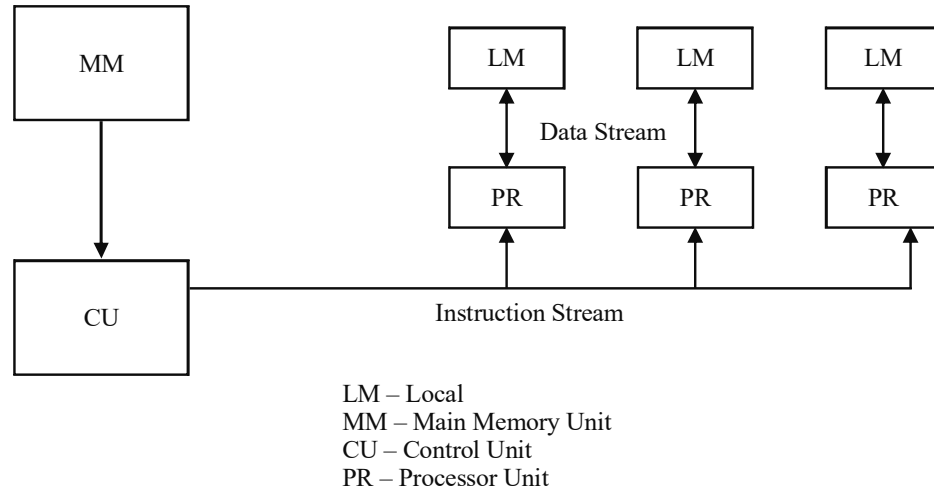
Fig. 4.82 SISD Organization

### Single Instruction Stream, Multiple Data Stream (SIMD)

It represents the organization of a computer which has multiple processors under the supervision of a common control unit. All processors receive the same instruction from the control unit but operate on different items of data. SIMD computers are used to solve many problems in science, which require identical operations to be applied to different data sets synchronously. Examples are, adding a set of matrices simultaneously, such as  $\sum_i \sum_k (a_{ik} + b_{jk})$ . Such computers are known as array processors. The SIMD computer organization is shown in Figure 4.83.

## NOTES

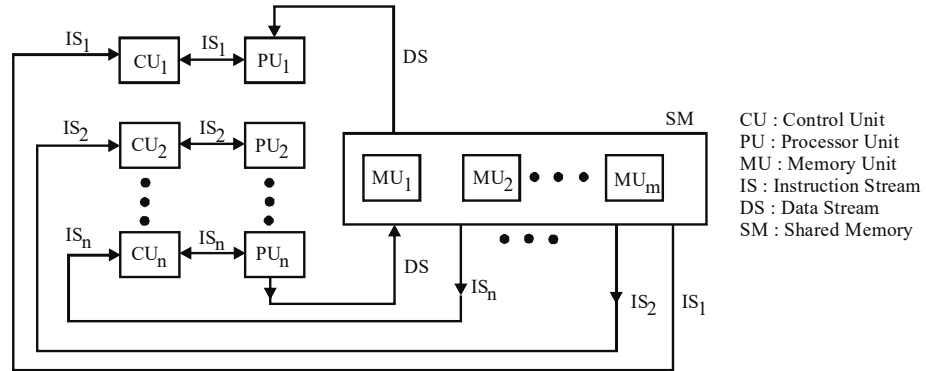
**NOTES**



*Fig. 4.83 SIMD Organization*

**Multiple Instruction Stream, Single Data Stream (MISD)**

It refers to the computer in which several instructions manipulate the same data stream concurrently. In the structure, different processing elements run different programs on the same data. This type of processor may be generalized using a two-dimensional arrangement of processing elements. Such a structure is known as systolic processor. The MISD computer organization is shown in Figure 4.84.



*Fig. 4.84 MISD Organization*

**Multiple Instruction Stream, Multiple Data Stream (MIMD)**

MIMD computers are general-purpose parallel computers. Their organization refers to a computer system capable of processing several programs at the same time. MIMD systems include all multiprocessing systems. The MIMD computer organization is shown in Figure 4.85.



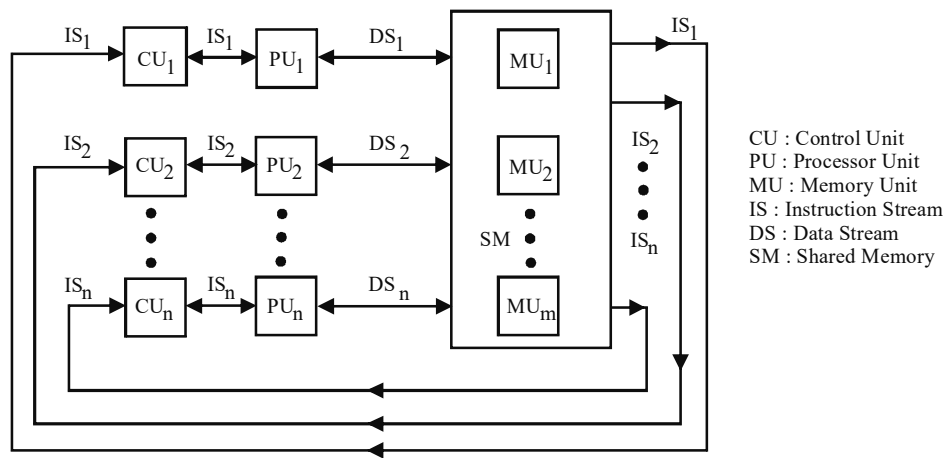


Fig. 4.85 MIMD Organization

**NOTES**

**4.11.1 Array Processors**

An array processor is a processor that is designed for performing calculations on a large-sized array of data. There are two types of array processors:

- Attached Array Processor
- SIMD Array Processor

An attached array processor is a peripheral device attached to a computer so that the performance of a computer can be improved for numerical computations. The purpose of the attached array processor is to improve the computer's performance by providing the functionality of vector processing for solving complex scientific problems.

Figure 4.86 shows the interconnection of an attached array processor with a computer.

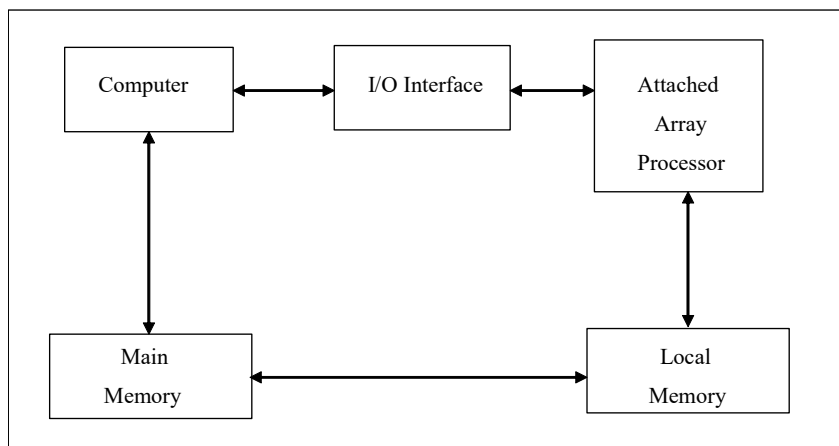


Fig. 4.86 Attached Array Processor

## NOTES

### SIMD Array Processor

An SIMD array processor has a single instruction multiple data stream organization that manipulates the common instruction by means of multiple functional units. This array processor consists of multiple Arithmetic and Logic Units (ALUs) that operate in parallel. ALUs work under the control of a common control unit performing the same operation and hence, achieve the single instruction multiple data stream organization.

The SIMD array processor consists of a set of processing elements where each Processing Element (PE) has its own local Memory (M) as shown in Figure 4.87. The processing elements may include the Arithmetic and Logic Unit (ALU), floating-point arithmetic unit and registers. The main memory of the CPU is used for the storage of the program. The operation of the processing element is controlled by the master control unit, whose main function is to decode the instructions and determine how the instruction is to be executed. Data operands are transferred to local memories. Each of the processing elements operates upon the data stored in its local memory.

Suppose we need to perform the vector addition  $c_i = a_i + b_i$ , for  $i = 1, 2, 3 \dots n$ . The master control unit first stores the  $i^{\text{th}}$  data element in a local memory  $M_i$ . It then gives the add instruction  $c_i = a_i + b_i$  to the processing elements causing the addition to take place simultaneously. The result of  $c_i$  is stored in the local memories. Thus, the whole process is performed in one cycle.

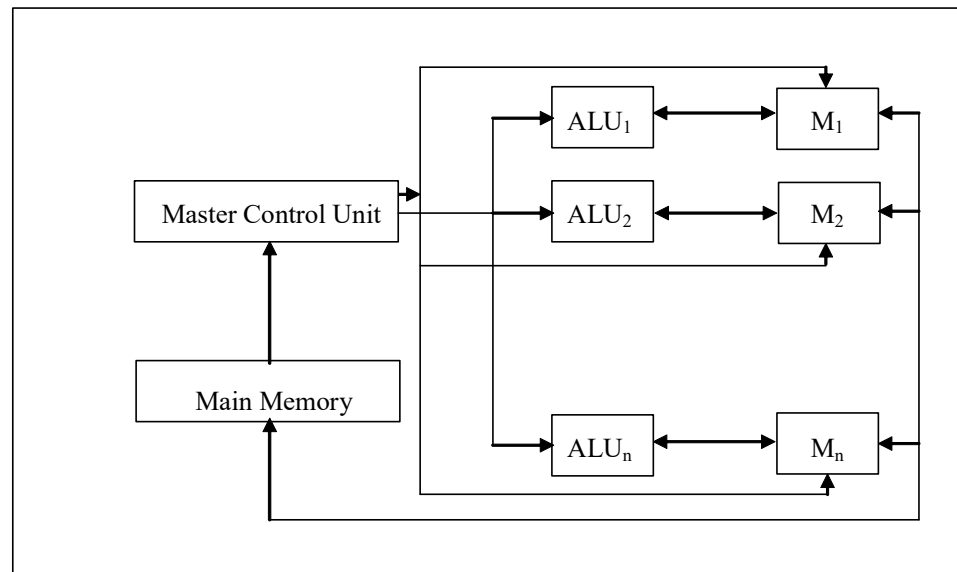


Fig. 4.87 SIMD Array Processor

### Check Your Progress

21. What is flash memory card?
22. Microcomputers memories are made up of which material?
23. Why is cache memory used?
24. What do you understand by virtual memory?
25. List the methods of parallelism.
26. When does data hazards occur?
27. How Flynn classified parallel computers into four categories?
28. Name the two types of array processors.

### NOTES

## 4.12 ANSWERS TO 'CHECK YOUR PROGRESS'

1. CPU tests and manipulates data and transfers information to and from other components, such as working memory, disk drive, monitor and keyboard.
2. The group of binary bits assigned to perform a specified operation is known as control word.
3. A stack is an ordered collection of items which permits the insertion or deletion of an item to occur only at one end. The insertion operation is known as push and the deletion operation is known as pop.  
A stack is also known as a Last-In-First-Out (LIFO) list. The stack can be considered as a storage method in which the items stored last are the first items to be removed.
4. The process of inserting an item into stack is known as a push operation. The process of deleting an item from a stack is known as a pop operation.
5. The stack pointer contains the address of the word that is currently on top of the stack, and which is a binary value.
6. A stack can also be implemented using the random access memory attached to the CPU. This can be implemented by assigning a portion of the memory for the stack operation, using the processor register as a stack pointer. The program counter indicates the address of the subsequent instruction stored in memory and the stack pointer indicates to the top of the stack.
7. The three basic notations are:
  - x + y Infix notation
  - + xy Prefix or polish notation
  - xy + Postfix or reverse polish notation
8. The different types of CPU organization are as follows:
  - Single Accumulator Organization
  - General Register Organization
  - Stack Organization

## NOTES

9. Immediate addressing is the simplest form of addressing where the operand is actually present in instruction, i.e., there is no operand fetching activity as the operand is given explicitly in the instruction.
10. Displacement addressing is a very powerful mode of addressing because it combines the capabilities of direct addressing and register indirect addressing.
11. Computer instructions are broadly classified into the following categories:
  - i. Data transfer instructions
  - ii. Data manipulation instructions
  - iii. Program control instructions
12. Addition, subtraction, multiplication and division are the four basic arithmetic operations.
13. RISC is a type of microprocessor that is designed with limited number of instructions.
14. A magnetic tape consists of a plastic ribbon with a magnetic surface. The data is stored on the magnetic surface as a series of magnetic spots. It has a large storage capacity of 2 to 8 GB and slow transfer rate of 160 kB/s to 1 MB/s.
15. Sensors are non-interactive types of devices, i.e. they are the devices which accept the non-online input and send this input data to computers.
16. MICR is a popularly used technique in the banking sector. All banks now issue cheques and drafts. As cheques enter an MICR machine, they pass through the magnetic field which causes the read head to recognize the character of the cheques. It has vastly helped the banking sector in authenticating the cheques.
17. As a new I/O device is designed on some new technology, it is required to make the device compatible with the processor. Designing an instruction set for every new device is not at all feasible.
18. Following are certain properties of a subroutine:
  - The subroutine call and return mechanism is automatic; therefore programmer will not execute it explicitly.
  - A subroutine can be called from more than one program. The top of stack will store the return address of the program calling it.
  - A subroutine call can appear in a subroutine, i.e., it can be nested.
19. Processor is not involved in I/O transfers in DMA. In other two techniques, on the other hand, each I/O transfer is performed by a set of instructions that are executed by CPU. So, with the DMA data transfer technique, the processor is available for other processing activities as it is not used for handling the data transfer activity. In the systems where the processor primarily uses cache, data transfer can take place in parallel, increasing overall system utilization.

20. The processor that communicates with remote terminals like telephone or any other serial communication media in serial fashion is called data communication processor (DCP).
21. A flash memory card is an electronic flash memory data storage device which is used in digital cameras, mobile computers, telephones, music players, video game consoles and other electronics.
22. Microcomputers memories are made of semiconductors fabricated on silicon chips.
23. Cache memory is used to store the data and information temporarily.
24. Virtual memory is a concept which permits the user to construct a program with size more than the total memory space available to it.
25. Parallelism has no advantages but this concept has different levels of effectiveness by using some methods. These methods are as follows:
  - Symmetric Multiprocessor (SMP)
  - Non-Uniform Memory Access (NUMA)
  - Uniform Memory Access (UMA)
  - Single Instruction Multiple Data (SIMD)
  - Multiple Instruction Multiple Data (MIMD)
26. Data hazards occur when the input is required for execution of an instruction depending on the output of the previous instruction still in the pipeline.
27. These categories are as follows:
  - Single Instruction Stream, Single Data Stream (SISD) Computer
  - Single Instruction Stream, Multiple Data Stream (SIMD) Computer
  - Multiple Instruction Stream, Single Data Stream (MISD) Computer
  - Multiple Instruction Stream, Multiple Data Stream (MIMD) Computer
28. The two types of array processors are: attached array processor and SIMD array processor.

## NOTES

---

### 4.13 SUMMARY

---

- CPU tests and manipulates data and transfers information to and from other components, such as working memory, disk drive, monitor and keyboard.
- The group of binary bits assigned to perform a specified operation is known as control word.
- A stack is an ordered collection of items which permits the insertion or deletion of an item to occur only at one end. The insertion operation is known as push and the deletion operation is known as pop.
- The conversion of an expression from the infix form to the reverse polish form must be done according to the operational hierarchy that follows for the infix notation.

## NOTES

- An instruction is a command given to a computer to perform a specified operation on some given data. The format in which the instruction is specified is known as instruction format.
- The number of addresses in the instruction can be reduced to two from three if the destination register is the same as one of the source registers.
- Addressing modes form the part of instruction set architecture. The instruction set is an important aspect of any computer organization. The Effective Address (EA) of an operand is the address of (or the pointer to) the main memory or register location in which the operand is contained.
- Immediate addressing is the simplest form of addressing where the operand is actually present in instruction. The simplest addressing mode where an operand is fetched from memory is direct addressing.
- Register addressing is a way of direct addressing where the address field refers to a register rather than the main memory address. Indirect addressing mode is used where the address of the operand is contained in register pair.
- In register indirect addressing mode, the operand field of an instruction holds the address of the address register to calculate the true address of the operand. Displacement addressing requires that the instruction should have two address fields, in which at least one is explicit.
- In index addressing mode, the address field references a main memory address and the reference register contains a positive displacement from that address. Indexing provides an efficient mechanism for performing iterative operations.
- The processor unit performs arithmetic and other data processing tasks as specified by a program.
- The control unit retrieves the instructions from a program (one by one), which are safely kept in the memory.
- Computer instructions are broadly classified into three different categories, data transfer instructions, data manipulation instructions and program control instructions.
- In microprogrammed organization the control unit is implemented through programming.
- For the execution of a program, the series of microinstructions must be determined which is different for different instructions.
- The essential components of a microprogrammed unit are microprogram sequencer, control address register, control memory, control buffer register and decoder.
- The Reduced Instruction Set Computer (RISC) pipeline instructs the verification in RISC cores.
- The peripheral devices can be thought of as transducers which can sense physical effects and convert them into machine-tractable data.

- An I/O interface is an entity that controls the data transfer from external device, main memory and/or CPU registers.
- In the interrupt driven techniques, the processor starts data transfer when it detects an interrupt signal which is issued when device is ready.
- The three types of exceptions are interrupts, traps and system calls.
- If a program requires any input or output, it lets the device controller or device to issue an interrupt.
- In DMA, the data is moved between a peripheral device and the main memory without any direct intervention of the processor.
- An IOP may be classified as a processor with the direct memory access capability that communicates with the I/O device.
- The data communication processor is an IOP that distributes and collects data from the remote terminals through telephone or other connection lines.
- Serial ports transfer data serially one bit at a time while parallel ports send and receive one byte/8-bit data at a time.
- RAM and ROM are considered as the two prime types of computer memory system in which RAM is considered as read write memory whereas ROM refers to read only memory. RAM is volatile whereas ROM is non-volatile memory.
- RAM is a volatile memory chip in which both read and write operations can be performed. Any random memory location can be accessed for information transfer to or from the memory and is called Read Write Memory (RWM).
- Dynamic Random Access Memory (DRAM) is single transistor memory cell that requires regular refreshes. This chip consists of small capacitors for each bit of memory.
- Static Random Access Memory (SRAM) is a volatile memory cell that does not require updates or periodic refresh cycles to keep the memory content intact. As compared to DRAM chip, SRAM chip is faster.
- Flash memory is a non-volatile computer storage chip that can be electrically erased and reprogrammed. It was developed from EEPROM (Electrically Erasable Programmable Read Only Memory) and must be erased in fairly large blocks before these can be rewritten with new data. The two types are NAND and NOR.
- Cache memory is used to store the data and information temporarily. It is implemented for Internet content by distributing it to multiple servers which are periodically refreshed. In fact, cache is a small and fast memory placed between the CPU and the main memory.
- The virtual memory permits the user to construct a program with size more than the total memory space available to it. This technique allows user to use the hard disk as if it is a part of main memory.
- The classification based on the multiplicity of instruction streams and data streams in a computer system is referred to as Flynn's classification.

## NOTES

- An array processor is a processor that is designed for performing calculations on a large-sized array of data.

## NOTES

---

### 4.14 KEY TERMS

---

- **CPU:** The ALU and CU of a computer system are jointly known as the CPU.
- **ALU:** The unit that does the arithmetic and logic operations needed for executing instructions.
- **Stack:** An ordered collection of items that allows the insertion or deletion of an item to occur only at one end.
- **Direct Addressing:** It is the simplest addressing mode where an operand is fetched from memory is direct addressing.
- **Data Manipulation Instructions:** The instructions that perform arithmetic, shift or logic operations to manipulate data
- **Microinstructions:** Each instruction is executed by a set of microoperations, termed as microinstructions.
- **Hard Disk:** A hard disk is one of the important I/O devices and is most commonly used as permanent storage device in any processor.
- **Bus Master:** It is the device that is allowed to initiate data transfer on the bus at any given time.
- **Polling:** It is the technique that identifies the highest priority resource by means of software.
- **Subroutine:** It is a self-contained program (piece of instruction code) that may be invoked or called by main program.
- **Cache Memory:** It is a small and fast memory placed between the CPU and main memory.
- **Parallel Computing:** Parallel computing means doing several tasks simultaneously, thus improving the performance of the computer system.
- **Array Processor:** A processor designed for performing calculations on large sized arrays of data.
- **SIMD Array Processor:** An array processor with a single instruction multiple data stream organization that manipulates the common instruction by means of multiple functional units.

---

### 4.15 SELF-ASSESSMENT QUESTIONS AND EXERCISES

---

#### Short-Answer Questions

1. Write the examples of output devices.
2. What is the purpose of including sign flag?



3. State the basic features of general register organization.
4. Define control world.
5. What do you mean by register stack?
6. What is an instruction formats?
7. What do you understand by register addressing?
8. Which mode of addressing is known as implied addressing?
9. What are data transfer instructions?
10. Which two tasks are performed to execute an instruction by microprogrammed control unit?
11. Define microinstructions.
12. What steps are executed in a microcode execution in one clock pulse?
13. What do you understand by peripheral devices?
14. Define I/O interface.
15. What is the function of memory bus?
16. What are the advantages of DMA?
17. Give the significance of IOP.
18. State the objective of virtual memory.
19. How does parallel processing work?
20. What are the two types of array processors? Explain each with suitable diagrams.

### Long-Answer Questions

1. What are the main components of CPU? Explain register level CPU organization with the help of diagram.
2. Define storage registers. Write the functions and prototypes used in transferring registers.
3. Instructions use operands and machine registers mainly in four different ways. Discuss these in brief.
4. What is the difference between zero-address, one-address and two address instructions?
5. Explain the role of stacks in programming.
6. Discuss all the characteristics of a good instruction format. Explain the structure of a typical instruction format.
7. Discuss addressing modes. What are the advantages and disadvantages of implied mode?
8. How displacement addressing is useful in addressing mode? Explain all the types of displacement addressing.
9. Describe the three data manipulation instructions.

### NOTES

## NOTES

10. Elaborate on microprogrammed control unit. Which components are used to implement the control process in the microprogrammed units?
11. Explain the functions of is Input-Output Interface. What are the types of command that an interface receive?
12. Describe the interrupt mechanism in detail.
13. Compare programmed I/O and DMA.
14. What is cache memory? Explain cache memory organization with the help of a diagram.
15. Explain Flynn's classification of the parallel computer architecture. Also discuss each class.

---

## 4.16 FURTHER READING

---

- Tanenbaum, Andrew S. 1999. *Structured Computer Organization*, 4th Edition. New Jersey: Prentice-Hall Inc.
- Mano, M. Morris. 1993. *Computer System Architecture*, 3rd Edition. New Jersey: Prentice-Hall Inc.
- Bartee, Thomas C. 1985. *Digital Computer Fundamentals*. New York: McGraw-Hill.
- Mano, M. Morris. 1979. *Digital Logic and Computer Design*. New Delhi: Prentice-Hall of India.
- Leach, Donald P. and Albert Paul Malvino. 1994. *Digital Principles and Applications*. New York: McGraw-Hill.
- Mano, M. Morris. 2002. *Digital Design*. New Delhi: Prentice-Hall of India.
- Kumar, A. Anand. 2003. *Fundamentals of Digital Circuits*. New Delhi: Prentice-Hall of India.
- Stallings, William. 2007. *Computer Organisation and Architecture*. New Delhi: Prentice-Hall of India.

---

# UNIT 5 PIPELINE, VECTOR PROCESSING AND MULTIPROCESSING

---

## NOTES

### Structure

- 5.0 Introduction
- 5.1 Objectives
- 5.2 Pipeline Processing
  - 5.2.1 Arithmetic Pipeline
  - 5.2.2 Instruction Pipeline
  - 5.2.3 Linear Pipeline
  - 5.2.4 RISC Pipelines
  - 5.2.5 Vector Processing
  - 5.2.6 Array Processing
- 5.3 Multiprocessors
  - 5.3.1 Interconnection Structure
  - 5.3.2 Characteristics of Multiprocessors
  - 5.3.3 Interprocess Arbitration
- 5.4 Interprocessor communication and Synchronization
  - 5.4.1 Racing Problem
  - 5.4.2 Problems of Critical Section
  - 5.4.3 Critical Section Algorithms
  - 5.4.4 Hardware Support for Mutual Exclusion
  - 5.4.5 Swap Instruction
  - 5.4.6 Binary Semaphore
  - 5.4.7 Implementation of Semaphores with a Waiting Queue
  - 5.4.8 Conditional Critical Region (CCR)
  - 5.4.9 Classical Problems in Concurrent Programming
  - 5.4.10 Readers and Writers Problem
  - 5.4.11 Deadlocks
  - 5.4.12 Resource Allocation Graph (RAG)
  - 5.4.13 Methods for Handling Deadlocks
  - 5.4.14 Introduction to File System and IO
  - 5.4.15 Organizing Files
- 5.5 Cache Coherence
- 5.6 Answers to 'Check Your Progress'
- 5.7 Summary
- 5.8 Key Terms
- 5.9 Self-Assessment Questions and Exercises
- 5.10 Further Reading

---

## 5.0 INTRODUCTION

---

A pipeline is a set of data processing elements connected in series, so that the output of one element is the input of the next one. Computer processor pipelining is sometimes divided into an instruction pipeline and an arithmetic pipeline. An instruction pipeline is a technique used in the design of computers and other digital electronic devices to increase their instruction throughput. The arithmetic pipeline represents the parts of an arithmetic operation that can be broken down and overlapped as they are performed.

## NOTES

A vector processor is a CPU which implements an instruction set containing instructions that operate on one-dimensional arrays of data known as vectors. Pipelining and vector processing speed up processing by performing multiple operations in parallel. Multiprocessors use two or more than two CPUs assembled in a single system unit. It refers to the execution of various software processes concurrently.

In this unit, you will study about the pipeline processing, vector processing, array processing, multiprocessors, interconnection structures, interprocessor arbitration, interprocessor communication and synchronization and cache coherence.

---

### 5.1 OBJECTIVES

---

After going through this unit, you will be able to:

- Understand pipelining
- Classify pipeline processors
- Explain principles of designing pipelined processors
- Know vector processing and its characteristics
- Illustrate pipelined vector processing methods
- Explain multiprocessors
- Get acquainted with characteristics of multiprocessors and interprocess arbitration
- Describe interprocessor communication and synchronization
- Define cache coherence

---

### 5.2 PIPELINE PROCESSING

---

The implementation technique, where multiple instructions are overlapped in execution is known as pipelining. The computer pipeline is divided in various stages each of which completes a part of an instruction in parallel. Each stage is connected one to the next; one to form a pipe in which instructions enter at one end, progress through the stages and exit at the other end.

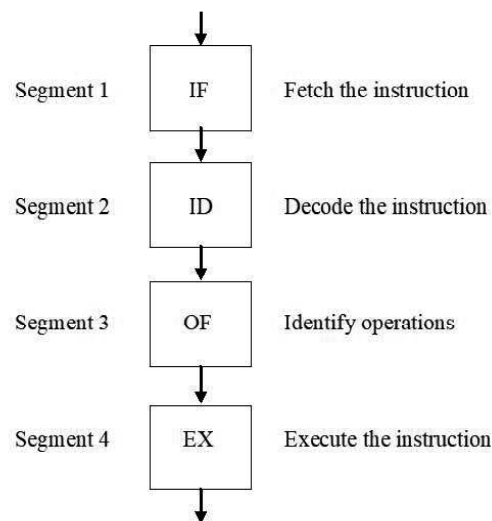
#### **Pipeline Computer**

Pipeline computers are those computers where a computer uses a sequence of stages (also known as segments) to execute an instruction. The computer's Central Processing Unit (CPU) contains one or more pipelines that interpret and execute instructions. It is the responsibility of the CPU to make sure that every stage of pipeline is always busy in executing an instruction. Once the instruction finishes its execution in one stage, the CPU passes that instruction to the next stage and gets another instruction from the previous stage, thereby moving several instructions along the pipeline simultaneously. Hence, this process is more efficient than the process in a non-pipeline computer where each instruction had to start its execution at the very first stage after the previous instruction finishes its execution.

In a digital computer, the execution of an instruction mainly involves the following four steps:

- Fetching the instruction from computer memory (IF)
- Decoding the fetched instruction (ID)
- Identifying the operation to be performed (OF)
- Executing the instruction (EX)

In a non-pipelined computer, normally these four steps (for a single instruction) must be completed before the next instruction is taken out from the memory for execution. However, in a pipelined computer, successive instructions are executed concurrently. The instructions are executed in different segments as shown in Figure 5.1.



*Fig. 5.1 Pipeline Computer*

In a pipelined computer, the instruction cycle consists of multiple pipeline cycles, where the flow of data (operands, intermediate or output results) from one segment to another segment takes place under a common clock pulse. Thus, the operations of all segments operate concurrently under a common clock cycle.

In a non-pipelined computer, the execution of a single instruction completes in four-clock cycles. However, in a pipeline computer, once a pipeline is filled, every next clock cycle will produce one output result. Thus, the instruction cycle is reduced to one-fourth of the original clock cycle in a pipeline computer.

Pipelining has been explained as a method of decomposing a sequential process into suboperations. Every subprocess is executed in a devoted segment, which operates parallelly with all other segments.

Pipelining is an effective method of increasing the execution speed of the processor, provided the following conditions are satisfied:

- It is possible to break up an instruction into a number of independent paths, each part taking nearly equal time to execute.
- There must be locality in instruction execution, i.e., instructions are executed in a sequence one after the other in the order in which they are

## NOTES

## NOTES

written. If instructions are not executed in a sequence but ‘jump around’ due to many branch instructions, then pipelining is not effective.

- Sufficient resources are available in a processor so that if a resource is required by the successive instructions in the pipeline, it is readily available.

A pipeline is basically seen as a compilation of processing parts through which binary data flows. Each segment performs partial processing, dictated by the way the task is partitioned. The result obtained from the computation in each segment is transferred to the next segment in the pipeline. The final result is obtained after data has passed through all segments. It is the characteristic of pipelines that several computations can be in progress in distinct segments at the same time.

The foremost way to view the pipeline framework is to think that each segment comprises an input register that holds the data information followed by a combinational circuit. The combinational circuit in the particular segment performs the suboperation. The result of the combinational circuit in a given segment is assigned to the input register of the next segment. A clock is assigned to every register after sufficient time has gone by to do all segment activity. Thus, in this manner, the information runs through the pipeline one step at a time.

Suppose we want to perform the combined multiply and add operations with a stream of numbers.

$$A_i * B_i + C_i, \text{ for } i = 1, 2, 3, 4, \dots, 7.$$

Each suboperation is to be executed in a dedicated segment in a pipeline structure. Each segment consists of one or two registers and a combinational circuit to carry out the operation as shown in Figure 5.2. R1, R2, R3, R4 and R5 are registers that receive new information with every clock pulse. Multiplier and adder are combinational circuits used in Figure 5.2. The suboperations done in every segment of the pipeline are as follows:

$R1 \leftarrow A_i,$	$R2 \leftarrow B_i$	Input $A_i$ and $B_i$
$R3 \leftarrow R1 * R2,$	$R4 \leftarrow C_i$	Multiply and input $C_i$
$R5 \leftarrow R3 + R4$		Add $C_i$ to product

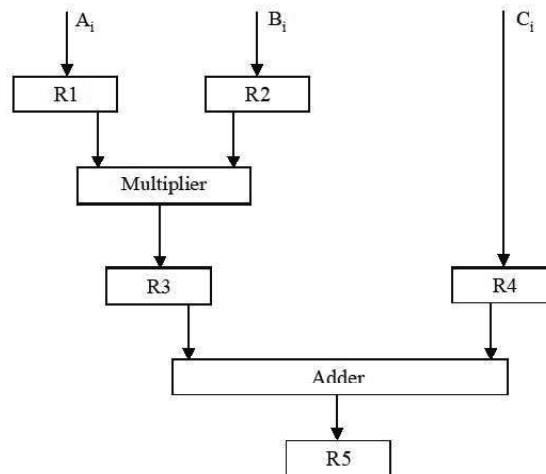


Fig. 5.2 Pipeline Processing

Table 5.1 Content of Registers in a Pipeline

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	$A_1$	$B_1$	–	–	–
2	$A_2$	$B_2$	$A_1 * B_1$	$C_1$	–
3	$A_3$	$B_3$	$A_2 * B_2$	$C_1$	$A_1 * B_1 + C_1$
4	$A_4$	$B_4$	$A_3 * B_3$	$C_1$	$A_2 * B_2 + C_2$
5	$A_5$	$B_5$	$A_4 * B_4$	$C_1$	$A_3 * B_3 + C_3$
6	$A_6$	$B_6$	$A_5 * B_5$	$C_1$	$A_4 * B_4 + C_4$
7	$A_7$	$B_7$	$A_6 * B_6$	$C_1$	$A_5 * B_5 + C_5$
8	–	–	$A_7 * B_7$	$C_1$	$A_6 * B_6 + C_6$
9	–	–	–	–	$A_7 * B_7 + C_7$

NOTES

The five registers are loaded with new data at every clock pulse, as illustrated in Table 5.1. With the first clock pulse, the value of  $A_1$  and  $B_1$  transfers into register R1 and R2, respectively. The product of R1 and R2 will transfer into R3 with the second clock pulse. At the same clock pulse, the value of  $C_1$  will transfer into register R4 and the value of  $A_2$  and  $B_2$  will transfer into register R1 and R2. The third clock pulse operates on all three segments simultaneously. It places  $A_3$  and  $B_3$  into R1 and R2, sends the product of R1 and R2 into R3, transfers  $C_2$  into R4 and places the sum of R3 and R4 into R5. It takes three clock pulses to fill the pipe and retrieve the first output from R5. From there on, each clock pulse produces a new output and moves the data one step down the pipeline. This will continue as long as the new input data flows into the system. When no additional data is accessible, the process continues till the last result comes out of the pipeline.

The general structure of a three-segment pipeline is shown in Figure 5.3. Each segment consists of a combinational circuit through which operands pass in a specified sequence. The combinational circuit  $C_i$  performs the required suboperation over the data flowing through the pipe. R1 is the register that will hold the intermediate results between the stages. A common clock is applied to all registers. Information flows through adjacent stages under the control of a common clock. Also, registers are placed in between two segments to separate them.

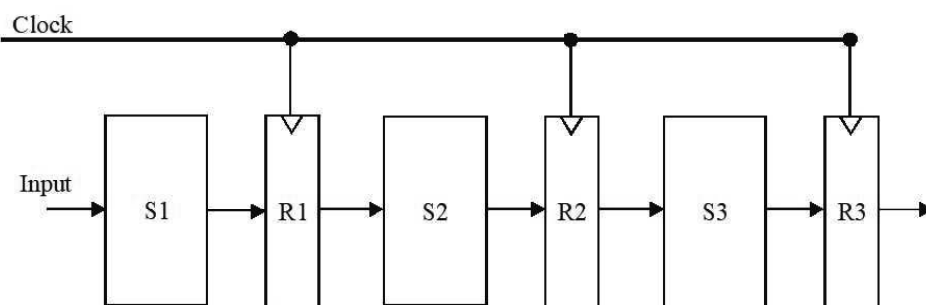


Fig. 5.3 Three Segment Pipeline

Space-Time Diagram for Pipeline

The behaviour of a pipeline can be illustrated with a space–time diagram. This is a diagram that shows the segment utilization as a function of time. The time in clock

NOTES

cycles, is given along the horizontal axis and the vertical axis gives the segment number.

		1	2	3	4	5	6	7	8	9	
Segment	1	T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>	T <sub>4</sub>	T <sub>5</sub>	T <sub>6</sub>				→ Clock Cycles
	2		T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>	T <sub>4</sub>	T <sub>5</sub>	T <sub>6</sub>			
	3			T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>	T <sub>4</sub>	T <sub>5</sub>	T <sub>6</sub>		
	4				T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>	T <sub>4</sub>	T <sub>5</sub>	T <sub>6</sub>	

Fig. 5.4 Space-Time Diagram

The diagram shown in Figure 5.4 shows six tasks T<sub>1</sub> through T<sub>6</sub> executed in four segments. Initially, task T<sub>1</sub> is handled by segment 1. After the first clock, segment 2 is busy with T<sub>1</sub>, while segment 1 is busy with task T<sub>2</sub>. Continuing in this manner, the first task T<sub>1</sub> is completed after the fourth clock cycle. From then onwards, the pipe completes a task with every clock cycle. No matter how many segments there are in the system, once the pipeline is full, it takes only one clock period to obtain an output.

Consider a case where a k-segment pipeline with a clock-cycle time t<sub>p</sub> is used to execute n tasks.

The first task T<sub>1</sub> needs the same time as kt<sub>p</sub> to finish its operation since there are k-segments in the pipe. The remaining (n - 1) task comes out of the pipe at the speed of one task per clock cycle. They will complete all the tasks after a time equivalent to (n - 1) t<sub>p</sub>.

Thus, to finish n tasks using a k-segment pipeline, the time needed will be k + (n - 1) clock cycles.

As for instance, to complete 6 tasks using the 4-segment pipeline, the time needed to finish all operations is 4 + (6 - 1) = 9 clock cycles.

Consider a non-pipeline unit that performs the same operation and takes time equal to t<sub>n</sub> to complete each task. The total time required for n tasks is nt<sub>n</sub>.

The speedup of a pipeline processing over an equivalent non-pipeline processing is defined by the ratio:

$$S = nt_n / (k + (n - 1))t_p$$

Therefore, as the amount of tasks increases, n becomes much bigger than k - 1, and k + n - 1 reaches the value of n. Thus, speedup becomes:

$$S = t_n / t_p$$

### 5.2.1 Arithmetic Pipeline

An arithmetic pipeline divides an arithmetic operation into suboperations for execution in the pipeline segments.

There are two areas of computer design where the pipeline organization is applicable.

- Arithmetic pipeline
- Instruction pipeline



In a very high-speed computer, arithmetic pipeline units are usually found. They are used to put into practice floating-point operations, multiplication of fixed-point numbers and related computations found in scientific problems.

Let us take an example of a pipeline unit for floating-point addition and subtraction.

The inputs to the floating-point adder pipeline are two normalized floating-point binary numbers.

$$X = A \times 2^a$$

$$Y = B \times 2^b$$

A and B are two fractions that represent the mantissas and a and b are the exponents. The floating-point addition or subtraction can be performed in four segments. The register R is placed between the segments to store intermediate results. The suboperations that are performed in the four segments are as illustrated in Figure 5.5.

- Compare the exponents
- Align the mantissas
- Add or subtract the mantissas
- Normalize the result

The exponents are compared by subtracting them to determine their differences. The larger exponent is selected as the exponent of the outcome. The exponent difference determines how many times the mantissa associated with the smaller exponent must be shifted to the right. This produces an alignment of the two mantissas. The two mantissas are added or subtracted in segment 3. The result is normalized in segment 4.

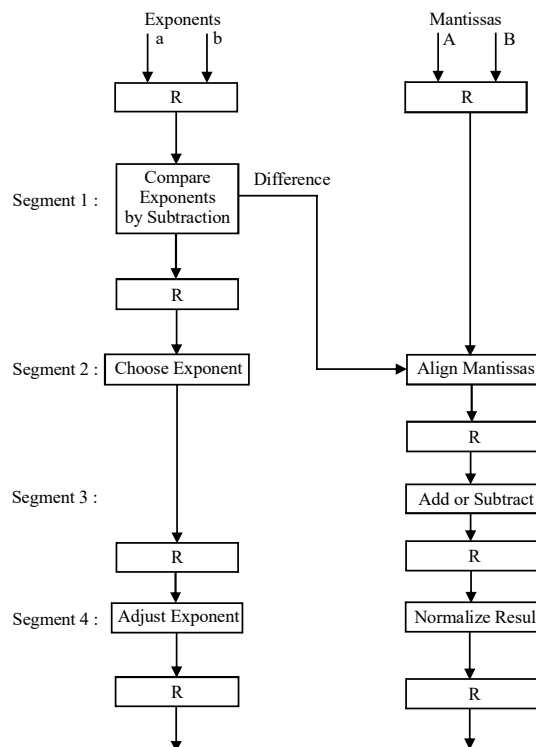


Fig. 5.5 Sub-operation performed in Four Segments

## NOTES

## NOTES

When an overflow occurs, the mantissa is shifted right and the exponent is incremented by one. If an underflow occurs, the number of leading zeros in the mantissa determines the number of left shifts in the mantissa and the number must be subtracted from the exponents.

Take, for example the following two normalized floating-point numbers:

$$X = 0.9703 \times 10^3$$

$$Y = 0.8200 \times 10^2$$

The two exponents are subtracted in the first segment to get  $3 - 2 = 1$ . The bigger exponent 3 is selected as the exponent of the outcome. The next segment shifts the mantissa of Y to the right to obtain:

$$X = 0.9703 \times 10^3$$

$$Y = 0.0820 \times 10^3$$

This aligns the two mantissas under the same exponent. The addition of the two mantissas in segment 3 produces the total:

$$Z = 1.0523 \times 10^3$$

The total is adjusted by normalizing the outcome so that it has a fraction with a non-zero first digit. Shifting the mantissa once to the right and increasing the exponent by one to get the result does this.

$$Z = 0.10523 \times 10^3$$

The different combinational circuits implemented in the floating-point pipelines are comparator, shifter, adder-subtractor, incrementer and decremter to carry out the different operations.

Suppose that the time delays of the four segments are  $t_1 = 55$  ns,  $t_2 = 75$  ns,  $t_3 = 100$  ns and  $t_4 = 70$  ns and the interface registers have a delay  $t_r = 15$  ns. Then, for pipeline floating-point adder-subtractor, the clock cycle is chosen to be  $t_p = t_3 + t_r = 115$  ns. In an equivalent non-pipeline floating point adder-subtractor, the delay time will be  $t_n = t_1 + t_2 + t_3 + t_4 + t_r = 315$  ns. Thus, the pipelined adder has a speedup of  $315/115 = 2.74$  over the non-pipelined adder.

Pipeline arithmetic is used in very high-speed computers specially involved in scientific computations. It is the basic principle behind vector processor and array processor. They are used to implement floating point operations, multiplication of fixed point numbers and similar computations encountered in computation problems. These computation problems can easily be decomposed in suboperations. Arithmetic pipelining is well implemented in the systems involved with repeated calculations such as calculations involved with matrices and vectors. Let us consider a simple vector calculation like  $A[i] + b[i] * c[i]$  for  $I = 1, 2, 3, \dots, 8$ .

The above operation can be subdivided into three segment pipelines. Each segment has some registers and combinational circuits. Segment 1 loads contents of  $b[i]$  and  $c[i]$  in register R1 and R2, segment 2 loads  $a[i]$  content to R3 and multiplies content of R1, R2 and stores them in R4 and finally segment 3 adds content of R3 and R4 and stores it in R5 as shown in Table 5.2.

**Table 5.2** Content of Registers in Pipeline

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	B1	C1	-	-	-
2	B2	C2	B1*C1	A1	
3	B3	C3	B2*C2	A2	A1+ B1*C1
4	B4	C4	B3*C3	A3	A2+ B2*C2
5	B5	C5	B4*C4	A4	A3+ B3*C3
6	B6	C6	B5*C5	A5	A4+ B4*C4
7	B7	C7	B6*C6	A6	A5+ B5*C5
8	B8	C8	B7*C7	A7	A6+ B6*C6
9			B8*C8	A8	A7+ B7*C7
10					A8+ B8*C8

**NOTES**

To illustrate the operation principles of a pipeline computation, the design of a pipeline floating point adder is given. It is constructed in four stages. The inputs are as follows:

$$A = a \times 2^p$$

$$B = b \times 2^q$$

Where a and b are two fractions and p and q are their exponents and here base 2 is assumed.

To compute the sum

$$C = A + B = c \times 2^r = d \times 2^s$$

Operations performed in the four pipeline stages are specified as follows:

1. Compare the two exponents p and q to reveal the larger exponent  $r = \max(p, q)$  and to determine their difference  $t = p - q$
2. Shift the right fraction associated with the smaller exponent by t bits to equalize the two components before fraction addition.
3. Add the preshifted fraction with the other fraction to produce the intermediate sum fraction c where  $0 \leq c < 1$ .
4. Count the number of leading zeroes, say u, in fraction c and shift left c by u bits to produce the normalized fraction sum  $d = c \times 2^u$ , with a leading bit 1. Update the large exponent s by subtracting  $s = r - u$  to produce the output exponent.

Figure 5.6 show how pipeline can be implemented in floating point addition and subtraction. Segment 1 compares the two exponents by subtraction. In segment 2 you choose the larger exponents. The larger exponent aligns the other mantissa by viewing the difference between two smaller number mantissa which are shifted to right by difference amount. Segment 3 performs addition or subtraction of mantissa while segment 4 normalizes the result. Various registers R are used to hold intermediate results.

NOTES

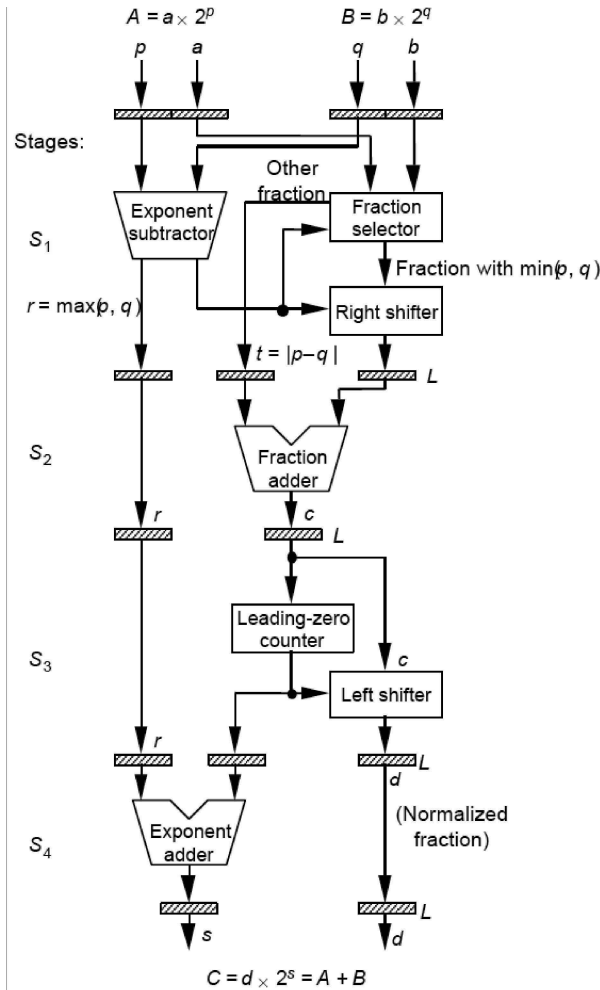


Fig. 5.6 Implementation of Pipeline

In order to implement pipelined adder you need extra circuitry but its cost is compensated if you implement it for large number of floating point numbers. Operations at each stage can be done on different pairs of inputs.

**Example 5.1:** How fast will be addition of 1000 floating point vector element using above pipeline as compared to non-pipeline adder?

**Solution:** The above pipeline has four segments and let us assume each segment requires one clock cycle time to execute.

To add 1000 pairs of numbers, without a pipelined adder would require 4000 cycles.

With 4-stage pipelined adder, the last sum will appear after 1000 + 4 cycles, so the pipeline is 4000/ 1004 = 3.98 times faster.

Lets take another example of multiplication.

The multiplication of 2 fixed point numbers is done by repeated add-shift operations, by using ALU which has built-in add and shift functions. Multiple number additions can be realized with a multilevel tree adder. The conventional Carry Propagation Adder (CPA) adds 2 input numbers say A and B to produce one output number called the sum A+B Carry Save Adder (CSA) receives three input numbers, say A,B and D and two output numbers, the sum S and the carry vector C.

A	=		1	1	1	1	0	1
B	=		0	1	0	1	1	0
D	=		1	1	0	1	1	1
C	=	1	1	0	1	1	1	
S	=		0	1	1	1	0	0
A+B+D	=	1	1	1	0	0	1	0

## NOTES

A CSA can be implemented with a cascade of full adders with the carry-out of a lower stage connected to the carry-in of a higher stage. A carry-save adder can be implemented with a set of full adders with all the carry-in terminals serving as the input lines for the third input number D and all the carry-out terminals serving as the output lines for the carry vector C.

This pipeline is designed to multiply two 6-bit numbers. There are five pipeline stages.

The first stage is for the generation of all  $6 \times 6 = 36$  immediate product terms, which form the six rows of shifted multiplicands. The six numbers are then fed into two CSAs in the second stage. In total, four CSAs are interconnected to form a three level merges the sum vector S and the carry vector C. The final stage uses a CPA which adds the two numbers C and S to produce the final output of the product  $A \times B$  (Refer Figure 5.7).

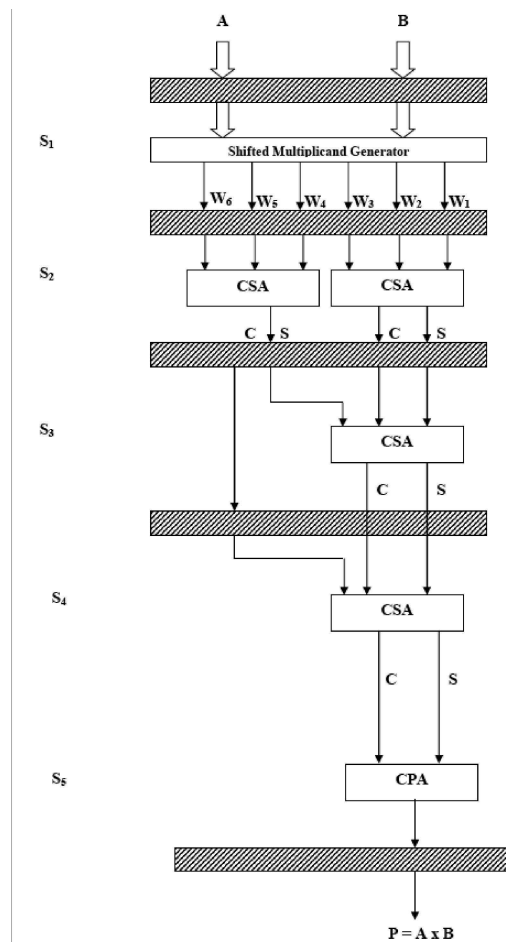


Fig. 5.7 Multiplication Procedure

## 5.2.2 Instruction Pipeline

An instruction pipeline operates on a stream of instructions by overlapping the fetch, decode and execute phases of the instruction cycle.

### NOTES

An instruction pipeline understands the consecutive instruction from the memory when the preceding instruction was being executed in the other segments. In this way, we can overlap instructions in the fetch and execute phases and perform operations simultaneously. The instruction mainly involves the following sequences of steps:

- **Instruction Fetch:** Fetch the instruction from memory.
- **Instruction Decode:** Decode the instruction.
- **Calculate Address:** Calculate the effective address of operands.
- **Operand Fetch:** Fetch the operands from memory.
- **Operation Execution:** Execute the instruction.
- **Result Storage:** Store the result in proper place.

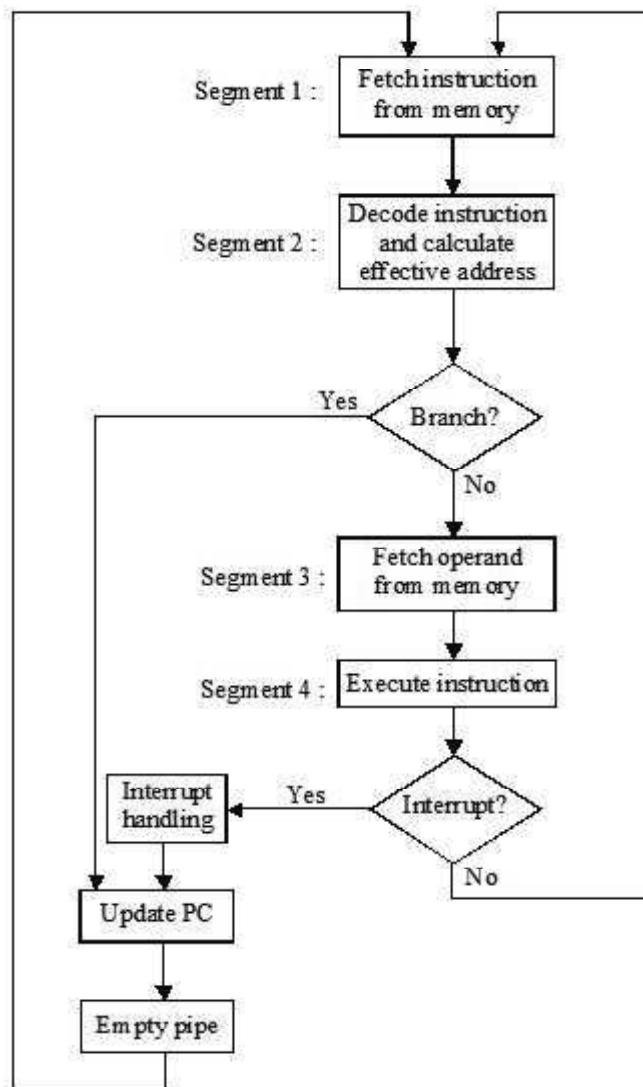
In a non-pipelined computer, all the above steps are performed for executing an instruction and then the next instruction is fetched from memory for execution. However, in a pipelined computer, these steps are performed in different segments. Suppose one segment is busy with fetching the instruction and at the same time the other segment is decoding another instruction. The instruction-fetching segment, the instruction-decoding segment, the operand-fetching segment and the execution segment would operate simultaneously in a pipelined computer. Some segments may be skipped for certain operations; for example a register mode instruction does not need an effective address calculation. Also, it may be possible that two or more segments might need memory access simultaneously, thus one segment has to wait while another segment is busy with memory access.

Let us take an example of four segment instruction pipeline as shown in Figure 5.8 where the decoding of the instruction and calculation of effective address are combined into one segment. Also, the execution of instruction is combined with the storing of result into one segment as nearly all the instruction places the outcome into a processor register after executing the instruction.

If an instruction in the sequence has a branch instruction, then that instruction causes a branch out of the normal sequence. In this situation, the pending operations of the last two segments are completed and information stored in the instruction buffer is deleted. Similarly, an interrupt request causes the pipeline to empty and start again from a new address value. Figure 5.8 shows the working of instruction pipeline. Horizontal axis shows the time that is divided into steps of equal period. Vertical axis shows the instructions. The four segments are abbreviated as follows:

- FI for fetch an instruction
- DA for decode instruction and calculate effective address
- FO for fetch the operand
- EX for execute the instruction

**NOTES**



*Fig. 5.8 Instruction Pipeline*

In addition, the processor has separate memories for instructions and data so that the operations in FI and FO proceed in time. Each segment operates on different instructions when there is no branch instruction. Thus, in the execution operation, instruction 1 is executed in segment EX, instruction 2 is executed in segment FO, instruction 3 be being decoded in segment DA and instruction 4 is in segment FI. Let the instruction 3 be a branch instruction. When this instruction is decoded in segment DA, the transfer of other instructions from FI to DA is stopped until the branch instruction is executed in Step 6. A new instruction is then fetched in Step 7 and the pipeline continues until a new branch instruction is faced again (Refer Figure 5.9).

**NOTES**

Step :	1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction 1	FI	DA	FO	EX									
		FI	DA	FO	EX								
(Branch) 3			FI	DA	FO	EX							
4				FI	-	-	FI	DA	FO	EX			
5					-	-	-	FI	DA	FO	EX		
6									FI	DA	FO	EX	
7										FI	DA	FO	EX

*Fig. 5.9 Timing Diagram*

Normally, the instruction pipeline deviates from its normal execution in three different situations:

- **Resource conflicts:** When two segments access the same memory location at same time, it is said to be resource conflicts. Employing different memories, one for instructions and other for data may solve this type of conflicts.
- **Data dependency:** It happens when an instruction requires the outcome of the earlier instructions, but is not yet accessible.
- **Branch difficulties:** This arises from branch and other commands that modify the value of PC.

**5.2.3 Linear Pipeline**

Multiple processors are cascaded linearly in a linear pipeline processor. Pipelining is a technique that decomposes any sequential process into smaller subprocesses, which are independent of each other so that each subprocess can be executed in a special dedicated segment and all these segments operate concurrently. Thus, the whole task is partitioned into independent tasks and these subtasks are executed by a segment. The result obtained as an output of a segment (after performing all computation in it) is transferred to next segment in pipeline and the final result is obtained after the data have been through all segments. So, it could be understood each segment consists of an input register followed by a combinational circuit. This combinational circuit performs the required sub-operation and register holds the intermediate result. The output of one combinational circuit is given as input to the next segment.

The concept of pipelining in computer organization is analogous to an industrial assembly line. As in industry, pipelining also has different divisions like manufacturing, packing and delivery division. Thus, pipeline results in speeding the overall process. Pipelining can be effectively implemented for systems having following characteristics:

- The system should repeatedly execute a basic function.
- The basic function must be divisible into independent stages such that each stage has minimal overlap.
- The complexity of the stages should be roughly similar.

The pipelining in computer organization has basic flow of information. To understand how it works for the computer systems let us consider a process



which involves four steps/segment and the process is to be repeated six times. If a single step takes  $t$  nsec time, then time required to complete one process is  $4t$  nsec and to repeat it 6 times we require  $24t$  nsec.

Now let us see how problem works behaves with pipelining concept. This can be illustrated with a space-time diagram shown in Figure 5.10. It shows the segment utilization as function of time. Let us assume there are six processes to be handled (represented in Table 5.3 as P1, P2, P3, P4, P5 and P6) and each process is divided into four segments (S1, S2, S3, S4). For sake of simplicity, assume that each segment takes equal time to complete the assigned job, i.e., equal to one clock cycle. The horizontal axis displays the time in clock cycles and vertical axis gives the segment number. Initially, process 1 is handled by segment 1. After the first clock segment 2 handles process 1 and segment 1 handles new process P2. Thus, first process will take four clock cycles and remaining processes will be completed one process each clock cycle. Thus, for this example total time required to complete whole job will be 9 clock cycles (with pipeline organization) instead of 24 clock cycles required for non-pipeline configuration.

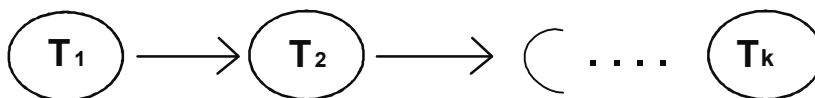
**NOTES**

*Table 5.3 Space-Time Diagram for pipeline*

	1	2	3	4	5	6	7	8	9
P1	S1	S2	S3	S4					
P2		S1	S2	S3	S4				
P3			S1	S2	S3	S4			
P4				S1	S2	S3	S4		
P5					S1	S2	S3	S4	
P6						S1	S2	S3	S4

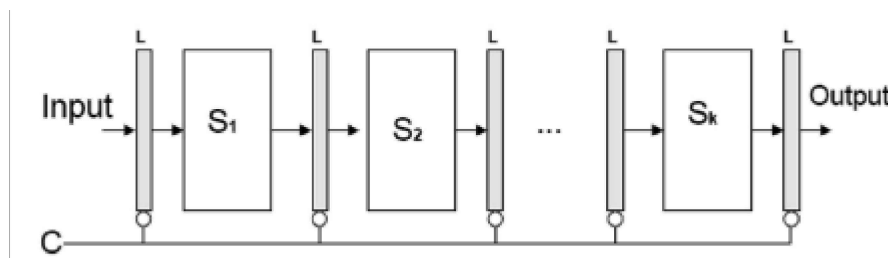
In Figure 5.10, let  $T$  be a task which can be partitioned into  $K$  subtasks according to the linear precedence relation:

$T = \{T_1, T_2, \dots, T_k\}$ , i.e., a subtask  $T_j$  cannot start until  $\{T_i \forall i \leq j\}$  are finished. This can be modelled with the linear precedence graph (refer Figure 5.10):



*Fig. 5.10 Linear Precedence Graph*

A linear pipeline, i.e., the precedence graph shows no feedback, i.e., iterative cycle or looping can always be constructed to process a succession of subtask with a linear precedence graph (Refer Figure 5.11).



*Fig. 5.11 Basic Structure and Control of a Linear Pipeline*

## NOTES

Processor ( $L$ =latch,  $C$ =clock,  $S_i$ =the  $i$ th stage).

- Stages are pure combinational circuits used for processing.
- Latches are fast registers to hold the intermediate data between the stages.
- Informational flow is controlled by a common clock with clock period 't', and the pipeline runs at a frequency of  $1/t$
- $t$  is selected as:  $t = \text{MAX}\{t_i\} + t_L = t_M + t_L$  where,  $t_i$ = propagation delay of stage  $S_i$ ,  $t_L$ =latch delay.
- Pipeline clock period is controlled by the stage with the max delay.
- Unless the stage delays are balanced, one big and slow stage can slow down the whole pipe.

### Speed-Up Ratio

The speed-up ratio is the ratio between maximum time taken by non-pipeline processes over the processes that use pipeline. Thus, in general if there are  $n$  processes and each process is divided into  $k$  segments (subprocesses), the first process will take  $k$  segments to complete the processes. But once the pipeline is full, that is the first process is complete, it will take only one clock period to obtain an output for each process. Thus, first process will take  $k$  clock cycles and the remaining  $n-1$  processes will emerge from the pipe at the one process per clock cycle thus total time taken by remaining processes will be  $(n-1)$  clock cycle time.

Let  $t_p$  be the one clock cycle time.

The time taken for  $n$  processes having  $k$  segments in pipeline configuration will be  $= k*t_p + (n-1)*t_p = (k+n-1)*t_p$ , the time taken for one process is  $t_n$  thus the time taken to complete  $n$  process in non pipeline configuration will be  $= n*t_n$

Thus, speed-up ratio for one process in non-pipeline and pipeline configuration is  $= n*t_n / (k+n-1)*t_p$ . If  $n$  is very large compared to  $k$  then  $= t_n / t_p$ , if a process takes same time in both case with pipeline and non pipeline configuration than  $t_n = k*t_p$ . Thus, speed up ratio will be  $S_k = k*t_p / t_p = k$ .

Theoretically, maximum speed-up ratio will be  $k$  where  $k$  is the total number of segments in which the process is divided. The following are various limitations due to which any pipeline system cannot operate at its maximum theoretical rate, i.e.,  $k$  (speed-up ratio).

1. Different segments take different times to complete their suboperations, and pipelining clock cycle must choose equal time delay of the segment with maximum propagation time. Thus, all other segments have to waste time waiting for the next clock cycle. The possible solution for improvement here if possible can be to subdivide the segment into different stages, i.e., increase the number of stages and if segments can not be subdivided then use multiple resources for segment causing maximum delays so that more than one instruction can be executed in to different resources and overall performance improves.
2. Additional time delay, may be introduced because of extra circuitry or additional software requirement that are needed to overcome various

hazards. The result should be stored in intermediate registers. Such delays are not found in non-pipeline circuit.

3. Further pipelining can be of maximum benefit if the whole process can be divided into suboperations which are independent of each other. But, if there is some resource conflict or data dependency, i.e., an instruction depends on the result of previous instruction which is not yet available than instruction has to wait till that result becomes available or conditional or non conditional branching, i.e., the bubbles or time delay is introduced.

## NOTES

### Efficiency

The efficiency of linear pipeline is measured by the percentage of time when a processor is busy over the total time taken, i.e., sum of busy time plus idle time. Thus, if  $n$  is the number of task,  $k$  is the stage of pipeline and  $t$  is the clock period then efficiency is given by:

$$\eta = n / [k + n - 1]$$

Thus, the larger the number of tasks in pipeline, the more will the pipeline be busy, hence, better will be its efficiency. It can be easily seen from expression as  $n \rightarrow \infty$ ,  $\eta \rightarrow 1$ .

$$\eta = S_k / k$$

Thus, efficiency of the pipeline is the speed-up divided by the number of stages or one can say actual speed ratio over ideal speed-up ratio.

### Throughput

The number of tasks completed by a pipeline per unit time are called throughput. This represents computing power of pipeline. We define throughput as

$$W = n / [k * t + (n - 1) * t] = \eta / t$$

In ideal case as  $\eta \rightarrow 1$  the throughput is equal to  $1/t$  that is equal to frequency. Thus, maximum throughput obtained is one output per clock pulse.

**Example 5.2:** A non-pipeline system takes 60 ns to process a task. The same task can be processed in six segment pipeline with a clock cycle of 10 ns. Determine the speed-up ratio of the pipeline for 100 tasks. What is the maximum speed that can be achieved?

**Solution:** Total time taken by non-pipeline to complete 100 tasks is =  $100 * 60 = 6000$  ns

Total time taken by pipeline configuration to complete 100 tasks is  
=  $(100 + 6 - 1) * 10 = 1050$  ns

Thus, speed-up ratio will be =  $6000 / 1050 = 4.76$

The maximum speed-up that can be achieved for this process is  
=  $60 / 10 = 6$

Thus, if total speed of non-pipeline process is same as that of total time taken to complete a process with pipeline, then maximum speed-up ratio is equal to number of segments.

## NOTES

**Example 5.3:** A non-pipeline system takes 50 ns to process a task. The same task can be processed in a six-segment pipeline with a clock cycle of 10 ns. Determine the speed-up ratio of the pipeline for 100 tasks. What is the maximum speed-up that can be achieved?

**Solution:** Total time taken by for non-pipeline to complete 100 tasks is =  $100 * 50 = 5000$  ns  
Total time taken by pipeline configuration to complete 100 task is  
=  $(100 + 6 - 1) * 10 = 1050$  ns  
Thus, speed up ratio will be =  $5000 / 1050 = 4.76$   
The maximum speedup that can be achieved for this process is  
=  $50 / 10 = 5$

**Example 5.4:** Suppose a process is subdivided into 4 sub tasks such that first and third task take 40ns to complete while second and fourth task take 50ns and 20ns, respectively. Calculate the speed-up the pipeline for 10 tasks and again for 100 tasks. What is maximum speed-up that can be achieved?

**Solution:** Total time for 10 tasks without pipeline  
1 task  $(40 + 50 + 40 + 20) = 150$ ns  
10 tasks =  $150 * 10 = 1500$  ns  
Total time with pipeline for 10 tasks is =  $(10 + 4 - 1) * 50 = 13 * 50 = 650$ ns  
Speed-up ratio for 10 tasks =  $1500 / 650 = 2.3$   
For 100 tasks without pipeline =  $150 * 100 = 15000$  ns  
Total time with pipeline for 100 tasks =  $(100 + 4 - 1) * 50 = 103 * 50 = 5150$  ns  
Speed-up ratio for 100 tasks =  $15000 / 5150 = 2.9$   
Maximum speed-up ratio for n process =  $150 * n / (n + 4 - 1) * 50$   
If n is very large, then  $n + 4 - 1 \sim n$   
Speed-up ratio =  $150 * n / n * 50 = 3$

**Example 5.5:** Consider a non-pipelined machine having 6 execution stages each of lengths 50 ns, 40 ns, 80 ns, 30 ns 50 ns and 50 ns. How much time does it take to execute 100 instructions without pipeline? In order to perform pipelining on this machine an overhead of the clock skew of 5 ns is added to each execution stage. Calculate the instruction latency on the pipelined machine? How much time does it take to execute 100 instructions? What is the speed-up obtained from pipelining?

**Solution:** Without pipeline total execution time for one stage =  $50 + 40 + 80 + 30 + 50 + 50 = 300$  ns  
Time to execute 100 instructions =  $100 * 300 = 30000$  ns  
With pipeline implementation, the length of the pipe stages must all be the same, i.e., the speed of the slowest stage plus overhead. With 5ns overhead it comes to:  
The length of pipelined stage = MAX (lengths of non-pipelined stages) + overhead =  $80 + 5 = 85$  ns

Instruction latency for one instruction = 85 ns

Time to execute 100 instructions =  $85 \times 6 \times 1 + 85 \times 1 \times 99 = 510 + 8415 = 8925\text{ns}$

Speed-up =  $30000 / 8925 = 3.38$

The two areas where pipeline organization is most commonly used are arithmetic pipeline and instruction pipeline. An arithmetic pipeline is a pipeline in which different stages of an arithmetic operation are handled along with the stages of a pipeline, i.e., it divides the arithmetic operation into suboperations. An instruction pipeline operates on a stream of instructions by overlapping the fetch, decode and execute phases of the instruction cycle as different stages of pipeline. RISC architecture supports pipelining more than a CISC architecture does. Pipelines are also used to compress and transfer video data. They are also used as specialized hardware to perform graphics display tasks. Pipelines and pipelining also applies to computer memory controllers and moving data through various memory staging places.

There are three prime disadvantages of pipeline architecture.

1. The first is complexity.
2. Many intermediate registers are required to hold the intermediate information as output of one stage will be input of next stage. These are not required for single unit circuit, thus, it is usually constructed entirely as combinational circuit.
3. The third disadvantage is its inability to continuously run the pipeline at full speed, i.e., the pipeline stalls for some cycle. There are phenomena called pipeline hazards which disrupt the smooth execution of the pipeline. If these hazards are not handled properly, they may give wrong result. Often it is required to insert delays in the pipeline flow in order to manage these hazards; such delays are called bubbles. Often it is managed by using special hardware techniques while sometime using software techniques such as compiler or code reordering, etc. Various types of pipeline hazards include:
  - Structural hazards that happen due to hardware conflicts.
  - Data hazards that happen due to data dependencies.
  - Control hazards that happen when there is change in flow of statement like to branch, jump or any other control flow changes conditions.
  - Exception hazard that happen due to some exception or interruption occur while execution in a pipeline system.

#### 5.2.4 RISC Pipelines

An efficient way to use instruction pipeline is a characteristic feature of RISC architecture. A Reduced Instruction Set Computer (RISC) processor pipeline operates in much the same way, although the stages in the pipeline are different. The length of the pipeline is dependent on the length of the longest step. Because RISC instructions are simpler than those used in pre-RISC processors (now called CISC, or Complex Instruction Set Computer), they are more conducive to

#### NOTES

## NOTES

pipelining. While CISC instructions vary in length, RISC instructions are all of the same length and can be fetched in a single operation. Ideally, each of the stages in an RISC processor pipeline should take 1 clock cycle so that the processor finishes an instruction for each clock cycle and averages one Cycle Per Instruction (CPI). Hence, RISC can achieve pipeline segments, requiring just one clock cycle, while CISC may use many segments in its pipeline, with the longest segment requiring two or more clock cycles.

As most RISC data manipulation operations have register to register operations, an instruction cycle has the following two phases.

I : Instruction fetch

E : Execute (performs an ALU operation with register input and output)

### Load and Store Instructions

The data transfer instructions in RISC are limited to load and store. These instructions use register indirect addressing and require three stages in pipeline, which are as follows:

I : Instruction fetch

A : Decode, evaluate effective address

E : Register-to-Memory or Memory-to-Register

To prevent conflicts between memory access, to fetch an instruction and to load or store operand, most RISC machines use two separate buses with two memories: one for storing the instruction and the other for storing data.

Another feature of RISC over CSIC as far as pipelining is considered is compiler support. Instead of designing hardware to handle the data dependencies and branch penalties, RISC relies on efficiency of the compiler to detect and minimize the delay encountered with these problems.

### Data Manipulation Instructions

While different processors have different numbers of steps, lets us consider a three segment Instruction pipeline. These segments are as follows:

I : Instruction fetch

A : Decode, Read Registers, ALU Operations

E : Execute instruction, Write a Register

The 'I' segment fetches the instruction from memory and decodes it. The ALU is used for three different functions, which are data manipulation, effective address calculation for LOAD and STORE operations or calculation of the branch address for a program control instruction depending on the type of instruction. The E segment directs the output of the ALU to one of three destinations, i.e., a destination register or effective address to a data memory for loading or storing or the branch address to program counter, depending upon decode instruction.

### Program Control Instructions

I : Instruction Fetch

A : Decode, Evaluate Branch Address

E : Write Register (PC)

## Delayed Load

Compiler analyses the instructions before and after the branch and rearranges the program sequence by inserting useful instructions in the delay steps (refer Figure 5.12).

Using No-Operation (NOP) instructions

Clock cycles:	1	2	3	4	5	6	7	8	9	10
1. Load	1	A	E							
2. Increment		I	A	E						
3. Add			I	A	E					
4. Subtract				I	A	E				
5. Branch to X					I	A	E			
6. NOP						I	A	E		
7. NOP							I	A	E	
8. Instr. in X								I	A	E

Rearranging the instructions

Clock cycles:	1	2	3	4	5	6	7	8	
1. Load	1	A	E						
2. Increment		I	A	E					
3. Branch to X			I	A	E				
4. Add				I	A	E			
5. Subtract					I	A	E		
6. Instr. in X							I	A	E

Load:  $R1 \leftarrow M[\text{address } 1]$   
 Load:  $R2 \leftarrow M[\text{address } 2]$   
 ADD:  $R3 \leftarrow R1 + R2$   
 Store:  $M[\text{address } 3] \leftarrow R3$

Three-segment pipeline timing

– Pipeline timing with data conflict

Clock cycle	1	2	3	4	5	6
Load R1	1	A	E			
Load R2		1	A	E		
Add R1 + R2			1	A	E	
Store R3				1	A	E

– Pipeline timing with delayed load

Clock cycle	1	2	3	4	5	6	7
Load R1	1	A	E				
Load R2		1	A	E			
NOP			1	A	E		
Add R1 + R2				1	A	E	
Store R3					1	A	E

The data dependency is taken care by the compiler rather than the hardware

Fig. 5.12 Delayed Load

In general, machine operation suitable for pipelining should have the following properties:

- Identical processes (or functions) are repeatedly invoked many times, each of which can be subdivided into subprocesses (or subfunctions).
- Successive operands are fed through the pipeline segments and require as few buffers and local controls as possible.
- Operations executed by distinct pipelines should be able to share expensive resources, such as memories and buses in the system.
- The operation code must be specified in order to select the functional unit or to reconfigure a multifunctional unit to perform the specified operation.

The most useful application of the pipelining concept is used in designing vector processors. A specially designed processor involves vector calculations.

### 5.2.5 Vector Processing

There are many computational problems, mainly mathematical and statistical applications, where the computational loads are too high and hence are beyond

## NOTES

**NOTES**

the capabilities of a conventional computer. These conventional computers may solve the problem but they may take few days or even weeks to complete the job. These complex computations are, therefore, performed using vector computations for faster processing. Some of the application areas that require vector processing are as follows:

- Weather Forecasting
- Artificial Intelligence
- Experts System
- Image Processing
- Seismology
- Gene Mapping
- Aerodynamics

These scientific problems require a computer known as Vector Processor with built-in instructions that perform multiple calculations on vectors (one-dimensional array) simultaneously. This vector processor is a CPU design where the instruction set includes operations that can perform mathematical operations on multiple data elements simultaneously. Vector processors are common in the scientific computing area.

A vector can be defined as an ordered set of one-dimensional array of data items. A vector V of length N is represented as  $V = \{V_1, V_2, V_3, \dots, V_N\}$ .

One of the most common computational operations performed in computers using vector processing is matrix multiplication. Suppose we have to perform  $C = A \times B$ , where all these three matrices A, B and C are one-dimensional vectors with n number of data elements. Then, the multiplication of matrices A and B can be shown as follows:

$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{pmatrix} \times \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} c_1 = a_1b_1 + a_1b_2 + \dots + a_1b_n \\ c_2 = a_2b_1 + a_2b_2 + \dots + a_2b_n \\ c_3 = a_3b_1 + a_3b_2 + \dots + a_3b_n \\ \vdots \\ c_n = a_nb_1 + a_nb_2 + \dots + a_nb_n \end{pmatrix}$$

where  $a_i$ ,  $b_i$  and  $c_i$  are the elements of vectors A, B and C, respectively.

This matrix multiplication can be computed using the following formula:

$$c_i = \sum a_i b_k, \text{ where } k= 1 \text{ to } n$$

Let the value of n be 3, then the matrix multiplication will be as follows:

$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \times \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} c_1 = a_1b_1 + a_1b_2 + a_1b_3 \\ c_2 = a_2b_1 + a_2b_2 + a_2b_3 \\ c_3 = a_3b_1 + a_3b_2 + a_3b_3 \end{pmatrix}$$

This matrix multiplication will require nine multiplications and six additions. Thus, in general, for a vector consisting of n elements, the total number of multiplications and additions needed to compute the matrix multiplication will be  $n^2$  multiplication and  $n(n - 1)$  additions.



## NOTES

For matrix multiplication computation, a pipeline vector processor as shown in Figure 5.13 consisting of multiplier pipeline and adder pipeline that can be used having three segments or stages. All registers attached to the segments are initialized to 0. The elements  $a_i$  and  $b_k$  are passed through the two pipelines. In the first clock cycle,  $a_1$  and  $b_1$  pairs are brought in the pipeline and multiplied in the multiplier pipeline. The pairs— $a_1$  and  $b_1$  are multiplied at the rate of one pair per cycle. Hence, after three clock cycles, the multiplier pipeline is filled with products  $a_1b_1$ ,  $a_1b_2$ , and  $a_1b_3$ . Thus, it takes three cycles to fill the multiplier pipeline and these products begin to be added to the next pipeline one by one. After the six clock cycles, the adder pipeline will get filled with the three product terms,  $a_1b_1$ ,  $a_1b_2$ , and  $a_1b_3$ , respectively and the next three products terms,  $a_1b_4$ ,  $a_1b_5$ , and  $a_1b_6$  are in the multiplier segments. Hence, it is only at the beginning of the seventh clock cycle the output of the adder pipeline will be  $a_1b_1$  and the output of the multiplier pipeline is  $a_1b_4$ . Thus, the seventh clock cycle starts the addition  $a_1b_1 + a_1b_4$  in the adder pipeline. The eighth cycle gives  $a_1b_2 + a_1b_5$  and so on.

During the tenth clock cycle, in the sum  $a_1b_1 + a_1b_4$ , the product  $a_1b_7$  will be added and in the next clock cycle, the product  $a_1b_8$  will be added in the sum of  $a_1b_2 + a_1b_5$  and so on till  $a_1b_n$ . Hence, the summation process will be as follows:

$$C = a_1b_1 + a_1b_4 + a_1b_7 + \dots + a_1b_2 + a_1b_5 + a_1b_8 + \dots + a_1b_3 + a_1b_6 + a_1b_9 + \dots$$

At last, when there are no more product terms to be added, zeros are inserted in the multiplier pipeline. The adder pipeline will consist of one partial product term in each of the three segments. The three partial sums are then added to compute the final sum.

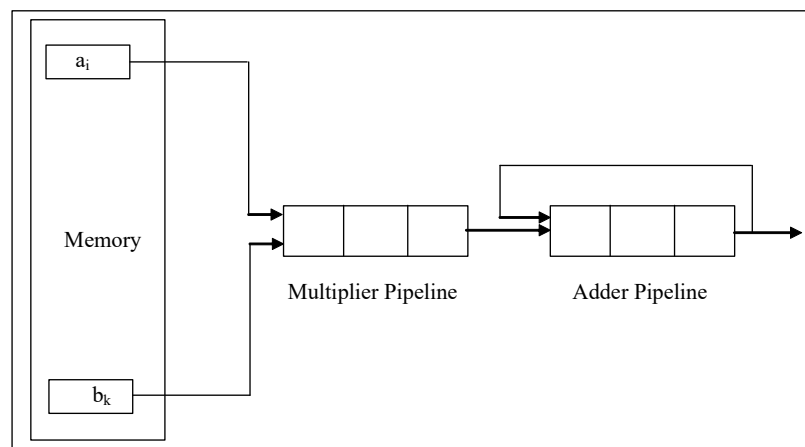


Fig. 5.13 Pipeline Vector Processor

### Vector Processing Requirements

A vector operand contains an ordered set of  $n$  elements, where  $n$  is called the length of the vector. Each element in a vector is a scalar quantity, which may be a floating point number, an integer, a logical value or a character.

A computer capable of vector processing eliminates the overhead associated with the time it takes to fetch and execute the instructions in the program loop and allow operations to be specified with a single vector instruction form as follows:

$$C(1:100) = A(1:100) + B(1:100)$$

The vector instruction includes the initial address of the operands, the length of the vectors and the operation performed all in one composition instruction. A possible instruction format for vector instruction is as follows:

Operation code	Base address source 1	Base address source 2	Base address destination	Vector length
----------------	-----------------------	-----------------------	--------------------------	---------------

## NOTES

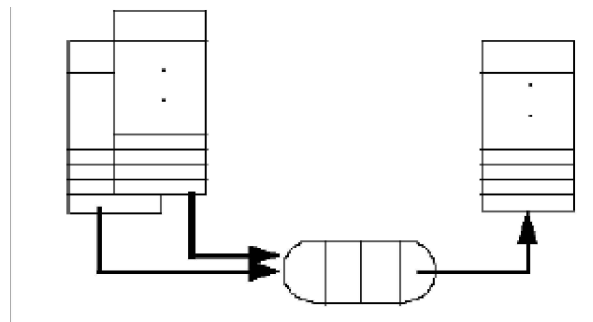
This is essentially a three address instruction, with three fields specifying the base address of the operand and an additional field that gives the length of data items in the vector.

- For a memory reference instruction, the **base addresses** are needed for both source operands and result vectors. If the operands and results are located in the vector register file, the designated vector registers must be specified.
- The **address increment** between the elements must be specified.
- The **address offset** relative to the base address should be specified. Using the base address and the offset, the relative effective address can be calculated.
- The **vector length** is needed to determine the termination of a vector instruction.

A vector processor consists of both a scalar processor unit and a vector processor unit, which could be thought of as an independent functional unit, capable of efficient vector operations.

### Vector Hardware

Vector processor unit is used to perform the vector operations efficiently. In these processors, the operands cannot be used directly from memory. Rather they are loaded into registers. Once the operation has been performed the results are put back into the registers rather than directly to memory. Vector hardware has the special ability to overlap or pipeline operand processing (Refer Figure 5.14).



*Fig. 5.14 Vector Hardware*

In this architecture, all vector functional units are pipelined such that it is fully segmented and each stage of the pipeline performs a computational step. Once the pipeline is full, you will get result at each step, provided no stall has been introduced.

The pipeline is divided into individual segments, such that, each segment is completely free to perform its computation and involves no hardware sharing. This improves the performance of the system as now it is possible that the machine can be working on separate operands at the same time. This ability enables it to produce one result per clock period as soon as the pipeline is full. The same instruction is carried out repeatedly using the pipeline technique, so the vector processor processes all the elements of a vector in exactly the same way. The

pipeline segments arithmetic operation, such as, floating point, multiplies into stages passing the output of one stage to the next stage as input. The next pair of operands may enter the pipeline after the first stage has processed the previous pair of operands. The processing of a number of operands may be carried out simultaneously.

The loading of a vector register is itself a pipelined operation, with the ability to load one element each clock period after some initial startup overhead.

Theoretical speedup depends on the number of segments in the pipeline. This is so that there is a direct relationship between the number of stages in the pipeline you can keep full and the performance of the code. The size of the pipeline can be increased by chaining. Thus the Cray combines more than one pipeline to increase its effective size. Chaining means that the result from a pipeline can be used as an operand in a second pipeline as illustrated in Figure 5.15.

## NOTES

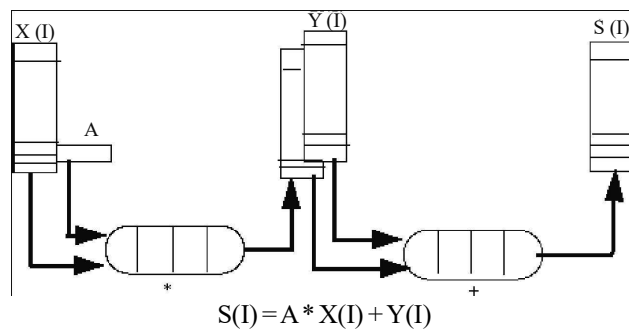


Fig. 5.15 Pipeline Chaining

This example shows how two pipelines can be chained together to form an effectively single pipeline containing more segments. The output from the first segment is fed directly into the second set of segments thus giving a resultant effective pipeline length of 8. Speedup (over scalar code) is dependent on the number of stages in the pipeline. Chaining increases the number of stages.

Most vector architectures have more than one pipeline; they may also contain different types of pipelines. Some vector architectures provide greater efficiency by allowing the output of one pipeline to be chained directly into another pipeline. This feature is called chaining and eliminates the need to store the result of the first pipeline before sending it into the second pipeline. Figure 5.16 demonstrates the use of chaining in the computation of a vector operation:

$$a * x + y,$$

where  $x$  and  $y$  are vectors and  $a$  is a scalar constant.

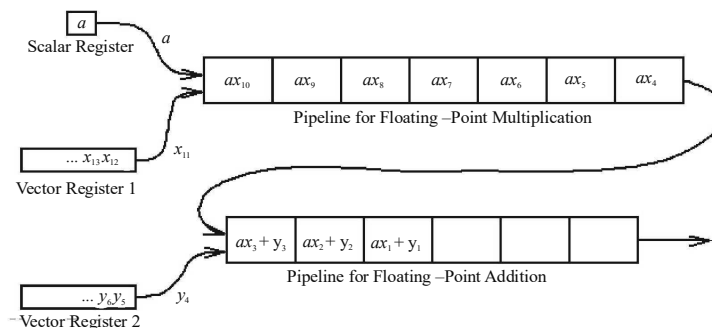


Fig. 5.16 Vector Chaining used to Compute  $a * x + y$

## NOTES

### Characteristics of Vector Processing

The characteristics of vector processing are as follows:

#### Vector Instructions

The ISA of a scalar processor is augmented with vector instructions of the following types:

##### *Vector-vector instructions*

f1:  $V_i \rightarrow V_j$  (e.g., MOVE Va, Vb)

f2:  $V_j \times V_k \rightarrow V_i$  (e.g., ADD Va, Vb, Vc)

##### *Vector-scalar instructions*

f3:  $s \times V_i \rightarrow V_j$  (e.g., ADD R1, Va, Vb)

##### *Vector-memory instructions*

f4:  $M \rightarrow V$  (e.g., Vector Load)

f5:  $V \rightarrow M$  (e.g., Vector Store)

##### *Vector reduction instructions*

f6:  $V \rightarrow s$  (e.g., ADD V, s)

f7:  $V_i \times V_j \rightarrow s$  (e.g., DOT Va, Vb, s)

#### Scatter and Gather Operations

Scatter and gather operations are used to process sparse matrices/vectors where only certain elements of a vector are needed in a computation. The gather operation, uses a base address and a set of indices to access from memory 'few' of the elements of a large vector into one of the vector registers. The scatter operation does the opposite of gather operation.

Most vector processors use gather operation for picking out the appropriate elements and putting them together into a vector or a vector register. If the elements to be used are in a regularly-spaced pattern, the spacing between the elements to be gathered is called the stride. For example, if the elements for every fifth element are the one that are used for the operation

$$a_1, a_6, a_{11}, a_{16}, \dots, a_{[5 * \text{floor}((n-1)/5) + 1]}$$

are to be extracted from the vector

$$(a_1, a_2, a_3, a_4, a_5, a_6, \dots, a_n)$$

then we will say that the stride is equal to 5.

A **scatter** operation reformats the output vector so that the elements are spaced correctly. Scatter and gather operations may also be used with irregularly spaced data.

f8:  $M \times V_a \rightarrow V_b$  (e.g., gather)

f9:  $V_a \times V_b \rightarrow M$  (e.g., scatter)

#### Masking Instructions

F10:  $V_a \times V_m \rightarrow V_b$  (e.g., MMOVE V1, V2, V3)

The masking operation allows conditional execution of an instruction based on a 'masking' register.

A Boolean vector can be generated as a result of comparing two vectors, and can be used as a masking vector for enabling and disabling component operations in a vector instruction. A compress instruction will shorten a vector under the control of a masking of vector. A merge instruction combines two vectors under the control of a masking vector.

## NOTES

### Pipelined Vector Processing Methods

Vector computations are often involved in processing large arrays of data. By ordering successive computations in the array, the vector array processing can be classified into three types which are as follows:

- **Horizontal Processing**—This is the one in which vector computations are performed horizontally from left to right in row fashion.
- **Vertical Processing**—This is the one in which vector computations are carried out vertically from top to bottom in column fashion.
- **Vector Looping**—This is the one in which segmented vector loop computations are performed from left to right and top to bottom in a combined horizontal and vertical method.

A simple vector summation computation illustrates these vector processing methods

Let  $\{a_i \text{ for } 1 \leq i \leq n\}$  be  $n$  scalar constants,  $X_j = (X_{1j}, X_{2j}, \dots, X_{mj})^T$  for  $j = 1, 2, 3, \dots, n$  be  $n$  column vectors and  $Y_j = (Y_{1j}, Y_{2j}, \dots, Y_{mj})^T$  be a column vector of  $m$  components. The computation to be performed is as follows:

- $Y = a_1 \cdot x_1 + a_2 \cdot x_2 + \dots + a_n \cdot x_n$
- $Y_1 = Z_{11} + Z_{12} + \dots + Z_{1n}$
- $Y_2 = Z_{21} + Z_{22} + \dots + Z_{2n}$
- $\vdots$
- $Y_m = Z_{m1} + Z_{m2} + \dots + Z_{mn}$

### Horizontal Vector Processing

In this method, all components of the vector  $y$  are calculated in sequential order,  $y_i$  for  $i = 1, 2, \dots, m$ . Each summation involving  $n-1$  additions must be completed before switching to the evaluation of the next summation.

### 5.2.6 Array Processing

An array processor is a specific processor type that performs the required computations on huge arrays of data. It can be created using a group of unique special processors which are specifically designed for calculating mathematical procedures at extremely high speeds, and are frequently under the control of another central processor. Some computers are designed for only using array processors that speed up video processing or can do fast floating-point mathematical operations. There are two different types of array processors, an attached array processor and a Single Instruction Multiple Data (SIMD) array processor.

Hence, an array processor is a processor that is designed for performing calculations on a large-sized array of data. The two types of array processors are discussed here.

## NOTES

### Attached Array Processor

An attached array processor is a peripheral device attached to a computer so that the performance of a computer can be improved for numerical computations. The purpose of the attached array processor is to improve the computer's performance by providing the functionality of vector processing for solving complex scientific problems. This can be achieved by means of a parallel processing technique with multiple functional units. An attached array processor consists of an attached Arithmetic and Logic Unit (ALU) that may contain one or more pipelined floating-point adders and multipliers.

The array processor is interfaced with the computer through an input/output interface. The computer treats the attached array processor as an external peripheral, which is a back-end machine driven by the computer. The main memory transfers data to a local memory through high-speed memory bus and the array processor receives the data from the local memory.

The objective of the attached array processor is to provide the conventional computers the functionality of the vector manipulations at a fraction of the cost of a supercomputer.

### Algorithms for Array Processors

Here we see a parallel algorithm to determine the maximum item in an array:

- For  $n$  array items, this algorithm uses  $n/2$  processors and takes  $Q(\log n)$  time
- Each processor takes 2 array elements from position  $M[\text{pid}]$  and  $M[\text{pid} + \text{incr}]$ , finds the max and copies it into  $M[\text{pid}]$

```
incr = 1;
while (incr < n)
    temp0 = M[pid]
    temp1 = M[pid + incr]
    if(temp0 > temp1)
        M[pid] = M[pid + incr]
    incr *= 2
```

It should be easy to see that the above algorithm iterates  $\log n$  times. Therefore, the complexity is  $4 \log n + 1$  number of operations or  $\log n$  comparisons.

### Linear Array of Processors

The algorithms are specifically used as unique functions on a two-dimensional mesh in the pattern-matching algorithm. An array of  $n$  processors is constructed from processors  $P_0 \dots P_{n-1}$ , where  $P_i$  is connected to  $P_{i-1}$  and  $P_{i+1}$ , if exists. It is considered that the text have been already allocated in the processors, such that the processor  $P_i$  stores the  $i$ th symbol of text  $S = T_0 \dots T_{n-1}$ . The prototype pattern  $P_0 \dots P_{m-1}$  is considered as an input to the array and then

first the symbols are entered into  $P_{n-1}$  and then transmitted through the array. The set order of the input symbols is referred as  $P_{m-1}$  through  $P_0$ . This specific algorithm is termed as Array Pattern Matching (APM) and is frequently used.

### SIMD Array Processors

An SIMD array processor has a single instruction multiple data stream organization that manipulates the common instruction by means of multiple functional units. This array processor consists of multiple ALUs that operate in parallel. ALUs work under the control of a common control unit performing the same operation and hence, achieve the SIMD stream organization.

The SIMD array processor consists of a set of processing elements where each Processing Element (PE) has its own local memory (M). The processing elements may include the ALU, floating-point arithmetic unit and registers. The main memory of the CPU is used for the storage of the program. The operation of the processing element is controlled by the master control unit, whose main function is to decode the instructions and determine how the instruction is to be executed. Data operands are transferred to local memories. Each of the processing elements operates upon the data stored in its local memory.

Suppose we need to perform the vector addition  $c_i = a_i + b_i$ , for  $i = 1, 2, 3 \dots n$ . The master control unit first stores the  $i^{\text{th}}$  data element in a local memory  $M_i$ . It then gives the add instruction  $c_i = a_i + b_i$  to the processing elements causing the addition to take place simultaneously. The result of  $c_i$  is stored in the local memories. Thus, the whole process is performed in one cycle.

### Array Instruction Set

Following table shows the instruction set of the processors in an array. The processors take their instructions from the Processing Element-Instruction Register (PE-IR), loaded by the ACU.

#### Arithmetic/Logical Instructions

Arithmetic/logical instructions are of the form:

< tr>

Opcodel	Immediate	---
Opcodem	Rx	Address
Opcode	Source	---

At the end of an operation, which updates the ACC, each active PE sends the value in its accumulator to its neighbours.

#### Opcodel

The operand is the immediate value in PE-IR.

#### Opcodem

The ACU reads the value in Rx from within its own registers, adds this to the value in the address field in AC-IR and loads this modified value into the address field of the instruction as it is copied into PE-IR. Each processing element accesses its own memory to obtain the operand using the modified address in PE-IR, i.e., it ignores the Rx field.

### NOTES

## NOTES

### Opcode

The Source can be:

E0, E1, W0, W1 or P in SIMD-1

N0, N1, E0, E1, W0, W1, S0, S1 or P in SIMD-2

<b>N</b>	operand = value sent by processor above
<b>E</b>	operand = value sent by processor to the right
<b>W</b>	operand = value sent by processor to the left
<b>S</b>	operand = value sent by processor below
<b>P</b>	operand = processor's own number within the array (0-7 in SIMD-1, 0-15 in SIMD-2)

### Major Characteristics of SIMD Architectures are:

- A single processor
- Synchronous array processors
- Data parallel architectures
- Hardware intensive architectures
- Interconnection network

An SIMD whose main component is an Associative Memory (AM) that is used in fast search operations consists of:

- Data register
- Mask register
- Word selector
- Result register

Associative processor architectures also belong to the SIMD classification: STRAN and Goodyear Aerospace's MPP (Massively Parallel Processor). The systolic architectures are a special type of synchronous array processor architecture. The SIMD instruction set contains additional instruction for IN operations, manipulating local and global registers, setting activity bits based on data conditions. Popular high-level languages such as FORTRAN, C and LISP have been complete to allow data-parallel programming on SIMDs.

### Check Your Progress

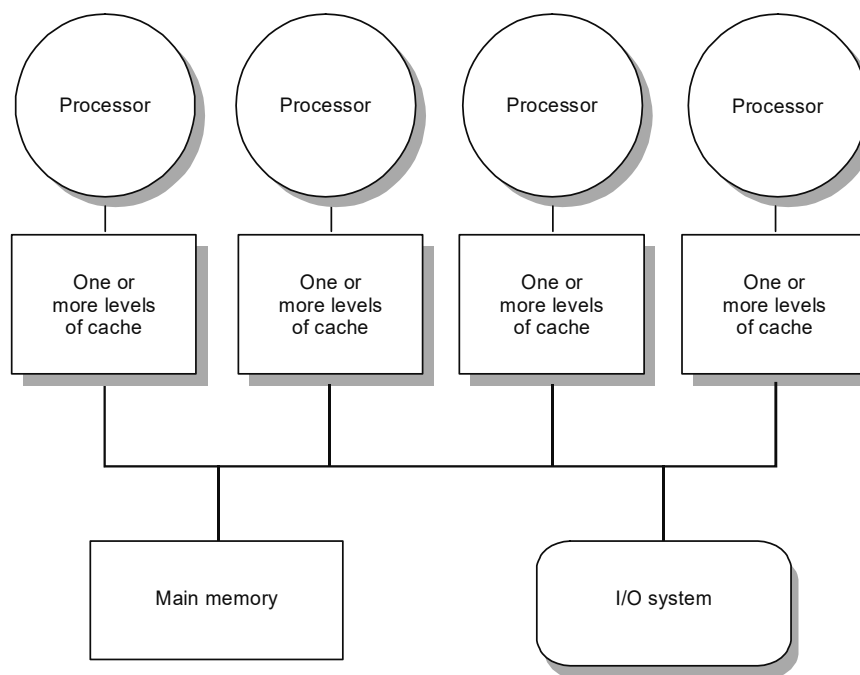
1. What is pipelining?
2. In which systems the arithmetic pipeline is used?
3. Write the steps involved in instruction pipeline.
4. Write strategies employed in resolving control dependencies due to branch instructions.
5. What are the areas of application of vector processing.



## 5.3 MULTIPROCESSORS

Multiprocessors use two or more than two CPUs assembled in single system unit. It refers the execution of various software processes concurrently. Cache coherency issue is solved by multiprocessor systems. The processors, for example Pentium Pro II, Power PC 603 and 604, digital Alpha keep cache in multiprocessors to solve the problem of cache coherence. These processors contain two, four and even eight processors within memory bus to share the memory problems. Large multiprocessors share a single bus to transmit the data directly from cache so that modified data aborts transaction to write back the system memory for next transaction, whereas original requester of data re-arbitrates data from the memory. The two types of multiprocessors are known as shared memory type multiprocessor and distributed memory type multiprocessor. The main memory is directly accessed by the assembled processors in multiprocessor system is known as shared memory multiprocessor in which the shared portion belongs actually to main memory referred to as global memory. Cache is used for this and known as high-speed buffer exist with almost each processor. In fact, data and instructions are accessed from local memory and global memory that is used by internetworking facility. Distributed memory type has own private memory. Various processors involved in shared memory type access the same variables at a time that can be referenced for data integrity. But in distributed memory a computational task is distributed for multiple processors involving distinct memory stack to reassemble the produced result (Refer Figure 5.17).

### NOTES



*Fig. 5.17 Memory Organization in Multiprocessor*

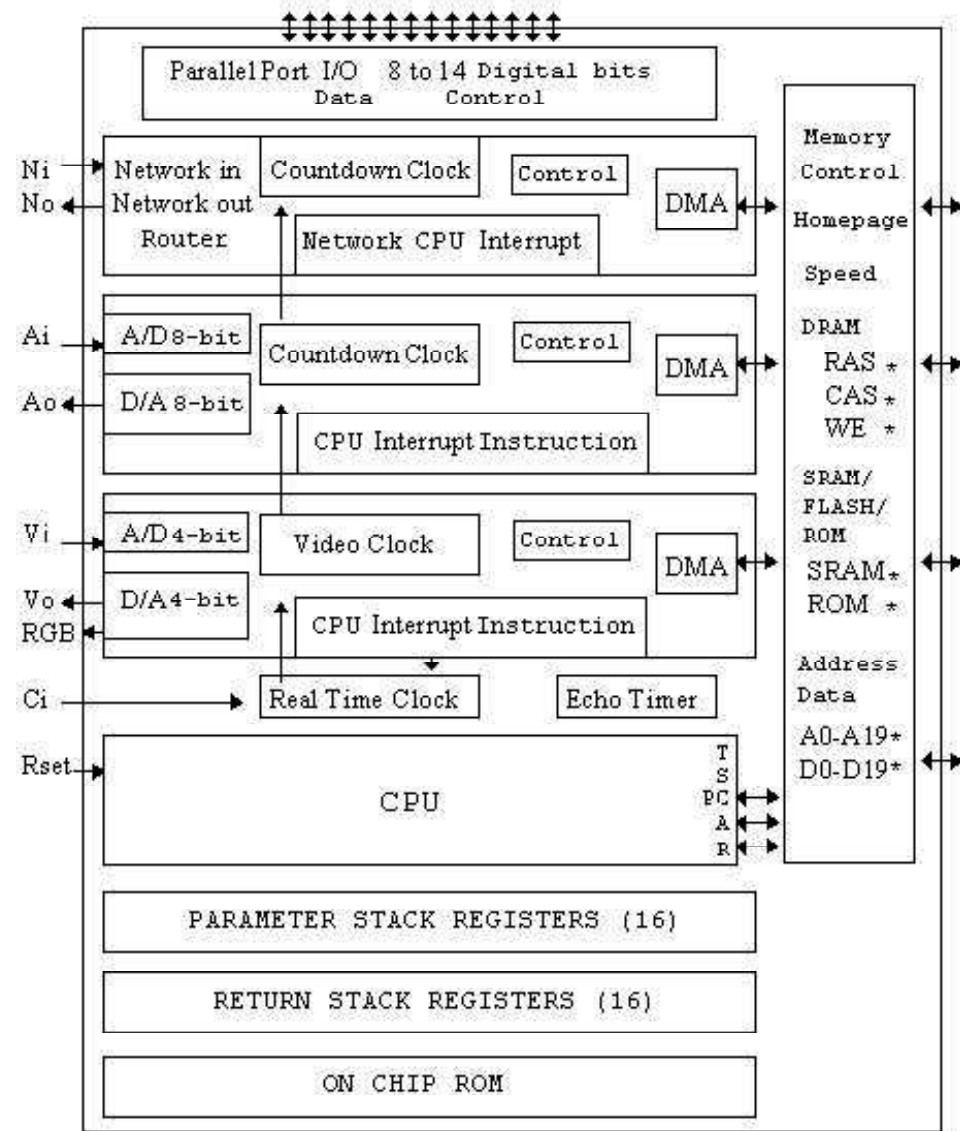
Multiprocessor uses large caches but limited processors that shares single memory bus.

**NOTES**

Both memory types (distributed and shared) are also referred to as tightly-coupled and loosely-coupled multiprocessors respectively and hence called coupled multiprocessors. Loosely coupled system uses file level and tightly-coupled works with data elements. Tightly coupled Multiprocessors use memory that is physically shared so that part or all of the memory available to a processor is also available to other processors. Basically, they do not have central processors instead they are having multiple processors. There is difference between shared and distributed memory type architecture is how each processor is assembled. The examples of OS support multiprocessors are UNIX, Linux, Windows 2000 etc. Figure 5.18 shows a functional diagram a typical multiprocessor.

**5.3.1 Interconnection Structure**

Following are the interconnection structure used in the multiprocessing.

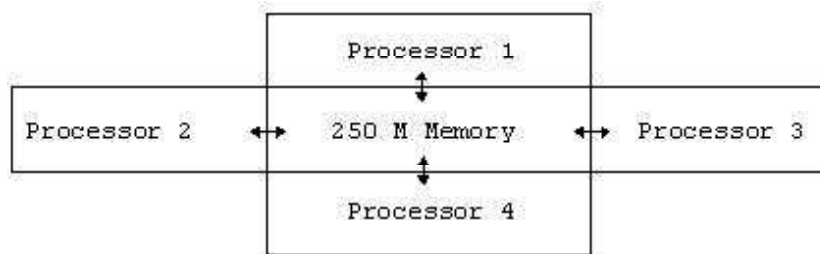


*Fig. 5.18 Functional Diagram of a typical Multiprocessor*

## Shared Memory Multiprocessors

Shared memory multiprocessor consists of a number of processors can be accessed among various shared memory modules. The assembled processors are connected physically to the various memory modules but each processor is logically connected to every memory modules (Refer Figure 5.19). The arrangement of memory modules is as follows:

- The system unit contains **m** memory modules and **p** processors.
- Each processor sends requests so that they can get information from memory module.
- The memory modules are synchronized that means two modules can get requests at the same time because memory access performs task in one time unit, i.e. one complete memory cycle.
- Memory module can work one request at a time. If more than one request is tagged with at a time, they are to be queued when memory cycle starts.



*Fig. 5.19 Four Processors Involved in Shared Memory Multiprocessors*

Let take an example in which shared memory is worked with four processors and sends requests to four modules. The following table defines the various states which are involved in the processing:

*Table 5.4 Various States and Their Functions*

State	Function
1	This state keeps all four requests in one module
2a	This state keeps three requests in one module and one in other module.
2b	This state keeps two requests in one module and two are in other module.
3	This state keeps two requests in one module and one request is kept each of the two modules.
4	This state keeps each request in all of the four modules

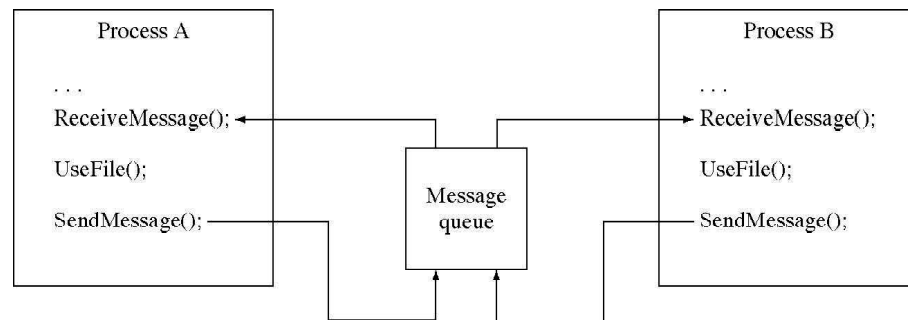
## Distributed Memory Multiprocessor

The distributed memory multiprocessor keeps a number of processors in which virtual storage space is assigned for redundant execution. Each processor is assembled to write the data. The memory is allotted with each processor which keeps the address of its own memory and full bandwidth of local memory without involving in interference the other processor. There is no limit for number of processors. The memory capacity in system is constrained because the connecting

## NOTES

## NOTES

processors are used in network. The distributed memory multiprocessors do not encounter with cache coherency problems. Each processor is worked with its own data. It keeps the data in local cache because data can be referenced by other processors. One main drawback of this type of multiprocessor is that interprocess communication task becomes very complex because if one processor needs other processor memory the message must be exchanged with each other which introduces two types of overhead. First overhead takes time to prepare and construct the message form one processor to other and second overhead refers to receiving processor that is in fact interrupted if one message is shared from one processor to other processors. Concurrent programmings, semaphore monitoring, etc. are the various techniques which are not directly applicable because they are implemented by a layered software approach. For example, if message is passed form one processor to other, it uses semaphore programming. The need of mutual exclusion in the operating system is to implement the semaphores but few machine instructions are busy to accept it. The process blocking is required in operating system calls at this design phase. The name servers are expensive to broadcast in a network if name resolves to this but at this level, name servers and global names use hierarchy that are basically local unique names. The solution for mutual exclusion is simple that generalizes to  $N$  number of processes. These processes follow the protocol to call the message passing system calls to follow the IPC patterns.



*Fig. 5.20 Mutual Exclusion Process*

Figure 5.20 shows how two-process mutual exclusion can take place in the operating system. Here Process A and Process B maintain a message queue through three methods namely, `ReceiveMessage()` ; , `UseFile()` ; and `SendMessage()` ; . The processes are decided after getting the received messages. The received messages are collected in the process that can be used in method `UseFile` and at last message is sent to the clients. This concept can be understood with the help of examination. The messages are solved at the user level but the mutual exclusion is needed to be implemented at the operating system level. So, this step is considered as reduce a problem to a special case. For example, all the commands and events that are used at command prompt are not easy to remember but this problem can be solved if use **Help** menu or **Help** command of the running application.

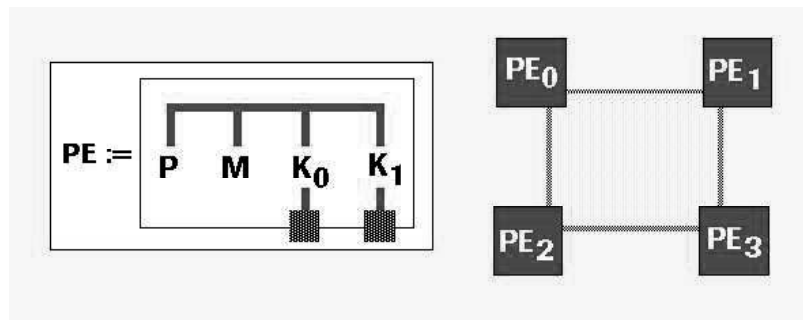


Fig. 5.21 Distributed Memory Processor

Figure 5.21, PE refers to processing element that explains how one message is passed from one to other by copying information in the local memory. Then it tells the local controller form transferring the information to the external devices. The controller also finds a stack for incoming message then it notifies the special processor for arrived message.

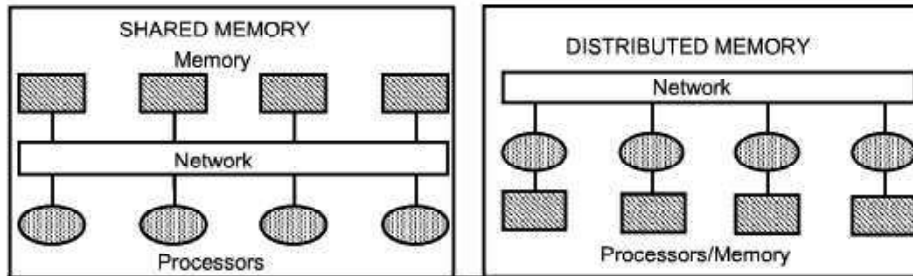


Fig. 5.22 Comparison of Shared and Distributed Memory Multiprocessors

Figure 5.22 displays Comparison of Shared and Distributed memory Multiprocessor.

### 5.3.2 Characteristics of Multiprocessors

Table 5.5 shows the characteristics of multiprocessors.

Table 5.5 Characteristics of Multiprocessors

Criteria	Characteristics
<b>Fast speed</b>	Multiprocessor supports fast processing to perform the task. Speed is decided in multiprocessor by a large number of components, for example PSK1, SSK, PSK etc. These systems components are connected with every serial system bus maintain the consistency of cache for high speed bus (SB1, SBn-1, SBn) etc.
<b>Memory organization</b>	Its memory organization is centralized in which bus uses large cache and effective way of scaling the memory bandwidth. It uses shared and distributed memory organization using at least 2-8 processors.
<b>Efficient system-on-chip</b>	Multiprocessor includes embedded processors, mixed signal circuits, digital logic techniques to make an efficient system-on-chip because all together are combined to make a heterogeneous multiprocessor.

## NOTES

## NOTES

<b>Thread-level parallelism</b>	It executes using reentrant programs multiple processes known as multithreading. These programs are necessary to run the thread. Basically it refers to two threads that execute the same code simultaneously. The programs are not thread-safe if programs with embedded data are not reentrant. The programs can be made as reentrant by removing the embedded data and allocated it for each execution in the multiprocessors. The most versions of MS/DOS are not reentrant but in the object-oriented programs, the data area of each object is allocated heap store therefore, they are automatically thread-safe means reentrant. Therefore, multiprocessors maintain thread-level parallelism that makes them
<b>Support fast service of interconnection network</b>	It requires interconnection network because it communicates among processors across net.
<b>Pipelining and scheduling</b>	Multiprocessor uses pipelining and scheduling to make architecture as parallel architecture.
<b>On-time multiprocessing</b>	It uses various multiprocessing, such as SISD (single instruction stream, single data stream), SIMD (single instruction stream, multiple data stream), MISD (multiple instruction stream, single data stream), and MIMD (multiple instruction stream, multiple data stream). SISD refers to uniprocessor, SIMD refers to multimedia processors, MISD refers to special purpose stream processors, and for example digital filter and MIMD refers to servers. The main characteristics of these multiprocessing is cache-coherent that allows the running software on any processor share with the other memory and system resources at minimal support.
<b>Different configuration</b>	Multiprocessors use bus in which CPU is configured as master/client arrangement in which environments are set with isolated engines to exchange the processing information over LAN or WAN.

### 5.3.3 Interprocess Arbitration

An interprocess arbitration system for multiprocessors shares a common bus. Arbitration fixes the priority criteria made by the multiprocessors. This operate a multiprocessor system in either synchronous or an asynchronous manner. In input-output organization, multiprocessor operating system controls all the processors by same clock. In this, common bus is prime factor because various multiprocessors must function in a synchronous manner. The major problem involved in interprocessing of multiprocessors with a common clock during pulse path run by circuits and buses. The availability of system is closely linked to the common clock cycle. In asynchronous system, a processor is added to existing system without modifying the algorithm because it controls the interprocess arbitration of access request by various processors having common bus. The two types of interprocess arbitration are serious arbitration system and parallel arbitration system. In series arbitration system, each processor is comprised with arbitrate circuit which is arranged to connect in line. These systems use clock. The clock manages the request of different processors by a common bus. If one processor fails, no circuit scrutinizes the interprocess arbitration to switch from one system to other. Arbitration is used to decentralize the decision to avail greater flexibility to the system that makes processors or microprocessors in a very short time (approximately 100 nanoseconds). To make a decision in probable priority combinations, such as fixed, cyclic or mixed, a fusible PROM memory is used. The combination of priorities is obtained by coding and scrambling circuits

which make the system producing in the form of integrated circuit. In short, an interprocess arbitration system connects a large number of processors to a common bus accessing all the resources used in file processing. It involves control lines that maintain the state of system, for example, occupied or current. The interprocess arbitration basically refers to a process in which the mechanism is communicated to exchange the data from one process with other processes without negotiating a request made by FIFO for storing ones and zeros that corresponds to received requests in the order that they are made. This process is communicated with **os.popen** (function in UNIX) and subprocess. Each arbiter independently reviews and processes the messages so that the computers communicate directly with each other on a peer to peer basis without the need for a master controlling program or other gateway for controlling and processing the messages as the messages are transmitted between computers. The interprocessing arbitration uses pipes, named pipes (include FIFO mechanism in which data is written to a pipe that is first named), message queuing and passing (messages are coordinated and managed with system kernel for application program interface), semaphores (programmed value is used to synchronize the information), shared memory (allows data to be exchanged on share basis), sockets (interprocessing is communicated with arbitration that requires third party over a network between client and server) and mutual exclusion (it wastes the processor time but primitive interprocess priors it).

Arbiters play main role in interprocess arbitration. The main application of interprocess arbitration is synchronization problem. Synchronization requires arbitration. The wait and signal instruction produced by register do not need arbiter. Arbiter is a type of device used to make binary decision based on instruction. For example, for A–B operation, an arbiter implies the decision that A begins before B ends. It also decodes a discrete decision which is basically based on a range of values. It produces a result either 0 or 1. It is designed by hardware designers. A bus arbitration circuit is used to determine which board connected to the system bus gets control of the bus for data transfers. One of the most common methods of arbitrating the bus is to assign a priority level to each board and to award bus ownership to the board with the highest priority request. This normally requires a combination of priority encoder and decoder logic to determine the priority.

A bus arbitration circuit is need for finding which part related to system bus gets the control of bus for data transfers. Most general way of arbitrating bus is giving a important strand of ever board and giving bus ownership to board with highest priority request. It generally needs a mixture of precedence encoder and decoder logic for finding priority. It uses wired-OR logic on the bus, implemented with open-drain outputs for determining the relative priority and eliminating the use of dedicated priority encoder/decoder logic.

A centralized arbitration is attached to the group of processors, whereas decentralized system requires less number of processors assembled with less number of circuits. Generally a choice is moved to interprocess arbitration to transmit on the bus signals whose frequency is considered as above 10MHz. The main function of this system is to fix a priority encoder that is preceded by circular network to access the network followed by an adder that adds number of

## NOTES

## NOTES

processor and the **value+1** in which the input is fixed priority encoder that is connected to the output of logic circuit. It makes logic circuits to the fixed priority and cyclic priority encoders. The request and the interprocess arbitration operate at fixed priority and requests of the arbitration correspond to the number of processors requesting access to the bus which is constructed in the form of interprocess.

A typical bus arbitration cycle begins by detection of an active BREQ signal driven by all requesting boards. If BREQ is inactive, it means that none of the boards on the bus has requested the bus. Table 5.6 lists the signals used in interprocess arbitration along with their functions.

**Table 5.6** Signals Used in Interprocess Arbitration with Their Functions

Signals	Functions
BAP	This signal tells that an arbitration of the access bus is possible during interprocessing.
DBA	This signal bus request.
BREQ	This signal on the bus indicates that a request from process arbitration is to be processed.
BECH	This signal is exchanged information by bus.
BNA	This signal on bus applies +1 to the priority of resolution circuits of the arbitration designate a new arbitration.
BM1 to BM3	This signal creates 3 lines of bus in which signals from the encoded number of processors.
BM4	This signal requests the validation signal make active if its logic level is 0 and validate signals from BM1 to BM3.
BAL	This signal represents synchronization signal decided by interprocess arbitration with a certain delay or signal DBA.

**Note:** Word overline for example, signal maintains the low bit or bytes in the process of message passing.

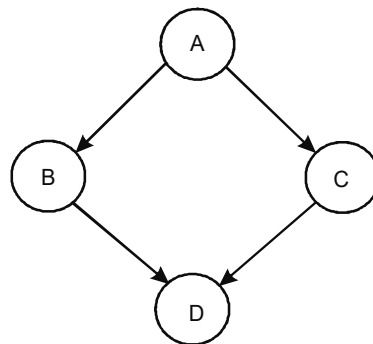
The conditions require for interprocessing arbitration are as follows:

**Mutual Exclusion:** If only one process holds a resource at a given time.

**Hold-And-Wait:** If one process holds the allocated resources and other waits for it.

**No Preemption:** If resource is not removed from a process holding.

**Synchronization Problem:** If busy waiting, programmer errors, deadlock or circular wait occurs in interprocessing.



**Fig. 5.23** Synchronization Problem for Four Actions



In Figure 5.23, process a send requests to B and C both. The process operation is based on priority basis. Process B and C send their request directly to C at a time. So this type of problem is determined by arbitration in interprocessing.

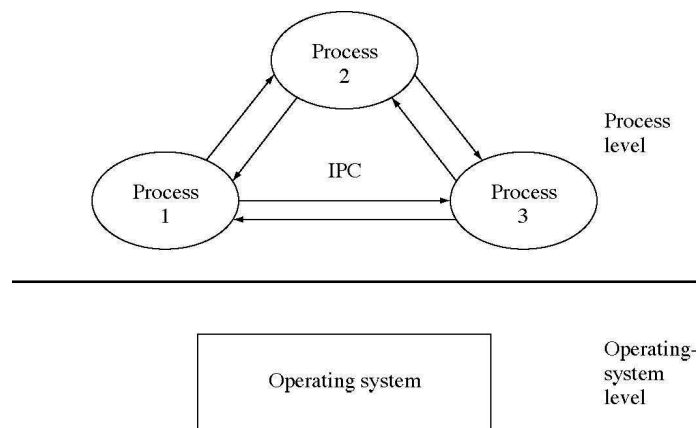
If any problem occurs at interprocessing arbitration, mutual exclusion is implemented with semaphores. The need of mutual exclusion in the operating system is to implement the semaphores but few machine instructions are busy to accept it. The process blocking is required in operating system calls at this design phase. The name servers are expensive to broadcast in a network if name resolves to this but at this level, name servers and global names use hierarchy that are basically local unique names. The solution for mutual exclusion is simple that generalizes to  $N$  number of processes. These processes follow the protocol to call the message passing system calls to follow the IPC patterns. For this the following statement is written in 'C' required:

```
int Attach_Msg_Queue(char *Queue_Name)
```

In the above statement, the function `Attach_Msg_Queue` returns `qid` as character type value that points to the queue identifier. The second statement is as follows:

```
int Detach_Msg_Queue(int Queue_id)
```

In the above statement, the function `Detach_Msg_Queue` returns `Queue_id` as integer data type value that points to the queue identifier.



**Fig. 5.24** IPC at User Process Level

In Figure 5.24, it is shown that the three processes are controlled in OS through IPC. The OS is set at OS level, whereas all the processes are set at process level. A general problem defined in OS can be taken as more than once, whereas a solution is given as to work well with the system. For example, interprocess communication patterns provide a typical way to use arbitration design pattern that arranges the objects to solve common problems and in frameworks skeleton code is used to solve common problems. The all patterns are set in operating system in that way that processes do not fail at critical times but sometimes processes do fail in networks. The solutions are hard but the portability is reduced if a single process failure that can cause the entire system to fail. To overcome this problem, fault-tolerant server system is used. This mechanism can be referred to as adding a new facility to the system hence known as arbitration.

## NOTES

---

## 5.4 INTERPROCESSOR COMMUNICATION AND SYNCHRONIZATION

---

### NOTES

Process synchronization is a mechanism used by the OS to ensure a systematic sharing of resources amongst concurrent resources. When the correctness of the computation performed by cooperating processes can be affected by the relative timing of the processes execution, it is referred to as a race condition.

Several processes access and manipulate the shared data concurrently. The final value of the shared data depends upon which process finishes last. In order to prevent race conditions, it is essential that concurrent processes are synchronized.

### 5.4.1 Racing Problem

Let us consider two processes  $P_0$  and  $P_1$  which are accessing the same integer value A.

Let  $A=1000$

$P_0$  : Read (A);

$A := A - 100$ ;

    Write (A);

$P_1$  : Read (A);

$A := A + 200$ ;

    Write (A);

The execution of these two processes will result in the change of value of A as  $A=1100$ . This will only happen if  $P_0$  and  $P_1$  will execute in any one of the following sequences:

(a)  $P_0$  followed by  $P_1$

(b)  $P_1$  followed by  $P_0$

But, if  $P_0$  and  $P_1$  are allowed to execute in an arbitrary fashion, then there would be the following two possibilities:

#### Possibility 1:

$P_0$ :

Read (A);

$A=A-100$ ;

Write (A);

$P_1$ :

Read (A);

$A=A+200$ ;

Write (A);

The sequence followed is:  $P_0, P_1, P_0, P_1$  the end result value of A is 1200, which is wrong.

**Possibility 2:**

$P_0$ :	$P_1$ :
Read (A);	
	Read (A);
$A = A + 200$ ;	
Write (A);	
	$A = A - 100$ ;
	Write (A);

The sequence followed is  $P_0, P_1, P_0$ , which leads to  $A = 900$ . This is again wrong. Such a situation is called a racing problem, which should be avoided.

**Critical Section**

It is basically a sequence of instructions with a clear indication of beginning and end for updating shared variables. In critical section, only one process is allowed to access the shared variables and all others have to wait. This condition is known as mutual exclusion. Mutual exclusion ensures that at most one process at a time has access to it during critical updates, i.e. it helps the system to maintain its integrity by allowing only one process to share the common resources.

**5.4.2 Problems of Critical Section**

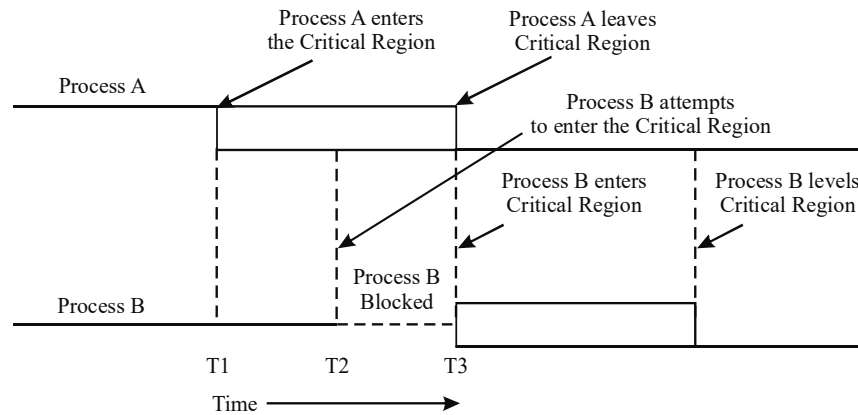
Any mechanism to control access to critical sections must satisfy the following three requirements, so that its can be ensured:

- (a) **Mutual Exclusion:** When one process is in a critical section that accesses a set of shared resources, no other processes can be in a critical section accessing any of those shared resources.
- (b) **Progress:** When no processes are in its critical section, and one or more processes are waiting to enter the critical section. A process that terminates outside its critical section may not prevent other processes from entering their critical sections.
- (c) **Bounded Wait:** When a process requests access to a critical section, a decision that grants it access may not be delayed indefinitely.

A process may not be denied access due to starvation or deadlock. Figure 5.25 shows Mechanism to Control Access to Critical Section

**NOTES**

## NOTES



*Fig. 5.25 Mechanism to Control Access to Critical Section*

Initial attempts to solve the problem of critical section were to have only two processes  $P_0$  and  $P_1$  with the general structure as:

```
do {
  Entry Section
  Critical Section
  Exit Section
  Remainder Section
} While (1);
```

Processes may share some common variables to synchronize their actions.

**Entry Section:** This refers to the code segment of a process that is executed when the process intends to enter its critical section. This code appears at the entry of a critical section. The execution of this code checks whether or not the process has the eligibility to enter the critical section.

**Critical Section:** This refers to the code segment where a shared resource is accessed by the process. At a time, only one of the cooperating process can enter into the critical section. Hence the shared resource is being used by only one process.

**Remainder Section:** This is the remaining part of a process's code. When a process is executing in this section, it implies that it is not waiting to enter its critical section. Hence, the processes in the remainder section are not considered to enter into the critical sections.

### 5.4.3 Critical Section Algorithms

#### Algorithm 1

This algorithm is applicable for only two processes  $P_0$  and  $P_1$ . Both are cooperating through a shared integer variable 'turn'. At a time its value will be either 0 or 1. If turn equals 0,  $P_0$  enters the critical section, and if the value is 1, then  $P_1$  enters the critical section.

```
P0: do {
  While (turn == 1); /* keep looping as long as turn
  equals 1*/
```

```

        /* this is entry section*/
    < Critical section >
Turn = 1; /* P1 enters into the critical section*/
        /* this is Exit Section*/

    < Remainder Section >

} While (1);
P1: do {
        While (turn == 0); /* keep looping as long as turn
equals 0 */
        /* this is the Entry Section */
    < Critical Section >
        Turn = 0; /* P0 enters into the critical
section*/
        /* this is Exit Section*/
        < Remainder Section >
    } While (1);

```

## NOTES

Let us observe the three conditions for controlling access to critical section:

- (a) **Mutual Exclusion:** The value of turn is either 0 or 1. Hence, at a time only one cooperating process will be there in the critical section. Thus, the requirement of mutual exclusion is satisfied.
- (b) **Progress:** Here the two cooperating processes are entering strictly alternatively only. In other words, if the value of turn is 0, then P<sub>0</sub> enters, and if the value of turn is 1, P<sub>1</sub> enters the critical region. Hence, a sequence followed is: P<sub>0</sub>, P<sub>1</sub>, P<sub>0</sub>, P<sub>1</sub>, P<sub>0</sub>, P<sub>1</sub>... and if the initial value of turn is 1, then sequence followed is: P<sub>1</sub>, P<sub>0</sub>, P<sub>1</sub>, P<sub>0</sub>, P<sub>1</sub>, P<sub>0</sub>.....

Consider the following two situations:

1. Turn = 0 but P<sub>1</sub> tries to enter the critical section earlier than P<sub>0</sub>; it cannot, until P<sub>0</sub> enters the critical region, executes and exits the critical region and changes the value of turn to 1.
  2. When P<sub>0</sub> exits the critical region, it changes the value of turn to 1. Suppose at this point, P<sub>1</sub> is executing in its remainder section and does not want to enter into the critical region again, but since the value of turn is set to 1 then no process is executing in the critical region at this time. But still P<sub>0</sub> cannot enter the critical region until and unless P<sub>1</sub> enters the critical region and it sets the value of turn equal to 0. Thus, this is the situation that no cooperating processes are in the critical-section but still P<sub>0</sub> has to wait. So, *the requirement of progress is not satisfied.*
- (c) **Bounded Wait:** Suppose a process P<sub>0</sub> requests to enter the critical section. There are two possibilities:
- (a) If turn = 0, then P<sub>0</sub> immediately enters the critical section.

- (b) If  $\text{turn} = 1$ , then  $P_1$  enters the critical section. In this way,  $P_0$  has to wait until  $P_1$  finishes executing in the critical section. Once exited, the value of  $\text{turn}$  is changed to 0 and  $P_0$  can enter into critical section. Hence, the requirement of bounded wait is met.

## NOTES

### Algorithm 2

This algorithm also works for only two processes. This takes into consideration as to who wants to enter the critical section. Boolean variables  $\text{Flag}[0]$ , and  $\text{Flag}[1]$  are used to synchronize the two cooperating processes. Initially, they are set to false, and are accessible to both the cooperating processes.

Whenever a process  $P_0$  intends to enter its critical section, it sets its  $\text{Flag}[0]$  which is accessible to process  $P[1]$  also. Now, the process  $P[0]$  will examine the  $\text{Flag}[1]$  of the other cooperating process  $P[1]$ . If  $\text{Flag}[1]$  is not set, then  $P[0]$  enters its critical section immediately. Else, if  $\text{Flag}[1]$  is also found to be set, it indicates that  $P[1]$  is also interested to enter its critical section. In that case,

$P[1]$  has to wait till  $\text{Flag}[1]$  is cleared by  $P[1]$ .

```
typedef enum Boolean {false, true};
Boolean flag [2]; /* initially both set to false*/
P0:
do {
flag [0] = true; /* intend to enter critical section*/
while (flag [1]); /* keep looping as long as flag is
true*/
/*Entry Section*/
< Critical Section >
Flag[0] = false; /* Exiting from the critical section*/
< Remainder Section >
} while (1);
P1:
do { flag [1] = true;
while (flag[0]) /* keep looping as long as flag is true*/
/* Entry Section */
< Critical Section >
Flag[1] = false; /* Exiting from the critical section*/
< Remainder >
} while (1);
```

Let us observe the three conditions for controlling access to critical section:

- (a) **Mutual Exclusion:** The value of  $\text{Flag}$  is either 0 or 1, i.e. either  $\text{Flag}[0]$  is set or  $\text{Flag}[1]$  is set. Hence, at a time only one cooperating process will be there in the critical section. Thus, *the requirement of mutual exclusion is satisfied.*
- (b) **Progress:** Let us observe when process  $P_0$  is running and it wants to enter in its critical section. It executes the statement  $\text{flag}[0] = \text{true}$ , and is preempted thereafter prior to execution of “while” statement. Now, if  $P[1]$  is at RUN state.  $P[1]$  also intends to enter the critical section, and sets flag

[1] to true. Now both the processes flag are set to true but none is in the critical section hence **deadlock** as both will continue looping in the while statement each waiting for the other to reset its flag. This is the situation when none is in the critical section, both waiting to enter, but none is able to enter. So, *the requirement of progress is not met.*

- (c) **Bounded Wait:** When process  $P_0$  sets its flag [0] to true. it checks for flag [1].

There are two possibilities:

- (a) Flag [1] is 'false'.  $P_0$  enters its critical section immediately.  
 (b) Flag [1] is 'true'. Process  $P_1$  is executing in its critical section.

Now,  $P_0$  waits for the moment when  $P_1$  exits its critical section, resets its flag [1] to false, thus enabling  $P_0$  to enter its critical section. Thus, after  $P_0$  indicates its intention to enter its critical section, and before it enters,  $P_1$  can enter critical section at most once. Hence, *the requirement of bounded wait is met.*

Unfortunately, a distinct possibility exists that both processes may set their Flags to 'true' almost simultaneously, thus preventing each other from entering their respective critical section (a deadlock).

An algorithm is required for breaking the tie in case of a deadlock. Peterson's algorithm serves this purpose. It has a synchronism mechanism to take into consideration both 'who intends to enter the critical section' and 'who should enter the critical section'.

### Algorithm 3 (Peterson's Algorithm)

```

    Int turn: /* initial value does not matter*/
    Boolean flag [2]; /* initially set to false*/

P0:
    do { flag[0] = true; /*intend to enter critical section*/
        turn = 1;
        while ( flag[1] && turn= =1); /* keep looping as long
as flag[1] is set to true && turn equals 1*/
        /* Entry Section*/
        < Critical Section >
        Flag[0] = false; /* exiting from critical section*/
        < Remainder Section >
    } While (1);

P1:
    do { flag[1] = true; /*intend to enter critical section*/
        turn = 0;
        while ( flag[0] && turn= =0); /* keep looping as long
as flag[0] is set to true && turn equals 0*/
        /* Entry Section*/
        < Critical Section >
        Flag[1] = false;          /* exiting from critical
section*/
    
```

## NOTES

```
< Remainder Section >  
    } While (1);
```

## NOTES

This algorithm meets all the three requirements of an ideal critical-section solution. Here, if both processes set their flag to true, then the value of 'turn' will break the tie since it can assume only one value, either 0 or 1.

If both flags are set to true and turn equals 0, then  $P_0$  is allowed to enter the critical section. If the value of 'turn' equals 1, then  $P_1$  proceeds.

### 5.4.4 Hardware Support for Mutual Exclusion

#### Test-and-Set (TS) Instruction

TS instruction provides a direct hardware support to mutual exclusion. It allows only one concurrent process to enter the critical section. It is able to make the programming task easier. It also improves the efficiency of the system.

The critical-section problem can be solved simply in a uniprocessor environment if we are able to prevent the occurrence of interrupts during the modification of a shared variable. Hence, the current sequence can be allowed to be executed without pre-emption. This is the approach taken by non-preemptive kernels.

Unfortunately, this solution is not feasible in a multiprocessor environment. Message passing to all the processors could be time-consuming in case of disabling the interrupts in a multiprocessor environment. As entry into the critical section is delayed due to such a message passing, the efficiency of the system decreases.

Therefore, many modern systems provide a special hardware support which helps in testing and modifying the content of a word or in swapping the contents of two words atomically.

This mechanism sets the global variable to 0, which indicates that the shared resource is available for being accessed. Each process has to execute TS instruction in order to use the resource available with a control variable as an operand. As a principle, TS takes one operand, the address of the control variable or a register, which may act as a semaphore.

You can implement the wait operation on the semaphore variable S using TS instruction set. However, it is possible only in case of the availability of the set with the supporting hardware.

#### Test-and-Set instruction

```
boolean TestAndSet (int S) {  
    if (S == 0) {  
        S = 1;  
        return true;  
    }  
    else {  
        return false;  
    }  
}
```



The importance of the above instruction is that it executes automatically. Thus, if two TestAndSet () instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.

### 5.4.5 Swap Instruction

```
void swap (int register, int memory)
{
    int temp;
    temp = memory;
    memory = register;
    register = temp;
}
```

Like the TestAndSet () instruction, the swap instruction is also executed atomically. Otherwise, mutual exclusion will not be ensured. This algorithm also ensures the requirement of progress, but it does not satisfy the requirement of bounded-wait. A process waiting to enter its critical section may have to wait for unduly long time or may have to wait forever. While a process is waiting, other cooperating processes may keep entering their critical sections repeatedly. So, the algorithm may cause *starvation* of waiting processes.

#### Advantages

The advantages of the TS instruction are as follows:

- It is applicable to any number of processes on either a single processor or multiple processors sharing main memory.
- It is simple and, hence, easily verifiable.
- It can be used to support multiple critical sections.

#### Disadvantages

The disadvantages of the TS instruction are as follows:

- Busy-waiting consumes a lot of the processor's time.
- Starvation may occur if a process leaves a critical section and there are more than one process in waiting.
- It results in deadlocks.

If a low-priority process has the critical region and a high-priority process needs the higher priority process will obtain the processor to wait for the critical region

#### Wait and Signal Operations

Wait and Signal is a modified version of the TS instruction which is designed to remove busy-waiting. Wait and Signal are two new and mutually exclusive operations. They become a part of the process scheduler's set of operations.

**Wait** gets activated whenever the process encounters a busy condition code. Its function is to set the Process Control Block (PCB) of the process to the blocked state. It links to the waiting queue to be sent to critical region.

### NOTES

## NOTES

**Signal** gets activated whenever a process leaves the critical region and the Flag is set to false. Following this, the process is selected to be sent to the READY state.

### Semaphores

A semaphore represents an abstraction of many important ideas in mutual exclusion. It is a protected variable which can be accessed and changed only by operations. It also controls synchronization by using an abstract data type. The synchronization tool called semaphore is used for the solution of the critical section problem, as discussed in the hardware-based solution using TestAndSet () and Swap () instructions. A common example of a semaphore is the flag-like signalling device that is used by railroads for indicating whether or not a track is clear. In case the track is clear, the semaphore's arm is raised. On the other hand, if the track is not clear or is busy, the arm of the semaphore is lowered and the train is not allowed to proceed.

In an operating system, if a resource is free, the semaphore SIGNALS and can be used by the process.

### 5.4.6 Binary Semaphore

A semaphore is a non-negative integer variable upon which two atomic operations, wait and signal, are defined.

```
Int S = 1; /* Let S be a binary semaphore, initialized to 1*/
```

- Wait (s); while  $S \leq 0$  do no operation  $S = S - 1$ ;

A semaphore is an integer variable that accepts non-negative integer values only by using two standard atomic operations: wait () requesting to enter into the critical section, then process  $P_i$  has to wait. So, at a time only one process will be executed or allowed to enter into the critical section. Process  $P_i$  will repeatedly check for the semaphore value to become 1. Then the value will be decremented by 1 ( $S = S - 1$ ) and the process will be allowed to enter into the critical section. The above-mentioned two instructions are executed atomically in order to ensure that no two waiting processes find the semaphore value as 1.

```
Signal (s); S = S + 1;
```

This primitive is executed only when a cooperating process is exiting from the critical section. This operation increments the value of semaphore to 1 in order to ensure that only one of the waiting processes enters into the critical section.

A process  $P_i$  can be synchronized to provide access to its critical section, as follows:

```
do{  
    Wait (& S) ;  
    <Critical Section>  
    Signal (&S);  
    <Remainder Section>  
} while (1);
```

Let us observe the three conditions of an ideal critical-section solution:

- (a) **Mutual Exclusion:** If no process is executing in critical section, semaphore value will be 1. The first process, executing wait operation, will decrement the value to 0 and enter its critical section. The process, which execute 'wait' operation while a cooperating process is executing in its critical section, will find the semaphore value to be 0 and keep looping in the 'while' loop of 'wait' operation. Due to this 'spinning' of a waiting process in the 'while' loop, the binary semaphores are also known as 'spin locks'. 'Signal' operation is executed as soon as a process exits from a critical section. The value of semaphore is incremented to 1. One of the processes in 'while' statement of 'wait' operation will find the value to be 1, exits the 'while' loop, decrements the value by 0 and enters the critical section.

'Wait' operation is to be executed atomically. The other processes will now find the value of semaphore as 0 and continue to loop in the 'while' loop. It means that at a time only one cooperating process is executing in the critical section, subject to satisfaction of the condition that 'wait' operation is executed atomically. So, *the requirement of mutual exclusion is met.*

- (b) **Progress:** When in the critical section, there is execution of any process, the semaphore value is 1. Then, one of the waiting processes, looping in the 'while' loop of 'wait' operation, will find the semaphore to 0 and enter the critical section. So, if there is no execution of any process in the critical section, then one of the waiting processes will soon occupy its critical section. Thus, *the requirement of progress is met.*
- (c) **Bounded Wait:** One of the waiting processes will get an arbitrary entry into its critical section when a cooperating processes executing in its critical section exits. This selection of a process being arbitrary, a process waiting to enter its critical section is likely to face starvation. So, *the requirement of bounded wait is not met.*

#### 5.4.7 Implementation of Semaphores with a Waiting Queue

The main disadvantage of the semaphore definition is that it requires busy-waiting. In other words, a process in critical section prevents all other processes from entering the critical section. Hence, the process should necessarily loop continuously in the entry code. Such a looping faces a severe problem in a real multi-programming system, in which various processes have to share a single CPU. In such a semaphore, the process has to 'spin' while awaiting the lock. This is why this type of semaphore is also termed as 'spinlock'.

In order to overcome the problem of busy-waiting, the process has the feature to block itself. The process is placed in the waiting queue associated with the semaphore by the block operation. Now the process is in the waiting state. The control is transferred to the CPU scheduler, which selects another process to execute.

## NOTES

## NOTES

This blocked process, waiting on a semaphore  $S$ , should be restarted when some other process executes a signal () operation. The process is restarted by wakeup () operation which changes the state of the process from waiting to ready state. The process is now located in the ready queue.

**Table 5.7** Implementation of Semaphores

State Number	Action		Results		
	Calling Process	Operations	Running in Critical Region	Blocked on S	Value of S
0					1
1	P1	P(S)	P1		0
2	P1	V(S)			1
3	P2	P(S)	P2		0
4	P3	P(S)	P2	P3	0
5	P4	P(S)	P2	P3, P4	0
6	P2	V(S)	P3	P4	0
7			P3	P4	0
8	P3	V(S)	P4		0
9	P4	V(S)			1

As presented in Table 5.7,  $P_3$  is placed in the Wait state on state 4. In states 6 and 8, when a process exits the critical region, the value of  $S$  is reset to 1, indicating that the critical region is free. This, in turn, triggers the awakening of one of the blocked processes, its entry into the critical region, and the resetting of  $S$  to zero. In state 7, processes  $P_1$  and  $P_2$  are not trying to do processing in that critical region and  $P_4$  is still blocked. After state 5, the longest waiting process,  $P_3$ , was the one selected to enter the critical-region, but that is only possible in case FCFS algorithm is used. In fact, this purely depends on the choice of the algorithm used by the process scheduler.

### 5.4.8 Conditional Critical Region (CCR)

The Conditional Critical Region (CCR) is a control structure in a higher-level programming language. It provides two features for process synchronization: (i) Providing mutual exclusion over accesses to shared data and (ii) Permitting a process to execute CCR to block itself until a specified Boolean condition becomes true.

Consider the following:

```
Var x: shared <type>:= <initial_value>;
begin
  pbegin
    repeat
```

```

region x do
  begin
    —
    await
    —
    end;
  {Remainder of the cycle}
  Forever
  —
  pend
end

```

## NOTES

Here variable x is shared with the attribute named as shared and can be used in any program. A process in CS uses variable x and executes until it reaches await (B) statement. If condition B evaluates to true, the process continues execution of the CS, else, it returns mutual exclusion and blocks itself awaiting for B to become true. Some other process can now enter the CCR. A process blocked on B is activated sometime in future when condition B is true and no other process is in a CS on x. It now requires the critical section on x and resumes its execution at the statement following await (B).

### Deadlocks and Starvation

If you implement a semaphore with a waiting queue, a situation may arise where two or more processes indefinitely await an event that can be caused only by one of the waiting processes. When such a state is reached, then processes are said to be **deadlocked**.

For example, consider the following two processes,  $P_0$  and  $P_1$ , each accessing two semaphores,  $S$  and  $Q$ . Set to the value 1.

$P_0$	$P_1$
Wait ( $S$ );	wait ( $Q$ );
Wait ( $Q$ );	Wait ( $S$ );
•	•
•	•
•	•
Signal ( $S$ );	Signal ( $Q$ );
Signal ( $Q$ );	Signal ( $S$ );

In the above example, as  $P_0$  executes Wait ( $S$ ), and at the same time,  $P_1$  executes Wait ( $Q$ ). When  $P_0$  executes Wait ( $Q$ ),  $P_0$  must wait until  $P_1$  executes Signal ( $S$ ). Since the execution of these signal ( ) operations is not possible,  $P_0$  and  $P_1$  are deadlocked.

If every process in the set awaits an event which can be caused only by some other process in the set, a process is said to be deadlocked.

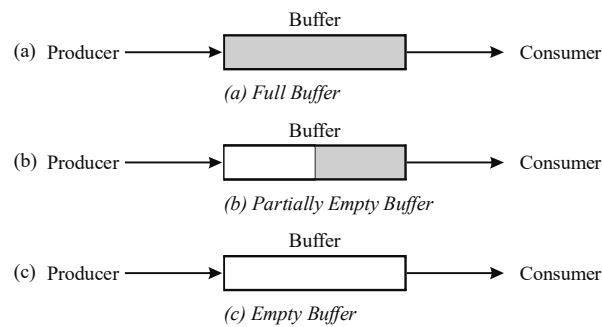
## 5.4.9 Classical Problems in Concurrent Programming

### Bounded Buffer Problem

#### NOTES

The classic problem of producers and consumers is one in which one process produces some data and some other process consumes it. The problem can be expanded to any number of pairs of producers and consumers.

Consider the case of the profile CPU. The pace with which the CPU is able to generate output data is much greater than the pace with which a line printer is able to print it. This involves a producer and a consumer having different speeds. So, a buffer, which offers the producer to temporarily store data to be retrieved by the customer at a greater speed, is required.



**Fig. 5.26** States of Buffer

As shown in Figure 5.26, the buffer can be in any of these three states: (a) full buffer, (b) partially empty buffer or (c) empty buffer.

The synchronization should necessarily prevent the producer from generating more data when the buffer is full as buffer can hold only a finite number of data. On the other hand, it should also prevent the consumer from retrieving the data when buffer is empty. This can only be performed by taking two counting semaphores—one for indicating the total number of full positions in the buffer and the other for indicating the total number of empty positions in the buffer.

The exclusion between processes is ensured by a third semaphore, called *mutex*.

We assume that the buffer pool consists of  $n$  buffers, each capable of holding one item. The *mutex* semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers. The empty semaphore is initialized to the value  $n$  and the full semaphore is initialized to the value 0. Suppose there is a producer process, which produces some data-items for the consumption by a consumer process. The producer is producing the data-items asynchronously.

It is possible that at times the producer may produce data-items at a rate faster than the consumer can consume.

#### **Semaphores Empty, Full, Mutex**

```
Data Buffer [N], Buf_P, Buf_C; /* Buffer is shared memory
for transfer
Of data-items from Producer to Consumer*/
```

```
/* Buf_P is a local buffer of producer*/  
/* Buf_C is a local buffer of consumer*/  
Int p, c; /* p is a pointer of producer to Buffer- where  
next data item is  
to be stored in the Buffer.*/  
/* c is a pointer of consumer to Buffer- where next data-  
item is to be fetched  
from the Buffer*/  
Empty.count = N; /* This indicates that all the N slots  
are empty in buffer */  
Full.count = 0; /* This indicates that none of the slots  
is full in buffer */  
Mutex.count = 1; /* This semaphore is used to achieve  
mutual exclusion  
between producer and consumer, while they transfer data  
in/out of buffer */  
P=0;  
C=0;
```

## NOTES

The producer operates as follows:

```
do {  
    . . .  
    // produce an item in Buf_P  
    . . .  
    Wait (Empty); /* Request an empty slot */  
    Wait (Mutex); /* To achieve mutual exclusion of access of  
buffer */  
    . . .  
    // Buffer [P] = Buffer_P /* Deposit the data item in  
buffer */  
    P = (P + 1) % N /*increment the producer pointer to next  
slot in buffer */  
  
    Signal (Mutex); /* Finished with accessing of buffer */  
    Signal (Full); /* A slot has been filled*/  
} while (1);
```

**The Structure of the Consumer Process**

```
do {  
  
    Wait (Full); /* waiting for a filled slot*/  
    Wait (Mutex); /* to achieve mutual exclusion of access of  
buffer*/  
    Buf_C=Buffer [C]; /* Fetch a data-item from buffer*/  
    C=(C+1)%N; /* Increment the consumer pointer to next slot  
in buffer*/  
    // remove an item from buffer to next slot
```

## NOTES

```
. . .  
Signal (Mutex); /* Finished with accessing of buffer*/  
    Signal (Empty); /* A slot has been emptied*/  
} while (1);
```

### 5.4.10 Readers and Writers Problem

The problem of readers and writers was first formulated by P.J. Courtois, F. Heymans and D.L. Parnas (1971). It occurs when two different kinds of processes have to access a shared resource, which include a file or database.

Airline reservation system is a typical example of the readers and writers problem. Those who seek information about flight are called the readers. As they read only the existing data, they are known as readers. They are not involved in modifying data. Many readers can be active simultaneously as no change in data is required. Thus, we can say that there is no requirement of the enforcement of mutual exclusion in case of readers.

On the other hand, those who are making reservation on a particular flight are referred to as writers. As writers are involved in modifying the existing database, they are required to be carefully accommodated. Thus, in case of the presence of many readers and writers, the enforcement of mutual exclusion is a must as the system cannot allow the reading and writing together.

Courtois, Heymans, and Parnas gave two solutions to this problem:

- (i) In the first solution, the readers are given priority over writers. If a writer is busy in the modification of data, readers have to wait. However, this solution can lead to a situation what is called writer starvation if there exists a continuous flow of readers.
- (ii) In the second solution, writers are given priority over readers. Here, the moment the writers arrive, the existing readers are allowed to finish processing. However, all other readers have to wait until the writer has finished his job. Thus, this solution leads to a condition what is called the readers' starvation if there exists a continuous flow of writers.

Thus, we can conclude that neither solution is desirable.

In solving the writers and readers problem, it is ensured that:

- When a writer is in the critical section, no other reader or writer can execute in critical section.
- When a reader is in its critical section, other readers can also enter their critical sections.

**Semaphore Mutex, Reader\_Writer** /\* count of both semaphore initialized to 1\*/ **Int Read\_count = 0** /\* this indicates that initially none of the readers is in its critical section.\*/

```
A WRITER operates as follows:  
Wait (&Reader_writer);  
<Perform Write Operations>  
Signal (&Reader_Writer)  
A READER operates as follows:
```



```
Wait (&Mutex);  
Read_count= Read_count+1;  
If (Read_count == 1) wait (&Reader_Writer);  
Signal (&Mutex);  
<perform Read Operations>  
wait (&Mutex);  
Read_count= Read_count-1;  
If (Read_count == 0) Signal (&Reader_Writer);  
Signal (&Mutex);
```

In the above algorithm, once a writer enters the critical section; no other reader or writer can enter the critical section. At this time, the readers who arrive first, will wait on the READER\_WRITER semaphore and the subsequent readers will wait on the mutex semaphore itself. If a reader is in the critical section, then other readers will also keep entering in the critical section, as long as there is still at least one reader in the critical section. During this time writers have to wait sometime for a long period and they might get starved.

#### 5.4.11 Deadlocks

A deadlock is a situation in which some processes wait for each other's actions indefinitely. In real life, deadlocks can arise when two processes wait for phone calls from one another, or when persons crossing a narrow bridge in opposite directions meet in the middle of the bridge. Deadlock is more serious than indefinite postponement or starvation because it affects more than one job. Because resources are tied up in deadlocks, the entire system is affected.

Processes involved in a deadlock remain blocked permanently which affects the throughput, resource efficiency and the performance of the operating system. A deadlock can bring the system to standstill.

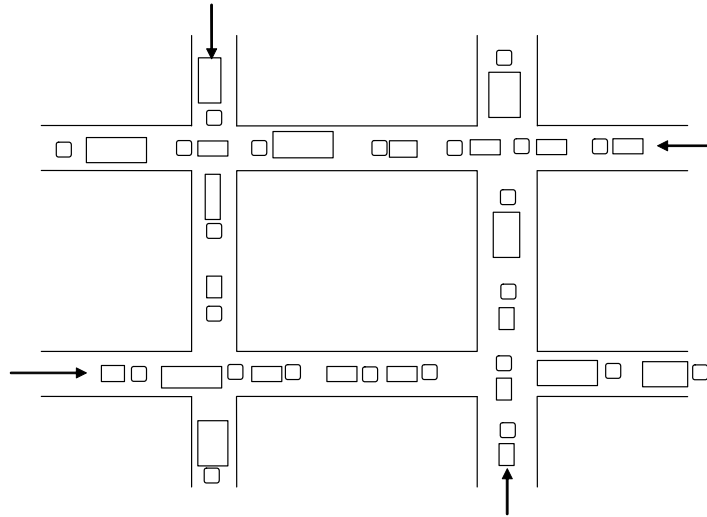
Operating system handles only deadlocks caused by sharing of resources in the system. Such deadlocks arise when some conditions concerning resource requests and resource allocations are held simultaneously.

Deadlock detection detects a deadlock by checking whether all conditions necessary for a deadlock hold simultaneously. The deadlock prevention and deadlock avoidance ensure that deadlocks cannot occur, by not allowing the conditions for deadlocks to hold simultaneously.

The most common example for deadlock is a traffic jam. In the example (Refer Figure 5.27) shown below, there is no proper solution to a deadlock; no one can move forward until someone moves out of the way, but no one can move out of the way until either someone advances or a rear of a line moves back. Only then can the deadlock be resolved.

## NOTES

## NOTES



**Fig. 5.27** A Classic Case of Traffic Deadlock

Thus, we can say that a deadlock refers to a situation in which two or more competing actions are waiting for the other to finish, and thus neither ever does this. For example, consider about the two trains approaching each other at a crossing. In this situation, both the trains stop, and none of them can restart until the other has gone.

In computer science, deadlock refers to a specific condition when two or more processes are each waiting for another to release a resource, or more than two processes are waiting for resources in a circular chain (Wikipedia).

### Some Examples

- The occurrence of deadlocks is common in the multiprocessing system. The reason for this is that in the multiprocessing system, several processes have to share a specific type of mutually exclusive resource known as a *software*, or *soft*, lock. There often exists a *hardware lock* (or *hard lock*) in computers that intend for the *time-sharing* and/or *real-time* markets. This lock provides an *exclusive access* to processes. This leads to a forced serialization. Deadlocks create troubles as we lack a *general* solution to this problem.
- Think of two people drawing diagrams with only one pencil and one ruler between them. If one person possesses the pencil and the other possesses the ruler, this would lead to a deadlock if the person having the pencil needs the ruler and vice versa. As it is not possible to satisfy both the requests, a deadlock is inevitable.
- In case of telecommunications, deadlock is a little more complex. Here, deadlock occurs when neither of the processes meets the condition for moving to another state (as described in the process's *finite state machine*) and each communication channel is empty. The second condition is ignored in case of other systems but is very important in the context of telecommunications.

- We may consider an example of a deadlock in database products. >>Client applications using the database may need an exclusive access to a table. To acquire such an access, a *lock* may be demanded by the applications. Think of a client application holding a lock on a table and attempting to obtain the lock on a second table which is already held by a second client application. This may lead to deadlock if the second application tries to obtain the lock possessed by the first application. (However, this particular type of deadlock is easily prevented, e.g., by using an *all-or-none* resource allocation algorithm.)

## NOTES

### Characteristics of a Deadlock

A deadlock occurs when the following four conditions are met:

- Mutual Exclusion:** Each resource is allocated to only one process at any given point of time.
- Hold and Wait:** The previously granted resources are not released by processes.
- No pre-Emption:** The previously granted resources are not taken away from the processes which hold them.
- Circular Wait:** There exists a chain of two or more processes. These processes should exist in such a way that each process in the chain holds a resource requested by the next process in the chain; there must exist a set  $(P_0, P_1, P_2, P_3, \dots, P_n)$  of waiting processes such that  $P_0$  is waiting for a resource that is held by  $P_1$ ,  $P_1$  is waiting for a resource that is held by  $P_2$ ,  $\dots$ ,  $P_{n-1}$  is waiting for a resource that is held by  $P_n$ , and  $P_n$  is waiting for a resource that is held by  $P_0$  see Figure 5.28.

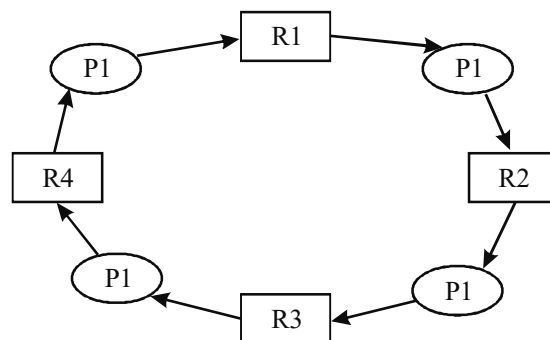


Fig. 5.28 Circular Wait

### 5.4.12 Resource Allocation Graph (RAG)

Three events concerning resource allocation can occur in a system: request for a resource, allocation of a resource and release of a resource (Refer Table 5.8).

A request can occur when some process  $P_i$  makes a request for a resource  $r_i$ . If  $r_i$  is currently allocated to some process  $P_k$ , process  $P_i$  gets blocked on an allocation event  $r_i$ . In effect,  $P_i$  is waits for process  $P_k$  so that  $r_i$  is released. A release event does the task to free  $r_i$  by  $P_k$ .

Table 5.8 Request Allocation and Release of Resources

NOTES

<b>Request</b>	A process requests a resource through a system call. If the resource is free, the kernel allocates it to the process immediately; otherwise, it changes the state of the process to block.
<b>Allocation</b>	The process becomes the holder of the resource allocated to it. The resource state information gets updated and the process' state changes to ready.
<b>Release</b>	A process releases a resource through a system call. If some processes are blocked on the allocation event for the resource, the kernel uses some tie-breaking rule, e.g. FCFS allocation, to decide which process should be allocated the resource

Symbols used in RAG

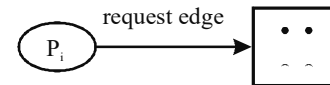
1. Process



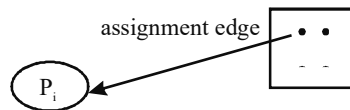
2. Resource type with four instances



3.  $P_i$  requests instances of  $R_j$



4. Process  $P_i$  is holding an instance of  $R_j$



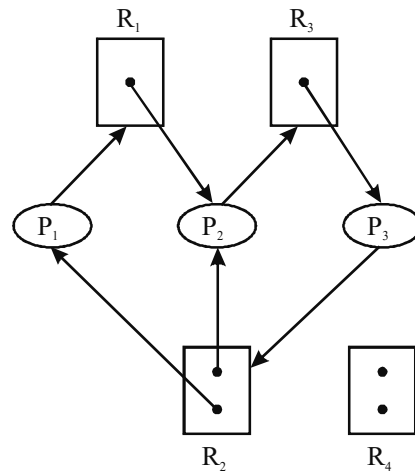
Basic facts:

1. If a graph contains no cycles, no deadlock.
2. If a graph contains a cycle
  - a. If only one instance per resource type, then a deadlock.
  - b. If several instances per resource type, possibility of a deadlock

**Possibility 1 for a Deadlock**

Figure 5.29 shows possibility 1 for a deadlock.

**NOTES**



Fig, 5.29 Possibility 1 for a Deadlock

$$E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$$

Two possibility cycles are:

- $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
- $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

**Possibility 2: Cycles but No Deadlock**

Figure 5.30 shows possibility 2 for a deadlock.

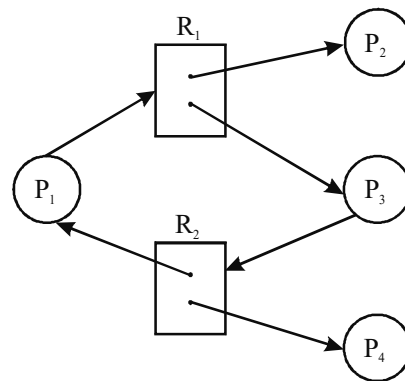


Figure 5.30 Possibility 2 for a Deadlock

Cycle is  $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$ .

No deadlock, observe that process  $P_4$  may release its instance of resource type  $R_2$ . That resource can then be allocated to  $P_3$  breaking this cycle.

**5.4.13 Methods for Handling Deadlocks**

There are three methods to handle a deadlock:

1. Deadlock Prevention: It ensures that the system will never enter a deadlock state
2. Deadlock Avoidance: It allows the system to avoid a deadlock.
3. Deadlock Detection: It allows the system to enter a deadlock and then recover

- (a) Ignore the problem, and pretend that deadlock never occur in the system

### Deadlock Prevention

#### NOTES

Let us discuss the four conditions that should be met in order to produce a deadlock:

(a) **Mutual Exclusion:**

- It is not required for sharable resources.
- It must hold for non-sharable resources.

(b) **Hold and Wait:**

- Require a process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.
- It may lead to low resource allocation.
- Starvation is a problem. A process may be held for a long time waiting for all its required resources.
- If it needs additional resources, it releases all of the currently held resources and then requests all of those it needs; one should be aware of all the needs well in advance

(c) **No Preemption:**

- Preempting a resource is necessarily a compulsory sharing. It can be applied to such devices whose state can be saved and later restored.
- If a process that is holding some resources requests another request that can not be immediately allocated to it, then all the resources currently being held are released.
- Preempted resources can be added to the list of resources the process is waiting for.
- Process can be restarted only if it can regain its old and new resources it is requesting in an alternative manner.

When a process requests some resources, first of all we check whether or not they are available. If yes, they are allocated. If no, we check if they are allocated to some other process waiting for some additional resources. If they are so allocated, the desired resources are pre-empted from the waiting process and are allocated to the requesting process. In case the resources are neither available nor held by a waiting process, the requesting process has to wait.

(d) **Circular Wait:**

- It imposes a total ordering on all resources types.
- It requires each process to request resources only in a strict increasing order.
- Resources from the same resources type have to be requested together.

Let  $R = \{R_1, R_2, R_3 \dots, R_n\}$  be the set of resources types. Then one- to-one function is defined as:

$F: R \rightarrow N$ , where  $N$  is the set of natural numbers.

$R$  {tape drive, disk drive, printer}

Then,

$F$  (tape drive) = 1,

$F$  (disk drive) = 5,

$F$  (printer) = 12.

### Protocol

- Each process can make a request only in an increasing order. In other words, any number of instances of resource type  $R_i$  can be initially requested by a process. Following this, the process can request instances of resource type  $R_j$  if and only when  $F(R_j)$  is greater than  $F(R_i)$ . If you need several instances of the same resource type, you need to issue a single request for all of them.
- Alternatively, we may need that whenever a process makes a request of an instance of resource type  $R_j$ , it has released any resource  $R_i$ , such that  $F(R_i)$  is greater than equal to  $F(R_j)$ .

If the above-mentioned two protocols are used, then the circular wait-condition can not hold.

### Deadlock Avoidance

We have seen that in deadlock prevention, one of the conditions that must be present for a deadlock to exist is prevented. In deadlock avoidance, a resource which may eventually lead to a deadlock is never allocated. This can be done by allocating resources to just one process at a time.

### Safe State

A system is considered to be in a safe state if there exists a safe execution sequence. By execution sequence we mean an ordering for process execution such that each process, on being executed, runs so long as it does not terminate or is blocked, and all requests for resources are immediately granted in case of their availability. A safe execution sequence means an execution sequence wherein all processes run to completion.

Hence a safe state is one where:

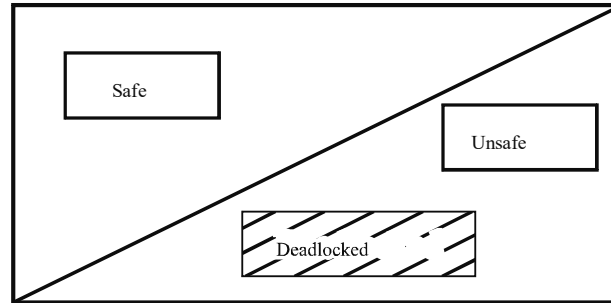
- There is no deadlock.
- There is some sequence by which all requests can be satisfied.
- To avoid deadlocks, we try to make only those transactions that will take us from our safe state to another.

### Unsafe State

An unsafe state refers to a state that is not safe, not necessarily a deadlocked state.

### NOTES

**NOTES**



**Fig. 5.31** Safe and Unsafe States

An unsafe state may currently not be deadlocked, but there is at least one sequence of request from process that would make the system deadlocked Figure 5.31 shows and Unsafe States.

**Simplest Algorithm**

Each process tells a max number of resources it will ever need. As a process runs, it requests resources but never exceeds the max number of resources. System schedules process and allocate resources in a way that ensures that no deadlock results. For example, the system has twelve tape drives.

Process	Max need	Current needs
$P_0$	10	5
$P_1$	4	2
$P_2$	9	2

Can a system prevent a deadlock even if all processes request the max? Well, right now, the system has three free tape drives.

If  $P_1$  runs just and completes, it will have  $(3+2)=5$  free tape drives.  $P_0$  can run to completion with those 5 free tape drives even if it requests max. Then  $P_2$  will execute. So, this schedule will execute without a deadlock.

If  $P_2$  requests two or more tape drives, can system give it the drives? No, because it cannot be sure that it can run all jobs to completion with only 1 free drive. So, the system must not give  $P_2$  two more tape drives until  $P_1$  finishes. If  $P_2$  asks for two more tape drives, the system suspends  $P_2$  until  $P_1$  finishes.

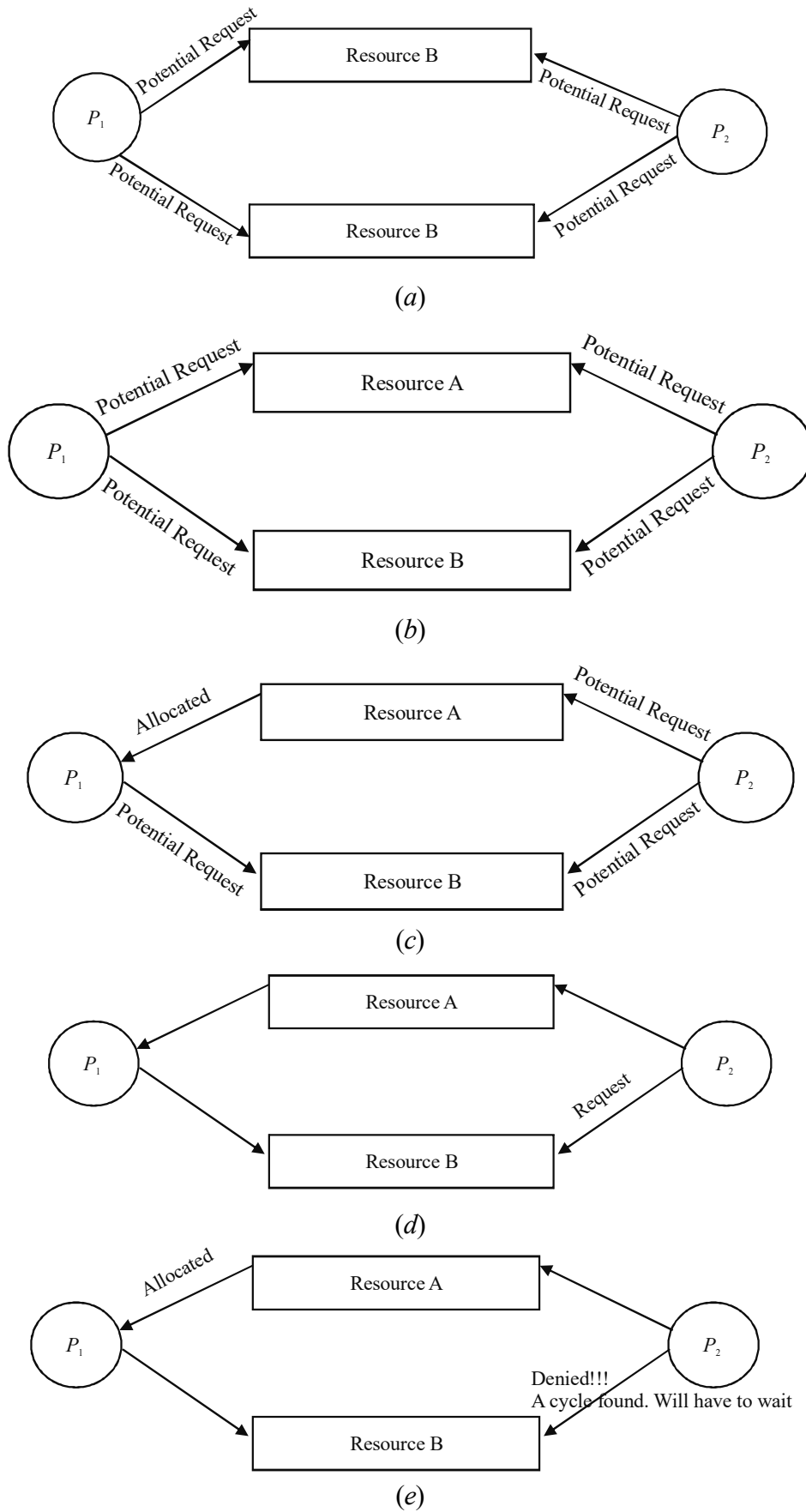
**RAG Algorithm**

Figure 5.32 shows stages of RAG algorithm. The features of RAG algorithm are as follows:

- It maintains a graph with a directed edge from each process to each resource it might request. (Needs a prior knowledge)
- Allocated resource reverses the edge direction.
- Released resource returns the edge to its original direction.
- The algorithm allocates a resource only if it can do that without creating a cycle in the graph.



**NOTES**



**Fig. 5.32** Stages of RAG Algorithm

## NOTES

### Banker's Algorithm

RAG is not applicable to a resource allocation system with multiple instances of each resource type. The deadlock avoidance algorithm that we will be describing next is applicable to such a system. However, this algorithm, commonly known as *Banker's algorithm*, is not as efficient as RAG scheme.

When a process enters the system, it must declare the maximum number of instances of each resource type that it may need. When a user requests a resource, we must ensure that after allocating it the system must be in a safe state. If it is ensured, the resources are allocated; else, the process has to wait until some other process releases resource types.

#### Data Structures

- **Available:** vector  $[1, \dots, m]$ , //  $m$  indicates the number of available resources of each type.

If available  $[j] = k$  // the number of instances currently available for resource  $j$ .

- **Max:** matrix  $[1, \dots, n, 1, \dots, m]$ ,  $\text{Max}[i, j] = k$  // the maximum number of instances of resource  $j$  that process  $i$  can request at any one time.
- **Allocation:** matrix  $[1, \dots, n, 1, \dots, m]$ ,  $\text{Allocation}[i, j] = k$  // Process  $i$  currently holds an instance of resource  $j$ .
- **Need:** matrix  $[1, \dots, n, 1, \dots, m]$ , // Process  $i$  needs more (additional) resource (instances) of type  $j$ .

$\text{Need}[i, j] = \text{Max}[i, j] - \text{Allocation}[i, j]$

#### Banker's Algorithm—Safety Procedure

1. Let Work and Finish be two vectors defined as:

Work  $[1, \dots, m]$ , Finish  $[1, \dots, n]$

Initialize Work = Available and Finish  $[i] = \text{false}$  for  $i=1,2,3,\dots,n$

2. Find an  $I$  such that:

a. Finish $[i] = \text{false}$

b. Need $_i \leq \text{Work}$

If yes, GOTO step 4

3. Work = Work + Allocation;

Finish $[i] := \text{true}$

GOTO step 2

a. If Finish  $[i] = \text{true}$  for all  $i$ , then the system is in safe state.

#### Resource-Request Algorithm

Let Request $_i$  be the request vector for process  $P_i$ . If Request $_i[j] = k$ , then process  $P_i$  wants  $k$  instances of resource type  $R_j$ . When a request for resources is made by process  $P_i$ , then following actions are taken:

1. If  $\text{Request}_i \leq \text{Need}_i$ , go to step 2. Otherwise, raise an error condition, since the process has executed its maximum claim.
2. If  $\text{Request}_i \leq \text{Available}$ , go to step 3, otherwise,  $P_i$  must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process  $P_i$  by modifying the state as follows:  
 $\text{Available} = \text{Available} - \text{Request}_i$   
 $\text{Allocation} = \text{Allocation} + \text{Request}_i$   
 $\text{Need}_i = \text{Need}_i - \text{Request}_i$

If the resulting resource-allocation state is safe, the transaction is completed and process  $P_i$  must wait for  $\text{Request}_i$  and the old resource-allocation state is restored.

**Example 5.6:**

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	7	5	3	3	3	2
$P_1$	2	0	0	3	2	2			
$P_2$	3	0	2	9	0	2			
$P_3$	2	1	1	2	2	2			
$P_4$	0	0	2	4	3	3			

**Need = Max–Allocation**

Process	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
							2	3	0
$P_0$	0	1	0	7	4	3			
* $P_1$	3	0	2	0	2	0			
$P_2$	3	0	2	6	0	0			
$P_3$	2	1	1	0	1	1			
$P_4$	0	0	2	4	3	1			

Now we have to perform the safety algorithm to find the safe sequence. After executing our safety algorithm, we find that the safe sequence is  $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ . So, the request for  $P_1$  can be immediately granted.

Now suppose that  $P_2$  requests one additional resource of type A and two of type C, so request  $\langle 1, 0, 2 \rangle$  is sent to the OS. For this, first we have to check the condition available.

Now suppose that process  $P_1$  requests one additional instance of resource type A and two instances of resource type C,

So request1 = (1, 0, 2)

request1 ≤ available

[(1, 0, 2) ≤ (3, 3, 2)], which is true.

**NOTES**

**NOTES**

**Safety Algorithm**

(i) Work =  $\langle 2 \ 3 \ 0 \rangle$

(ii) Work =  $\langle 2 \ 3 \ 0 \rangle + \langle 3 \ 0 \ 2 \rangle = \langle 5 \ 3 \ 2 \rangle \langle P_1, P_3, P_4, P_2, P_0 \rangle$

(iii)  $\langle 5 \ 3 \ 2 \rangle + \langle 2 \ 1 \ 1 \rangle = \langle 7 \ 4 \ 3 \rangle$

What if  $P_4$  then requests (3, 3, 0)? And if  $P_0$  requests (0, 2, 0)?

**Deadlock Detection**

In RAG, a direct arrow is drawn from the process to the resource rectangle to represent each pending resource request. To indicate each granted request, on the other hand, a direct arrow is drawn from a resource dot to the process (Figure 5.33).

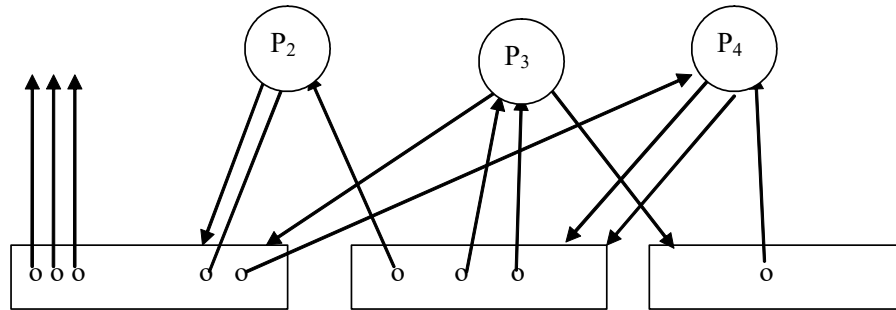


Fig. 5.33 Resource Allocation Graph

Table 5.9 Resource Usage

Process	Current Allocation			Outstanding Requests			Resources Available		
	$R_1$	$R_2$	$R_3$	$R_1$	$R_2$	$R_3$	$R_1$	$R_2$	$R_3$
$P_1$	3	0	0	0	0	0	0	0	0
$P_2$	1	1	0	1	0	0			
$P_3$	0	2	0	1	0	1			
$P_4$	1	0	1	0	2	0			

To reduce a RAG, you need to check the arrows associated with each process and resource (Refer Table 5.9).

- If a resource contains only arrows pointing away from it (meaning it has no request pending), all its arrows are erased.
- If a process contains only arrows pointing towards it (meaning all its requests have been granted), all its arrows are erased.

If a process has arrows pointing away from it, but for each such request arrow, there is an available resource dot (a dot without an arrow leading from it) in the resource the arrow points to, erase all processes arrow (Refer Figure 5.34).

NOTES

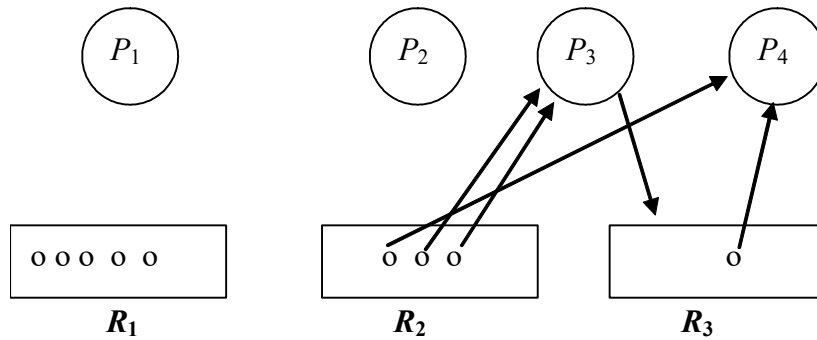


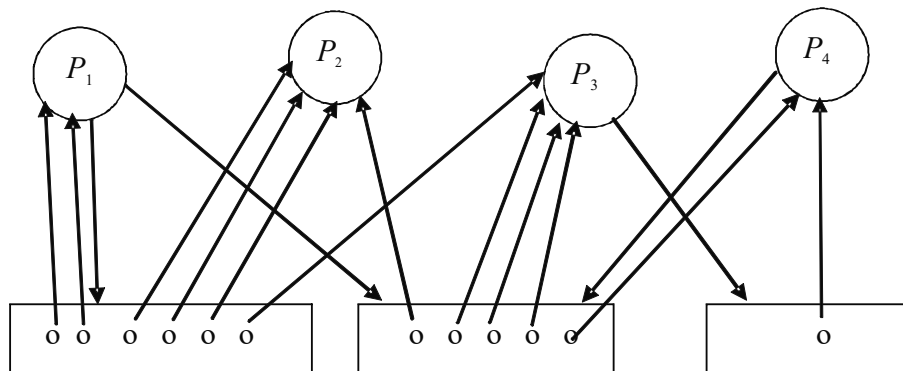
Fig. 5.34 Reduced Resource Allocation Graph

**Example 5.7:**

From the following resource usage table, draw the resource allocation graph.

Process	Current Allocation			Outstanding Requests			Resources Available		
	$R_1$	$R_2$	$R_3$	$R_1$	$R_2$	$R_3$	$R_1$	$R_2$	$R_3$
$P_1$	2	0	0	1	1	0			
$P_2$	3	1	0	0	0	0	0	0	0
$P_3$	1	3	0	0	0	1			
$P_4$	0	1	1	0	1	0			

**Solution:**

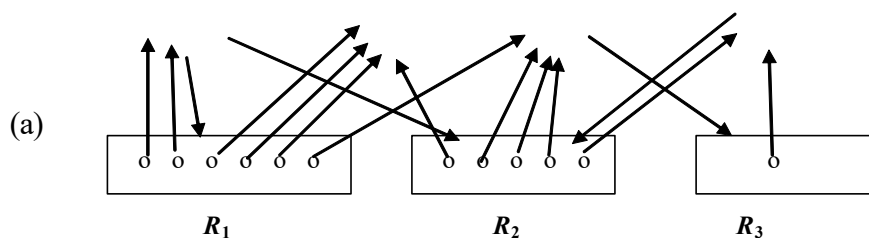


Resource allocation graph

**Example 5.8:**

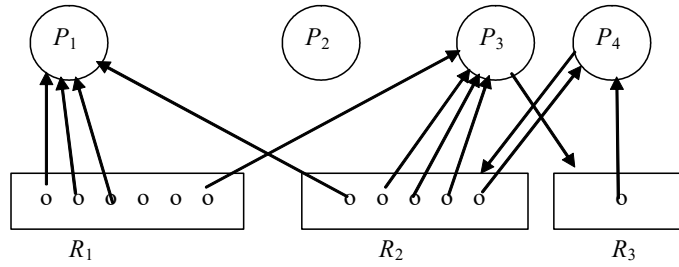
From the resource usage table given in Example 5.6, draw the reduced resource allocation graph.

**Solution:**

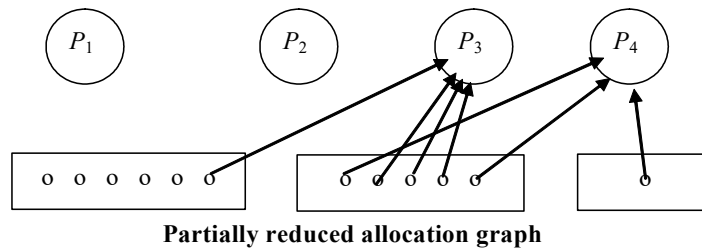


**NOTES**

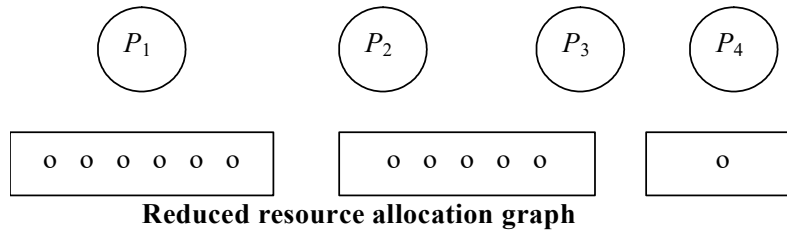
(b) Process  $P_2$  only has arrows pointing towards it. So, the links can be erased and requests can be granted to process  $P_1$  from the resources  $R_1$  and  $R_2$ . Hence, we have the following:



(c) Process  $P_1$  only has arrows pointing towards it. So, the links can be erased and requests can be granted to process  $P_3$  from resource  $R_2$ . Hence, we have the following:



(d) Processes  $P_3$  and  $P_4$  only have arrows pointing towards them. So, the links can be erased.

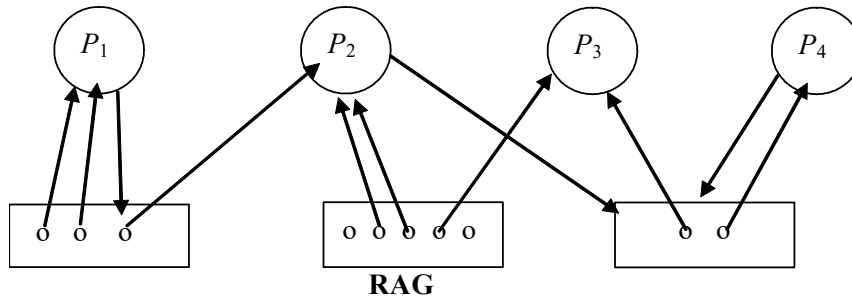


**Example 5.9:**

From the given table, draw the resource allocation graph.

Process	Current Allocation			Outstanding Requests			Maximum Allocation			Resources Available		
	$R_1$	$R_2$	$R_3$	$R_1$	$R_2$	$R_3$	$R_1$	$R_2$	$R_3$	$R_1$	$R_2$	$R_3$
$P_1$	2	0	0	1	0	0	0	2	0	1	0	20
$P_2$	1	2	0	0	0	0	1	2	5	2		
$P_3$	0	1	1	0	0	0	0	1	4	2		
$P_4$	0	0	1	0	0	0	1	2	0	1		

**Solution:**



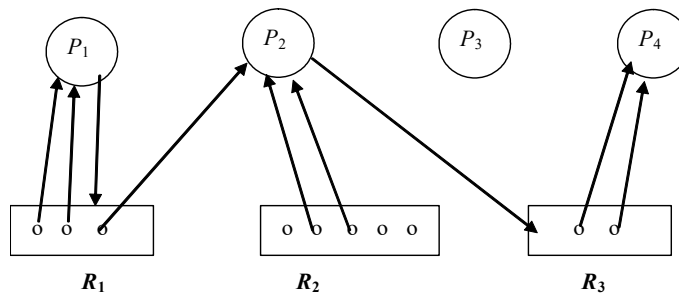
**NOTES**

**Example 5.10:**

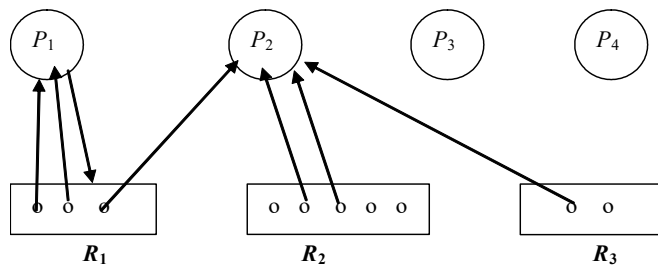
From the table given in Example 5.7, draw the reduced RAG.

**Solution:**

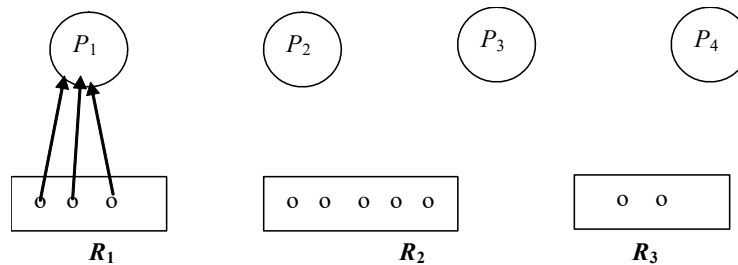
- (a) Process  $P_3$  has only arrows towards it. So, erase those arrows and allocate the resources to process  $P_4$ :



- (b) Process  $P_4$  has only arrows pointing towards it. So, they can be erased and the resource can be allocated to process  $P_2$ .

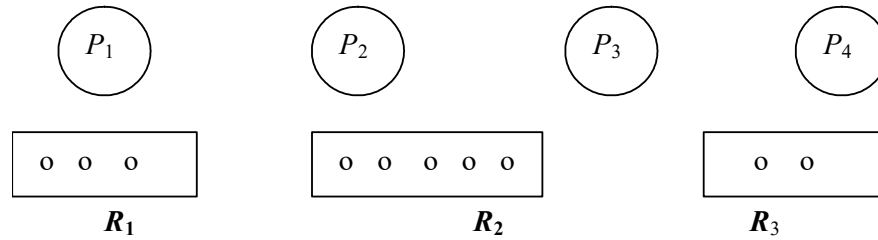


- (c) Process  $P_2$  has only arrows pointing towards it. So, they can be erased and the resource from  $R_1$  can be given to process  $P_1$ .



- (d) Process  $P_1$  has now all the arrows pointing towards it. So, they can be finally erased to get the reduced resource allocation graph.

## NOTES



### 5.4.14 Introduction to File System and IO

A file is a logical collection of information. A file system is a collection of files. It may also include a variety of other objects that share many of properties of files (such as I/O devices).

#### Attributes of a File

The attributes of a file are as follows:

**Name:** The symbolic file name is the only information kept in human readable form.

**Identifier:** This unique tag, usually a number, identifies a file within the file system. It is the non-human-readable name for the file.

**Type:** This information is needed for the systems that support different types.

**Location:** This information is a pointer to a device and to the location of this file on that device.

**Size:** The current size of the file, and possibly the maximum allowed size, are included in this attribute.

**Protection:** Access control information determines who can do reading, writing, executing, and so on.

**Time, Date and User Identification:** This information may be kept for creation, last modification and last use. These data can be useful for protection, security and usage monitoring.

#### File Operations

The various file operations are as follows:

**Creating a File:** There are two requirements for creating a file. First, there must be space in the file system for the file. Second, an entry, for the file must be made in the directory.

**Writing a File:** For writing a file, we make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the location of the file.

**Reading a File:** For reading from a file, we use system call specifying the name of the file, and where the next block of the file should be put. Pointer is used to write to the file.

**Repositioning within a File:** The directory is searched for the appropriate entry, and the current file-position is set to a given value.



**Deleting a File:** For deleting a file, we search the directory for the named file. Having found the associated directory entry, release all file space, so that it can be used by other files, and erase the directory entry.

**Truncating a file:** The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this f<sup>n</sup> allows all attributes to remain unchanged – except for the length but lets the file be reset to length zero and its file space released.

## NOTES

### Structure of a Directory

File system allows user to organize file and other file system objects through the use of directories (Refer Figure 5.35).



*Fig. 5.35 Structure of a Directory*

Operations that are to be performed on a directory are as follows:

- Search for a File
- Create a File
- Delete a File
- List a Directory
- Rename a File
- Traverse the File System

### 5.4.15 Organizing Files

A file organization defines two things concerning a file: the arrangement of records in the file and the procedure to be used to access the records. It determines how efficiently the I/O medium would be used. It exploits the characteristics of I/O device to provide the file processing efficiency for a specific access pattern. For example, a disk record has a unique address and a read/write operation can be performed on any disk record by specifying its address.

On magnetic disks, files can be organized in one of the three ways: sequential, direct, or indexed sequential.

### **Sequential File Organization**

#### **NOTES**

In the sequential file organization, records are stored in an ascending or descending sequence by the key field. They are the easiest to implement because records are retrieved serially, one after the other. To find a specific record, the file is searched from its beginning until the requested record is found. It supports two kinds of operations: read the next (or previous) record and skip the next (or previous) record.

Most I/O devices can be accessed sequentially. Hence, sequential files are most crucial dependent on device characteristics.

### **Direct Access File Organization**

The direct file organization provides convenience and efficiency of file processing when records are accessed in a random order. These can be implemented only on direct access storage devices. These files give users the flexibility of accessing any record in any order without having to begin a search from the beginning of the file to do so. Hence, it is known as 'random organization', and its files are called as 'random access files'.

There are two types of addresses, relative addresses and logical addresses, through which the records can be identified. Relative address allows finding the relative address of the beginning of the file, while logical address is required when the records are stored and retrieved from a file.

The method is quite simple in which a user identifies a field and designates it as a key field as it can uniquely identify the records. The program used to store the data follows a set of instructions called a hashing algorithm, which transforms each key into a number and records logical address. This is given to the file manager, which takes the necessary steps to translate the logical address into a physical address.

Direct access files can be updated more quickly than sequential files because records can be quickly rewritten to their original address after modifications have been made. Since the order of storing the records is not mandatory, adding or deleting records is quite easier.

### **Indexed Sequential File Organization**

It combines both the sequential and direct access. It is created and maintained through Indexed Sequential Access Method (ISAM) software package, which removes the burden of handling overflows and preserving record order from the shoulders of the programmer. An index helps to determine the location of a record from its key value. It contains index entries such as key, disk address for all key values existing in a file. To access a record with key  $k$ , the index entry containing  $k$  is found by searching the index, and the disk address mentioned in the entry is used to access the record. If the index is smaller than a file, this arrangement provides high access efficiency because search in the index is more efficient than search in the file.

To select the best of these three types of file organization, the programmer or analysts usually considers the following practical characteristics:

- Volatility of the data
- Activity of the file
- Size of the file
- Response time

## NOTES

### 5.5 CACHE COHERENCE

Cache memory is assembled between main memory and CPU. It is considered as semi-conductor memory, which is made up of static RAMs. The accessing time of cache memory is approximately 10 ns, whereas main memory is about 50ns. Basically, it keeps the data and instruction codes. The main function of cache memory is to reduce the average accessing time for data and instruction codes. These are basically stored in the main memory. For this, cache memory uses cache controller. In these days, special integrated chips are used in cache controller. The 32-bit and 64-bit microprocessors work in high speed and corresponding clock rates that lie between 400 MHz and 1000 MHz. Caches perform well in transferring data frequently. It is required because all processors are supposed to share the same address as other processors do. It is used with data items at a time. If specified data items are updated by one processor without informing the other processors, it causes incorrect executions and inconsistencies. A protocol is maintained in the multiprocessor system if data items are lost or overwritten before transferring the data from cache to target memory. Multiple processors use separate cache sharing the common memory. It is essential to keep the cache in coherence state because the shared operand can be changed throughout the entire system. This mechanism utilizes two methods. The first method prefers through a directory based sharing and second method prefers snooping method. The directory-based system maintains a common directory in which coherence takes place between various caches. Directory works here as a ‘filter’ in which processor is given a permission to load the information and data from primary memory to cache memory.

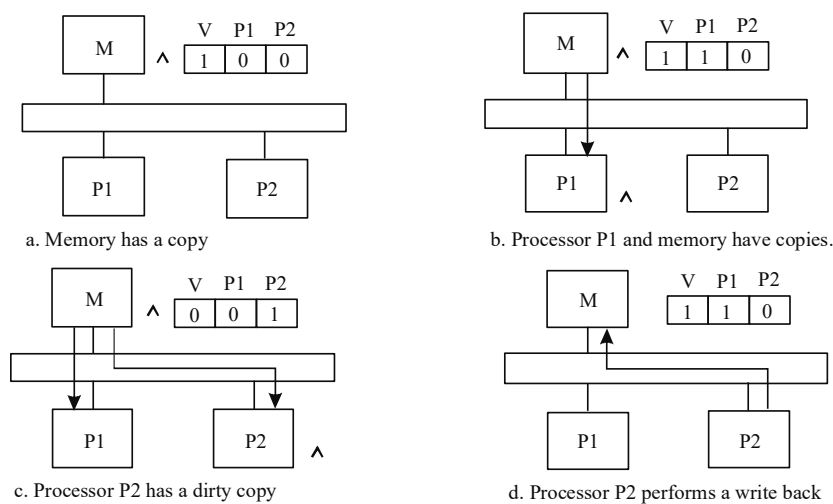
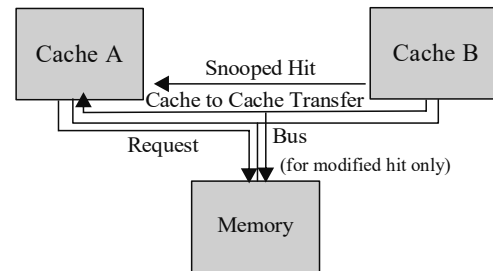


Fig. 5.36 Directory Sample

In Figure 5.36, part (a) represents memory has a copy whereas part (b) represents processor  $P_1$  and memory has copies. Processor  $P_2$  keeps a dirty copy in part (c) and processor  $P_2$  performs a write-back operation part (d).

## NOTES



**Fig. 5.37** Snooping Read-Hit in Cache-to-Cache Transfer

In figure 5.37, cache **A** requests from the memory controller carrying address which is snooped by cache **B**. After reading the snooped address by cache **B**, it sends hit-modified signal (**HIT#**). Bus is also involved in requesting the data. To get this idea, let take an example. There is cache block about 32 bytes long. It takes 32 bytes from DRAM acquiring 1 clock arbitration for the bus and one clock is put to refer the address onto the address bus. The requirement of 6 clocks is to access the DRAM, 2 clocks are required to transfer the data and 10 clocks are required to write back the data. The entry for updating and invalidating data items can sometimes change other caches along with the entry. It supports write-invalidate and write-update in the following way:

- **Write-Invalidate:** a processor gains exclusive access of a block before writing by invalidating all other copies.
- **Write-Update:** when a processor writes, it updates other shared copies of that block.

Whereas, snooping caches the bus monitors known as ‘**snoop**’ to copy the block of requested data on the bus. The following conditions are required if cache memory system is coherent:

- $P$  writes to  $X$ , no other processor writes to  $X$ ,  $P$  reads  $X$  and receives the value previously written by  $P$
- $P_1$  writes to  $X$ , no other processor writes to  $X$ , sufficient time elapses,  $P_2$  reads  $X$  and receives value written by  $P_1$ .
- Two writes to the same location by two processors are seen in the same order by all processors known as write serialization.
- The memory consistency model defines ‘time- elapsed’ before the effect of a processor is shared by others.

In the preceding conditions,  $P_1$  and  $P_2$  are the processors and  $X$  represents the data items. Caching data means the multiple copies of data are available for processing, for example, copy in main memory, secondary memory cache and primary memory cache. The main problem with cache coherence is that all cache copies of data items are true so that available data items in main memory reflecting the same data items. In these days, suitable hardware is assembled to achieve the cache coherence. It does not make any effect on software. A kernel-device driver maintains cache coherence by disabling the data cache on each processor. The data

cache is accessed input-output space. It prevents burst reads or writes that supports wasting of huge number of cycles for every time if processors access data. The main task of cache coherence is to suppress the problems generated by sharing data. Table 5.10 shows the cache protocol used in removing the problem in cache coherence.

Table 5.10 Cache Protocol

Request	Source	Block state	Action
Read hit	Proc	Shared/excl	Read data in cache
Read miss	Proc	Invalid	Place read miss on bus
Read miss	Proc	Shared	Conflict miss: place read miss on bus
Read miss	Proc	Exclusive	Conflict miss: write back block, place read miss on bus
Write hit	Proc	Exclusive	Write data in cache
Write hit	Proc	Shared	Place write miss on bus
Write miss	Proc	Invalid	Place write miss on bus
Write miss	Proc	Shared	Conflict miss: place write miss on bus
Write miss	Proc	Exclusive	Conflict miss: write back, place write miss on bus
Read miss	Bus	Shared	No action; allow memory to respond
Read miss	Bus	Exclusive	Place block on bus; change to shared
Write miss	Bus	Shared	Invalidate block
Write miss	Bus	Exclusive	Write back block; change to invalid

Cache coherency reads the block of memory known as cache line are sent to a memory cache. The size of cache line is fixed (range starts from 16 bytes to 256 bytes). Line size is determined as per application. For this, cache circuits are configured by the system designers.

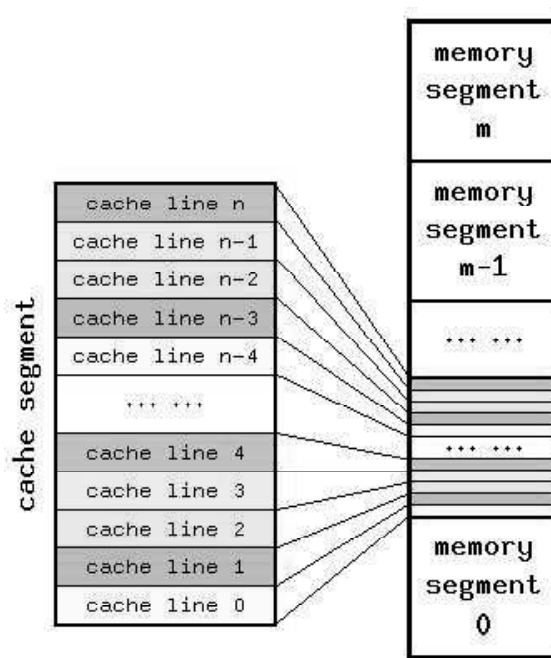
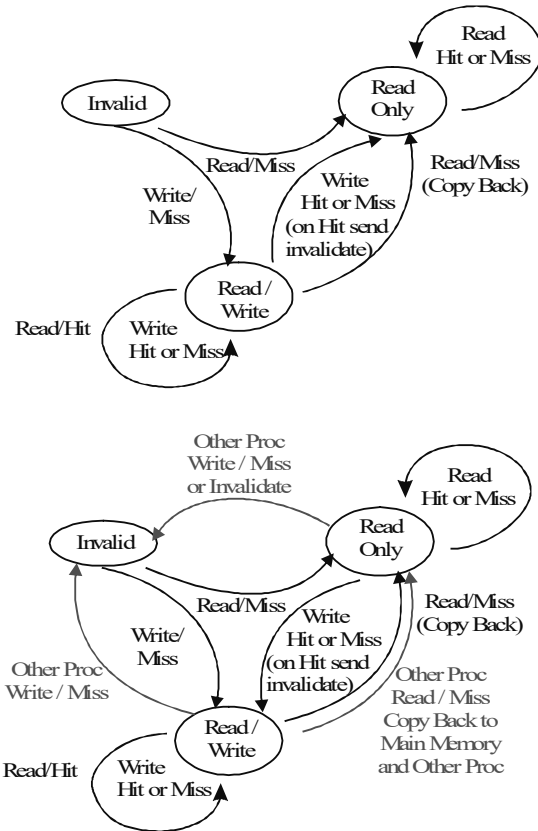


Fig. 5.38 Cache Segment

## NOTES

**NOTES**

In Figure 5.38, the cache line starts from cache line 0 to cache line n which makes collectively cache segment. The pending reads can be tracked with read resources. After requesting read resources, a corresponding read response appears on the bus.



**Fig. 5.39** Hit and Miss Cache Coherence

In Figure 5.39, the two types of cache schemes are used. They are known as write-through and write-back schemes. The updating of main memory is possible through cache. The main memory contains same data as write-through contains. This mechanism is used in Direct Memory Access (DMA) for transforming the data. The solution of cache coherence problem is solved by two methods:

- The shared writable data is kept as non-cacheable.
- The writable data maintains centralized global table which exists in one cache.
- Snoopy cache controller is used for write operation.

**Check Your Progress**

6. Why are scatter and gather operations used in vector processing?
7. Define semaphore.
8. What is the main function of cache memory?

---

## 5.6 ANSWERS TO ‘CHECK YOUR PROGRESS’

---

1. Pipelining is an effective method of increasing the execution speed of the processor.
2. Arithmetic pipeline is found in high-speed computer. It is well implemented in systems involved with matrices and vectors.
3. The instruction mainly involves the following sequences of steps:
  - Instruction fetch
  - Instruction decode
  - Calculate address
  - Operand fetch
  - Operation execution
  - Result storage
4. The following strategies are employed in resolving control dependencies due to branch instructions:
  - Assume branch not taken
  - Reducing branch delay
  - Dynamic branch prediction
  - Branch target buffer
  - Loop buffer
  - Branch delay slot
5. Some of the areas of application of vector processing are weather forecasting, artificial intelligence, experts system, image processing, seismology, gene mapping and aerodynamics.
6. Scatter and gather operations are used to process sparse matrices/vectors where only certain elements of a vector are needed in a computation.
7. A semaphore represents an abstraction of many important ideas in mutual exclusion. It is a protected variable which can be accessed and changed only by operations. It also controls synchronization by using an abstract data type.
8. The main function of cache memory is to reduce the average accessing time for data and instruction codes. These are basically stored in the main memory.

### NOTES

---

## 5.7 SUMMARY

---

- Pipeline computers are those computers where a computer uses a sequence of stages (also known as segments) to execute an instruction.
- In a digital computer, the execution of an instruction mainly involves, fetching the instruction from computer memory, decoding the fetched instruction, identifying the operation to be performed and executing the instruction.

## NOTES

- In a pipelined computer the operations of all segments operate concurrently under a common clock cycle.
- The behaviour of a pipeline can be illustrated with a space-time diagram. The time in clock cycles is given along the horizontal axis and the vertical axis gives the segment number.
- Arithmetic pipelining is well implemented in the systems concerned with repeated calculations involved with matrices and vectors.
- An instruction pipeline understands the consecutive instruction from the memory when the preceding instruction was being executed in the other segments.
- The instruction pipeline deviates from its normal execution in case of resource conflicts, data dependency and branch difficulties.
- An arithmetic pipeline is a pipeline in which different stages of an arithmetic operation are handled along with the stages of a pipeline.
- An instruction pipeline operates on a stream of instructions by overlapping the fetch, decode and execute phases of the instruction cycle as different stages of pipeline.
- An efficient way to use instruction pipeline is a characteristic feature of RISC architecture. The length of the pipeline is dependent on the length of the longest step.
- Computational problems with very high computational loads which are beyond the capabilities of a conventional computer are performed using vector computations for faster processing.
- A vector can be defined as an ordered set of one-dimensional array of data items. Vector processor unit is used to perform the vector operations efficiently.
- Chaining eliminates the need to store the result of the first pipeline before sending it into the second pipeline.
- By ordering successive computations in the array the vector array processing can be classified into horizontal processing, vertical processing and vector looping.
- An array processor is a specific processor type that performs the required computations on huge arrays of data. It can be created using a group of unique special processors which are specifically designed for calculating mathematical procedures at extremely high speeds and are frequently under the control of another central processor.
- There are two different types of array processors, an attached array processor and a SIMD array processor.
- The algorithms are specifically used as unique functions on a two-dimensional mesh in the pattern-matching algorithm. An array of  $n$  processors is constructed from processors  $P_0 \dots P_{n-1}$ , where  $P_i$  is connected to  $P_{i-1}$  and  $P_{i+1}$ , if exists.
- The SIMD array processor consists of a memory, an array control unit and the two-dimensional array of simple processing elements.



- Multiprocessors use two or more than two CPUs assembled in a single system unit. It refers to the execution of various software processes concurrently.
- An interprocess arbitration system for multiprocessors shares a common bus. Arbitration fixes the priority criteria set by the multiprocessors.
- Process synchronization is a mechanism used by the OS to ensure a systematic sharing of resources amongst concurrent resources.
- Cache memory is assembled between main memory and CPU. It is considered as semiconductor memory, which is made up of static RAMs.

## NOTES

---

### 5.8 KEY TERMS

---

- **Pipeline Computers:** It refers to those computers which uses a sequence of stages to execute an instruction.
- **Space-Time Diagram:** It is a diagram that shows the segment utilization as a function of time.
- **Speed-Up Ratio:** It is the ratio between the maximum times taken by non-pipeline processors over the processors that use pipeline.
- **Throughput:** It is the number of tasks completed by a pipeline per unit time.
- **Mutual Exclusion:** When one process is in a critical section that accesses a set of shared resources, no other processes can be in a critical section accessing any of those shared resources.
- **Bounded Wait:** When a process requests access to a critical section, a decision that grants it access may not be delayed indefinitely.
- **Critical Section:** This refers to the code segment where a shared resource is accessed by the process. At a time, only one of the cooperating process can enter into the critical section. Hence the shared resource is being used by only one process.

---

### 5.9 SELF-ASSESSMENT QUESTIONS AND EXERCISES

---

#### Short-Answer Questions

1. What is the use of arithmetic pipeline?
2. How does an instruction pipeline operates?
3. What is the efficiency of a linear pipeline?
4. State some applications of pipelining.
5. What are the disadvantages of pipeline architecture?
6. What is the basis of RISC pipelines?
7. Define vector processing.
8. What is the use of linear array processors?

## NOTES

9. Define SIMD array processor.
10. What are masking instructions?
11. How do you explain multiprocessors?
12. Define cache coherence.

### Long-Answer Questions

1. Explain the importance of space-time diagram for pipelining.
2. Describe the working of arithmetic pipeline with the help of an example.
3. Write a note on instruction pipeline.
4. Explain the basic structure of linear pipeline.
5. Explain the architecture of RISC pipelines.
6. Describe the characteristics of vector processing.
7. Discuss the array processing specifications with the help of examples.
8. Explain the process of multiprocessors in detail.
9. What do you mean by interprocess arbitration? Explain in detail.
10. Write an explanatory note on interprocessor communication and synchronization.
11. Describe the problems of critical section in detail.

---

## 5.10 FURTHER READING

---

- Tanenbaum, Andrew S. 1999. *Structured Computer Organization*, 4th Edition. New Jersey: Prentice-Hall Inc.
- Mano, M. Morris. 1993. *Computer System Architecture*, 3rd Edition. New Jersey: Prentice-Hall Inc.
- Bartee, Thomas C. 1985. *Digital Computer Fundamentals*. New York: McGraw-Hill.
- Mano, M. Morris. 1979. *Digital Logic and Computer Design*. New Delhi: Prentice-Hall of India.
- Leach, Donald P. and Albert Paul Malvino. 1994. *Digital Principles and Applications*. New York: McGraw-Hill.
- Mano, M. Morris. 2002. *Digital Design*. New Delhi: Prentice-Hall of India.
- Kumar, A. Anand. 2003. *Fundamentals of Digital Circuits*. New Delhi: Prentice-Hall of India.
- Stallings, William. 2007. *Computer Organisation and Architecture*. New Delhi: Prentice-Hall of India.

## NOTES

---

