

COMPUTER ORGANIZATION AND ARCHITECTURE

MCA

First Semester

Paper III



Institute of Distance and Open Learning
University of Mumbai

Dr. Rajan Welukar

Vice Chancellor
University of Mumbai
Fort, Mumbai - 400 032

Prin. Dr. Naresh Chandra

Pro-Vice Chancellor
University of Mumbai
Fort, Mumbai - 400 032

Dr. Dhaneswar Harichandan

Professor-cum-Director
Institute of Distance and Open Learning
University of Mumbai, Mumbai - 400 098

Authors

Shamim Akhtar: Units (1.0-1.5.2, 1.7-1.19) © Shamim Akhtar, 2013

Vivek Kesari: Units (1.6, 2.2.1, 2.2.3, 2.3-2.3.1, 2.5-2.5.2, 2.5.5, 3.3.1, 3.4.2, 3.4.4, 3.5, 5.2.1-5.2.3) © Vivek Kesari, 2013

Deepti Mehrotra: Units (2.2.2, 2.3.4, 2.4, 2.5.4, 2.5.6, 4.2.3-4.2.5) © Deepti Mehrotra, 2013

Vikas Publishing House: Units (2.0-2.2, 2.3.2-2.3.3, 2.3.5-2.3.6, 2.4.1-2.4.3, 2.5.3, 2.6-2.11, 3.0-3.3, 3.3.2, 3.4-3.4.1, 3.4.3, 3.6-3.11, 4.0-4.2.2, 4.3-4.8, 5.0-5.2, 5.3-5.9) © Reserved, 2013

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 Publisher.

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 Publisher, 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.



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: 576, Masjid Road, Jangpura, New Delhi 110 014

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

SYLLABUS-BOOK MAPPING TABLE

Computer Organization and Architecture

Syllabus

Unit 1 – Digital Logic

Boolean Algebra; Gates; Combinational Circuits: Implementation of Boolean Functions (Algebraic Simplification, Karnaugh Maps), Multiplexers/Demultiplexers, Decodes/Encodes, Adders (Half, Full); Sequential Circuits: Flip-Flops (S-R, J-K, D), Registers (Parallel, Shift), Counters (Ripple, Synchronous).

Unit 2 – The Computer System

Computer Function and Interconnection: Computer Functions, Interconnection Structures, Bus Interconnection; Memory System Design: Memory Hierarchy and SRAM, Advanced DRAM Organization, Interleaved Memory, Associative Memory, Nonvolatile Memory, RAID; Cache Memory: Cache Memory Principles, Elements of Cache Design, Improving Cache Performance; Input/Output: External Devices, I/O Modules, Programmed I/O, Interrupt-Driven I/O, Direct Memory Access; I/O Channels and Processors.

Unit 3 – Central Processing Unit

Instruction Set - Characteristics and Function: Machine Instruction Characteristics, Type of Operands, Types of Operations; Instruction Set - Addressing Modes and Formats: Addressing, Instruction Formats; CPU Structure and Function: Processor Organization, Register Organization, Instruction Cycle, Instruction Pipelining; RISC; Instruction Level Parallelism and Superscalar Processors: Superscalar versus Super Pipelined, Limitations, Instruction Level Parallelism and Machine Parallelism, Instruction Issue Policy, Register Renaming, Branch Prediction, Superscalar Execution, Superscalar Implementation.

Unit 4 – Control Unit

Control Unit Operation: Micro-Operation, Control of the Processor, Hardwired Implementation; Microprogrammed Control (Basic Concepts).

Unit 5 – Parallel Organization

Microprocessor Organizations: Types of Parallel Processor Systems, Parallel Organization; Symmetric Multiprocessors: Organization; Clusters: Cluster Configurations, Cluster Computer Architecture.

<p>2.3.5 Non-Volatile Memory</p> <p>2.3.6 Redundant Array of Independent Disks or RAID</p> <p>2.4 Cache Memory</p> <ul style="list-style-type: none"> 2.4.1 Cache Memory Principles 2.4.2 Elements of Cache Design 2.4.3 Improving Cache Performance <p>2.5 Input/Output</p> <ul style="list-style-type: none"> 2.5.1 Peripheral Devices 2.5.2 External Devices 2.5.3 I/O Modules 2.5.4 Programmed I/O 2.5.5 Interrupt-Driven I/O 2.5.6 Direct Memory Access <p>2.6 I/O Channels and Processors</p> <p>2.7 Summary</p> <p>2.8 Key Terms</p> <p>2.9 Answers to 'Check Your Progress'</p> <p>2.10 Questions and Exercises</p> <p>2.11 Further Reading</p>	UNIT 1 DIGITAL LOGIC <ul style="list-style-type: none"> 1.1 Introduction 1.2 Unit Operations 1.3 Logic Gates 1.4 Processor Address 1.5 Implementation of Boolean Functions 1.6 Logic Circuits 1.7 Combinational Circuits 1.8 Address 1.9 Multiplexer 1.10 Demultiplexer 1.11 MUX-Decoder 1.12 D-Mux-Decoder 1.13 D-R-Mux 1.14 Address Generation 1.15 Registers 1.16 Encoder 1.17 D-Encoder 1.18 Decoder 1.19 Shift Registers 1.20 Ripple Counter 1.21 JK-Flip-Flop 1.22 SR-Flip-Flop 1.23 T-Flip-Flop 1.24 HK-Flip-Flop 1.25 DHK-Flip-Flop 1.26 Asynchronous Flip-Flop 1.27 Registers 1.28 Latch 1.29 Gated Latch 1.30 Key Latch 1.31 Answers to Check Your Progress 1.32 Questions and Exercises 1.33 Register Read/write 1.34 Gated Registers 1.35 Shift Registers 1.36 Ripple Counter 1.37 JK-Flip-Flop 1.38 SR-Flip-Flop 1.39 T-Flip-Flop 1.40 HK-Flip-Flop 1.41 Address Generation 1.42 Demultiplexer 1.43 MUX-Decoder 1.44 D-Mux-Decoder 1.45 D-R-Mux 1.46 Address 1.47 Multiplexer 1.48 Encoder 1.49 Decoder 1.50 JK-Flip-Flop 1.51 SR-Flip-Flop 1.52 T-Flip-Flop 1.53 HK-Flip-Flop 1.54 DHK-Flip-Flop 1.55 Asynchronous Flip-Flop 1.56 Registers 1.57 Latch 1.58 Gated Latch 1.59 Key Latch 1.60 Answers to Check Your Progress 1.61 Questions and Exercises 1.62 Register Read/write 1.63 Gated Registers 1.64 Shift Registers 1.65 Ripple Counter 1.66 JK-Flip-Flop 1.67 SR-Flip-Flop 1.68 T-Flip-Flop 1.69 HK-Flip-Flop 1.70 Address Generation 1.71 Demultiplexer 1.72 MUX-Decoder 1.73 D-Mux-Decoder 1.74 D-R-Mux 1.75 Address 1.76 Multiplexer 1.77 Encoder 1.78 Decoder 1.79 JK-Flip-Flop 1.80 SR-Flip-Flop 1.81 T-Flip-Flop 1.82 HK-Flip-Flop 1.83 DHK-Flip-Flop 1.84 Asynchronous Flip-Flop 1.85 Registers 1.86 Latch 1.87 Gated Latch 1.88 Key Latch 1.89 Answers to Check Your Progress 1.90 Questions and Exercises 1.91 Register Read/write 1.92 Gated Registers 1.93 Shift Registers 1.94 Ripple Counter 1.95 JK-Flip-Flop 1.96 SR-Flip-Flop 1.97 T-Flip-Flop 1.98 HK-Flip-Flop 1.99 DHK-Flip-Flop 1.100 Asynchronous Flip-Flop
UNIT 3 CENTRAL PROCESSING UNIT	
<p>3.0 Introduction</p> <p>3.1 Unit Objectives</p> <p>3.2 Instruction Set</p> <ul style="list-style-type: none"> 3.2.1 Characteristics and Functions 3.2.2 Types of Operands 3.2.3 Types of Operations <p>3.3 Instruction Set : Addressing Modes and Formats - An Introduction</p> <ul style="list-style-type: none"> 3.3.1 Addressing Modes 3.3.2 Instruction Formats <p>3.4 CPU Structure and Function</p> <ul style="list-style-type: none"> 3.4.1 Processor Organization 3.4.2 Register Organization 3.4.3 Instruction Cycle 3.4.4 Instruction Pipelining <p>3.5 Reduced Instruction Set Computing or RISC</p> <p>3.6 Instruction Level Parallelism and Superscalar Processors</p> <ul style="list-style-type: none"> 3.6.1 Superscalar versus Superpipelined 3.6.2 Limitations 3.6.3 Instruction Issue Policy 3.6.4 Superscalar Execution 3.6.5 Superscalar Implementation 3.6.6 Instruction Level Parallelism and Machine Parallelism <p>3.7 Summary</p> <p>3.8 Key Terms</p> <p>3.9 Answers to 'Check Your Progress'</p> <p>3.10 Questions and Exercises</p> <p>3.11 Further Reading</p>	161-223
UNIT 4 CONTROL UNIT	
<p>4.0 Introduction</p> <p>4.1 Unit Objectives</p> <p>4.2 Control Unit Operation: Basic Concepts</p> <ul style="list-style-type: none"> 4.2.1 Functions of Control Unit 4.2.2 Control of the Processor 	225-266

- 4.2.3 Hardwired Implementation
- 4.2.4 Microprogrammed Control
- 4.2.5 Microinstructions
- 4.3 Microoperation
 - 4.3.1 Data Manipulation Instructions
- 4.4 Summary
- 4.5 Key Terms
- 4.6 Answers to 'Check Your Progress'
- 4.7 Questions and Exercises
- 4.8 Further Reading

UNIT 5 PARALLEL ORGANIZATION

267-306

- 5.0 Introduction
- 5.1 Unit Objectives
- 5.2 Microprocessor Organization
 - 5.2.1 Parallel Organization: Basics
 - 5.2.2 Types of Parallel Processor Systems
 - 5.2.3 Flynn's Classification of Computers
- 5.3 Symmetric Multiprocessors
 - 5.3.1 Organization of Symmetric Multiprocessing
- 5.4 Clusters
 - 5.4.1 Cluster Configurations
 - 5.4.2 Cluster Computer Architecture
- 5.5 Summary
- 5.6 Key Terms
- 5.7 Answers to 'Check Your Progress'
- 5.8 Questions and Exercises
- 5.9 Further Reading

INTRODUCTION

In computer science and engineering, computer architecture refers to specification of the relationship between different hardware components of a computer system. It may also refer to the practical art of defining the structure and relationship of the sub-components of a computer system, while computer organization helps to optimize performance based products. For example, software engineers need to know the processing ability of processors. They may need to optimize software in order to gain the most performance at the least expense. This can require quite detailed analysis of the computer organization. Computer organization also helps plan the selection of a processor for a particular project.

Computer architecture comprises the visible attributes of a system whereas computer organization refers to the attributes that directly affect the execution of a program. In fact, it would be appropriate to say that computer organization comprises the operational units and the way they are connected to each other and help to meet the specifications of the architecture. The instruction set, the number of bits to represent various types of data, the input-output systems and the memory addressing methods are all aspects of architecture. Computer models may exist with the similar architecture but different organization. The architecture may last for years but its organization will definitely change with time and advances in technology.

The performance of modern computer architecture can be described in MIPS per MHz, i.e., Millions of Instructions Per Second or Per Millions of Cycles Per Second of Clock Speed. This metric explicitly measures the efficiency of the architecture at any clock speed. A faster clock can make a faster computer. Historically, many people measured the speed by the clock rate usually in MHz (Mega Hertz) or GHz (Giga Hertz). This refers to the cycles per second of the main clock of the CPU. Other factors that influence the computer speed are its functional units, bus speeds, available memory and the type and order of instructions in the programs being run. In a typical home computer, the simplest and most reliable way to speed performance is usually to add Random Access Memory (RAM).

This book, *Computer Organization and Architecture*, is aimed at providing the readers with knowledge on computer systems and their functional units, the central processing unit, the memory hierarchy of computers as well as the concepts of microprocessor organization, computer cluster architecture and organization. It discusses stack organization, instruction formats, instruction classifications and addressing modes in detail. It also explains the memory hierarchy giving details of the main memory, auxiliary memory, cache memory, virtual memory as well as the mapping process. The content of the book is expected to help in the understanding of the computer architecture and its organization which would enable readers to use it in understanding the basics. The book follows the self-instruction format wherein each unit begins with an 'Introduction' to the topic of the unit followed by an outline of the 'Unit Objectives'. The detailed content is then presented in a simple and structured form interspersed with 'Check Your Progress' questions to facilitate a better understanding of the topics discussed. The 'Key Terms' help the student revise what he/she has learnt. A 'Summary' along with a set of 'Questions and Exercises' is also provided at the end of each unit for effective recapitulation.

NOTES

UNIT 1 DIGITAL LOGIC

Structure

- 1.0 Introduction
- 1.1 Unit Objectives
- 1.2 Logic Gates
- 1.3 Boolean Algebra
- 1.4 Implementation of Boolean Functions
- 1.5 Logic Circuits
 - 1.5.1 Combinational Circuits
 - 1.5.2 Sequential Circuits
- 1.6 Adders
 - 1.6.1 Half Adder
 - 1.6.2 Full Adder
- 1.7 Boolean Simplification
 - 1.7.1 Algebraic Simplification
 - 1.7.2 Karnaugh Maps
- 1.8 Decoder
- 1.9 Encoder
- 1.10 Demultiplexers
- 1.11 Multiplexers
- 1.12 Flip-Flop
 - 1.12.1 D Flip-Flop
 - 1.12.2 J-K Flip-Flop
 - 1.12.3 T Flip-Flop
 - 1.12.4 S-R Flip-Flop
 - 1.12.5 Asynchronous Flip-Flop
- 1.13 Registers
- 1.13.1 Parallel Registers
- 1.13.2 Shift Registers
- 1.14 Counters
 - 1.14.1 Ripple Counters
 - 1.14.2 Synchronous Counters
- 1.15 Summary
- 1.16 Key Terms
- 1.17 Answers to 'Check Your Progress'
- 1.18 Questions and Exercises
- 1.19 Further Reading

NOTES

1.0 INTRODUCTION

A computer is all-purpose device that can be uniquely programmed for carrying out a finite set of arithmetic or logical operations. In computing, a sequence of operations can be changed as per the requirement, hence the computer is capable of solving more than one type of problem. Conventionally, a computer consists of one processing unit, typically a Central Processing Unit (CPU), the various types of memory and peripheral devices. The processing unit carries out arithmetic and logic operations, and a sequencing and control unit that can change the order of operations based on

NOTES

stored information. Peripheral devices permit the information to be retrieved from an external source and to display the result of operations that are saved or retrieved. A general purpose computer has four main components: the Arithmetic Logic Unit (ALU), the Control Unit (CU), the memory and the Input and Output devices which are collectively termed as I/O devices. All these components are interconnected by busses that are made of groups of wires. The ALU is capable of performing two categories of operations: arithmetic and logic. The set of arithmetic operations that a particular ALU supports includes addition, subtraction, multiplication, division, trigonometry functions and square roots. An ALU may also compare numbers and return Boolean truth values, ‘True’ or ‘False’. Logic operations involve Boolean logic: AND, OR, XOR and NOT. These are used to create complicated conditional statements and process Boolean logic. In this unit, you will learn about the basic concepts of digital logic.

Boolean algebra was developed in 1854 by George Boole. Typically, Boolean algebra is the algebra of truth values 0 and 1 or equivalently of subsets of a given set. Characteristically, the Boolean algebra deals with the values ‘0’ and ‘1’ which are considered as two integers or as the truth values ‘False’ and ‘True’, respectively. In either case they are called bits or binary digits. A logic gate is a physical device that implements a Boolean function to perform a logical operation on one or more logic inputs and to produce a single logic output. Logic circuits include devices, such as multiplexers, registers, ALUs and computer memory through comprehensive microprocessors which may contain more than 100 million logic gates. Logic gates can also be used for storing data in digital form. A storage component can be constructed by connecting several gates in a ‘latch’ circuit. The complicated designs of storage components that use clock signals are called ‘flip-flops’. The combination of multiple flip-flops in parallel that is specifically used to store a multiple-bit value is termed as a ‘register’.

In digital electronics, an adder is referred as a digital circuit used to perform addition of numbers. Although adders can be constructed for many numerical representations but the most common adders operate on binary numbers. A multiplexer or mux is a device that selects one of several analog or digital input signals and forwards the selected input into a single line. Typically, multiplexers are used to increase the amount of data that can be sent over the network within a certain amount of time and bandwidth. On the other hand, a demultiplexer or demux is a device that takes a single input signal and selects one of many data-output-lines, which is connected to the single input. Finally, in this unit you will learn about flip-flops, registers and counters. A flip-flop or latch is a digital circuit that has two stable states and can be used for storing information. The circuit can be made to change state by signals applied to one or more control inputs and will have one or two outputs. It is the basic storage element in sequential logic. Flip-flops can be either simple (transparent or asynchronous) or clocked (synchronous). In digital circuits, a shift register is a cascade of flip-flops which shares the same clock in which the output of each flip-flop is connected to the data input of the next flip-flop in the chain. Shift registers can have both parallel and serial inputs and outputs.

1.1 UNIT OBJECTIVES

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

- Understand the significance of logic gates
- Explain the rules and usage of Boolean algebra
- Discuss the process of implementation of Boolean functions
- Describe logic circuits
- Understand the functions of adders, half adder and full adder
- Explain the significance of function simplification and Karnaugh maps
- Describe demultiplexers and multiplexers
- Explain flip-flop and its various types
- Discuss registers and their various types
- Understand the functions of counters

NOTES

- A digital computer uses binary number system for its operation. The computer receives, stores, understands and manipulates information composed of only 0s and 1s. The manipulation of binary information is done by logic circuits known as logic gates.
- NOT Gate**
- The basic NOT gate has only one input and one output. The output is always the opposite or negation of the input. Table 1.1 is the truth table for NOT gate.

Table 1.1 Truth Table for NOT Gate

A	F
0	1
1	0

Symbol: $F = A'$

Figure 1.1 illustrates NOT gate representation.



Fig. 1.1 NOT Gate

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:

NOTES

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

In Table 1.2, 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)

Figure 1.2 represents the AND gate.



Fig. 1.2 AND 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. Table 1.3 is the truth table that summarizes OR gate operations.

Table 1.3 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)

Figure 1.3 represents OR gate.



Fig. 1.3 OR Gate

Truth table for AND gate is as follows:
A AND B = 1 if both A and B are 1, otherwise 0.

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. Table 1.4 summarizes the truth table for the XOR gate.

Table 1.4 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)

Figure 1.4 represents XOR gate.

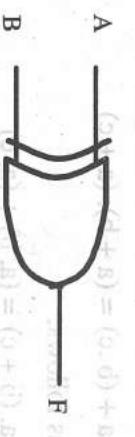


Fig. 1.4 XOR Gate

1.3 BOOLEAN ALGEBRA

Boolean algebra is named after George Boole, who used it to study human logical reasoning. For example, any event can be either true or false. To explain this logic, connectives are used that can be of any of the following three basic forms:

1. A OR B

2. A AND B

3. NOT A

Boolean algebra consists of a set of elements B, with two binary operations {+} and {.} and a unary operation {'}, such that the following axioms hold:

- The set B contains at least two distinct elements x and y.

- **Closure:** For every x, y in B,

- $x + y$

- $x \cdot y$

- $x' = y$

- **Commutative Laws:** For every x, y in B,

- $x + y = y + x$

- $x \cdot y = y \cdot x$

- **Associative Laws:** For every x, y, z in B,

- $(x + y) + z = x + (y + z) = x + y + z$

- $(x \cdot y) \cdot z = x \cdot (y \cdot z) = x \cdot y \cdot z$

NOTES

- Identities (0 and 1):
 - $0 + x = x + 0 = x$ for every x in B
 - $1 \cdot x = x \cdot 1 = x$ for every x in B

NOTES

- **Distributive Laws:** For every x, y, z in B ,
 - $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$
 - $x + (y \cdot z) = (x + y) \cdot (x + z)$

- **Complement:** For every x in B , there exists an element x' in B such that,
 - $x + x' = 1$
 - $x \cdot x' = 0$

Duality Principle: Every valid Boolean expression (equality) remains valid if the operators and identity elements are interchanged.

$$+ \leftrightarrow \cdot \\ 1 \leftrightarrow 0$$

For example, Given the expression,
 $a + (b \cdot c) = (a + b) \cdot (a + c)$

Its dual expression is as follows:

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$$

The advantage of this theorem is that if you prove one theorem, the other follows automatically.

For example, if $(x + y + z)' = x' \cdot y' \cdot z'$ is valid, then its dual is also valid:

$$(x \cdot y \cdot z)' = x' + y' + z'$$

Apart from the axioms/postulates, there are other useful theorems. These entire theorems are useful for reducing the expression.

1. Idempotency

$$(a) x + x = x \quad (b) x \cdot x = x$$

Proof of (a):

$$x + x = (x + x) \cdot 1 \quad (\text{Identity})$$

Boolean algebra postulate 3. M ATOM

Boolean algebra postulate 4. ANTI-DISTRIBUTIVE LAW: If x, y, z are elements of B , then $x + (y \cdot z) = (x + y) \cdot (x + z)$. This law is also known as De Morgan's law.

$$= x \quad (\text{Identity})$$

2. Null elements for '+' and '.' operators

$$(a) x + 1 = 1 \quad (b) x \cdot 0 = 0$$

3. Involution

$$(x')' = x$$

4. Absorption

$$(a) x + x \cdot y = x$$

5. Absorption (variant)

$$(a) x + x' \cdot y = x + y$$

$$(b) x \cdot (x' + y) = x \cdot y = x \cdot (y \cdot x)$$

- (a) $(x + y)' = x' \cdot y'$
 (b) $(x \cdot y)' = x' + y'$

7. Consensus
 (a) $x \cdot y + x' \cdot z + y \cdot z = x \cdot y + x' \cdot z$
 (b) $(x + y) \cdot (x' + z) \cdot (y + z) = (x + y) \cdot (x' + z)$

The set $B = \{0, 1\}$ and the logical operations OR, AND and NOT satisfy all the axioms of Boolean algebra.

A Boolean expression is an algebraic statement containing Boolean variables and operators. Theorems can be proved using the truth table method. They can also be proved by an algebraic manipulation using axioms/postulates or other basic theorems.

1.4 IMPLEMENTATION OF BOOLEAN FUNCTIONS

$A = f(x, y, z)$

A Boolean function is an expression formed with binary variables, the two binary operators OR and AND, the unary operator NOT, and the equal and parenthesis signs. The result is also a binary value. The general usage is ‘.’ for AND, ‘+’ for OR and ‘,’ for NOT.

Precedence of Operators

The brackets are used in writing Boolean expressions that further uses operator precedence rule for performing Boolean function. Precedence (highest to lowest):
 $\rightarrow : \rightarrow +$

For example,

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

In order to avoid confusion, use brackets to overwrite precedence.

Truth Table

A truth table is a mathematical table that is specifically used in logical operations including Boolean algebra, Boolean functions and propositional calculus. It consists of every possible combination of inputs and its corresponding outputs.

Inputs	Outputs
...	...
...	...
...	...

For basic logic gates, the truth tables are already discussed. Now, for the complex digital systems, it is very important to derive the truth table.

A truth table describes the behaviour of a system that is to be designed. This is the starting point for any digital system design. A designer must formulate the truth table first. Hence, it is the responsibility of the designer to decide the number of output bits to represent the behaviour of the system.

NOTES

For example, if you have to design a 2-bit multiplier which multiplies two inputs A and B, each of the two bits, then it should be noted that the output must be at least of 4 bits since the maximum result that you can have from this multiplication is 1001(9) corresponding to the maximum value of both the inputs, i.e., 11(3). The block diagram of 2-bit multiplier is shown in Figure 1.5.

NOTES

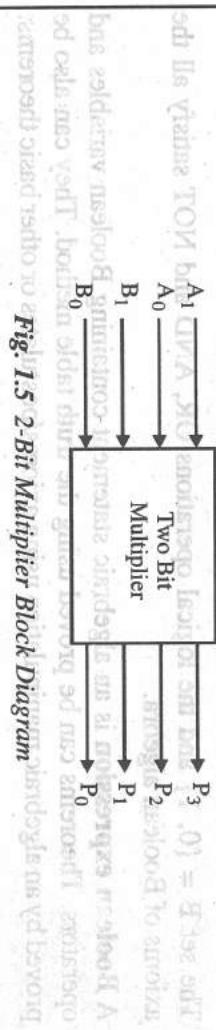


Fig. 1.5 2-Bit Multiplier Block Diagram

In the truth table formation, inputs are taken as A_1A_0 for A input and B_1B_0 for B input. Output resulting from multiplication is to be represented as $P_3P_2P_1P_0$ where P_3 is the Most Significant Bit (MSB) and P_0 is the Least Significant Bit (LSB) bit. If $A = 10$, i.e., 2 and $B = 11$, i.e., 3, then the result of multiplication will be 0110, i.e., 6. So, the bits at the output will be $P_3 = 0, P_2 = 1, P_1 = 1, P_0 = 0$. The complete truth table for the multiplier 2-bit is shown in Table 1.5.

Table 1.5 Truth Table for 2-Bit Multiplier

A_1	B_0	B_1	B_0	P_3	P_2	P_1	P_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	1	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	1	1	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	1	1	0	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	1	0	1	0
1	1	1	1	1	1	1	1

After the truth table, you have to write the Boolean expression for the output bit and then define the reduced expression using logic gates.

Whenever a Boolean expression for any output signal is to be written from the truth table, only those input combinations are written for which the output is high. For example, let us write the Boolean expression for Table 1.6.

To form the Boolean expression for Table 1.6, we have to consider the output

x	y	z	F ₁	F ₂	F ₃	F ₄
0	0	0	0	0	0	0
0	0	1	0	1	1	1
0	1	0	0	0	0	0
0	1	1	0	0	1	1
1	0	0	0	1	1	1
1	0	1	0	1	1	1
1	1	0	1	1	0	0
1	1	1	0	0	0	0

NOTES

The Boolean expression for the output F₁ will be F₁ = x.y.z'. This is in the Sum-of-Products form, which will be discussed later.

As can be seen from Table 1.6, output F₁ is 1 only when input xyz is 110. This is represented as x.y.z'. Similarly, you can write the output expression for the rest of the output signals.

F₂ can be reduced using Boolean algebra and can be written as follows:

$$\begin{aligned} F_2 &= x'.y'.z + x.y'.(z' + z) + x.y.(z' + z) \\ &= x'.y'.z + x.y' + x.y \\ &= x'.y'.z + x.(y' + y) \\ &= x'.y'.z + x \\ &= (x' + x).(y'.z + x) \end{aligned}$$

= 1. (y'.z + x)

Similarly, it can be shown that F₃ = F₄ = x.y' + x'.z

Complement of Functions

For a function F, the complement of this function F' is obtained by interchanging 1 with 0 and vice versa in the function's output values. As an example, take the following function F₁ and its complement, F'₁:

Table 1.7 Truth Table of Function and its Complement

x	y	z	F ₁	F' ₁
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	1	0
1	1	1	0	1

The same can also be verified using the Boolean algebra technique. In Table 1.7, if $F_1 = xyz'$, then its complement will be:

NOTES

$$\begin{array}{c}
 F' = (x, y, z)' \\
 = x' + y' + (z)' \text{ (Using De Morgan's law)} \\
 = x' + y' + z
 \end{array}$$

This is the same as that obtained from the truth table by algebraic manipulation which is given as follows:

$$\begin{aligned}
 F' &= x'y'z' + x'y'z + x'yz' + x'yz + xy'z' + xy'z + xyz \\
 &= x'y.(z' + z) + x'y.(z' + z) + x.y.(z' + z) + xyz \\
 &= x'y + x.(y' + y) + x.(y' + y.z) \\
 &= x' + x(y' + y). (y' + z) \\
 &= x' + x. (y' + z) \\
 &= (x' + x)(x' + y + z) \\
 &= (x' + y + z)
 \end{aligned}$$

The following are some more general forms of **De Morgan's theorems** specifically used for obtaining complement functions:

$$\begin{aligned}
 (A + B + C + \dots + Z)' &= A' \cdot B' \cdot C' \cdot \dots \cdot Z' \\
 (A \cdot B \cdot C \cdot \dots \cdot Z')' &= A' + B' + C' + \dots + Z'
 \end{aligned}$$

Standard Forms

There are several types of Boolean expressions that are used to represent Boolean functions. A Boolean expression in canonical form is unique. The following are two standard canonical forms for writing a Boolean expression:

- Sum-Of-Product (SOP)
- Product-Of-Sum (POS)

Before using SOP and POS forms, you must know the following terms:

- **Literal:** A variable on its own or in its complemented form is known as a literal.

Examples: x, x', y, y'

- **Product Term:** It is a single literal or a logical product (AND) of several literals.

Examples: $x, x.y.z', A'.B, A.B$

- **Sum Term:** It is a single literal or a logical sum (OR) of several literals.

Examples: $x, x+y+z', A'+B, A+B$

- **Sum-Of-Products (SOP) Expression:** It is a product term or a logical sum (OR) of several product terms.

Examples: $x, x+y.z', x.y'+x'.y.z, A.B+A'.B'$

- **Product-Of-Sum (POS) Expression:** It is a sum term or a logical product (AND) of several sum terms.

Examples: $x \cdot x \cdot (y + z)$, $(x + y') \cdot (x' + y + z)$, $(A+B) \cdot (A'+B')$

Every Boolean expression can either be expressed as a Sum-of-Product or Product-of-Sum expression. For example,

SOP:	$x' \cdot y + x \cdot y' + x \cdot y \cdot z$	0	0	1
POS:	$(x + y) \cdot (x' + y) \cdot (x' + z')$	0	1	1
Both:	$x' + y + z$ or $x \cdot y \cdot z$	0	1	1

Neither: $x \cdot (w' + y \cdot z)$ or $z' + w \cdot x' \cdot y + v \cdot (x \cdot z + w')$

Minterm and Maxterm

Consider two binary variables x, y . Each variable may appear as itself or in the complemented form as literals (i.e., x, x' and y, y'). For two variables, there are four possible combinations with the AND operator, namely:

$$x' \cdot y', x \cdot y, x \cdot y' \text{ and } x \cdot y$$

These product terms are called **Minterms**. In other words, a Minterm of n variables is the product of n literals from the different variables. In general, n variables can give 2^n Minterms.

Similarly, a **Maxterm** of n variables is the sum of n literals from the different variables.

$$(x + y + z), (x + y + z'), (x + y' + z), (x + y' + z'), (x + y + z'')$$

$$(x' + y + z), (x' + y + z'), (x' + y' + z), (x' + y' + z'), (x' + y + z'')$$

Examples: $x + y', x' + y, x + y', x + y$

In general, n variables can give 2^n Maxterms.

The Minterms and Maxterms of 2 variables are denoted by m_0 to m_3 and M_0 to M_3 , respectively. In Table 1.8, all the Minterms and Maxterms are illustrated along with their notations.

Table 1.8 Minterms and Maxterms

Minterms			Maxterms		
x	y	Term	Notation	Term	Notation
0	0	$x' \cdot y'$	m_0	$x + y$	M_0
0	1	$x' \cdot y$	m_1	$x + y'$	M_1
1	0	$x \cdot y'$	m_2	$x' + y$	M_2
1	1	$x \cdot y$	m_3	$x' + y'$	M_3

If you examine carefully, each Minterm is the complement of the corresponding Maxterm. For example, $m_2 = x \cdot y'$, and $m_2' = (x \cdot y)' = x' + (y')' = x' + y = M_2$. In other words, Minterm is the sum of terms of the corresponding Maxterm with its literal complemented.

Canonical Form: Sum of Minterms

Canonical form is a unique way of representing Boolean expressions. Any Boolean expression can be written in the form of the sum of Minterm. A Σ symbol is used for showing the sum of Minterms. Table 1.9 illustrates Minterms.

NOTES

x	y	z	$F_1 = \Sigma m(0)$	$F_2 = \Sigma m(1)$	$F_3 = \Sigma m(2)$
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	0	1	0	1	1
1	1	0	1	1	0
1	1	1	1	1	0

Sum-of-Minterms can be obtained by summing the Minterms of the function (where result is a 1) as follows:

$$F_1 = x \cdot y \cdot z' = \Sigma m(6)$$

$$F_2 = x' \cdot y \cdot z + x \cdot y' \cdot z' + x \cdot y \cdot z + x \cdot y \cdot z' + x \cdot y \cdot z = \Sigma m(1,4,5,6,7)$$

$$F_3 = x' \cdot y' \cdot z + x' \cdot y \cdot z + x \cdot y' \cdot z' + x \cdot y' \cdot z = \Sigma m(1,3,4,5)$$

Canonical Form: Product of Maxterms

Maxterms are sum terms. For Boolean functions, the Maxterms of a function are the terms for which the result is 0. Boolean functions can be expressed as Products-of-Maxterms. In Table 1.9, each output F_1 , F_2 and F_3 can be represented in Product-of-Maxterm. A Π symbol is used to represent Product-of-Maxterms.

$$F_1 = (x + y + z) \cdot (x + y + z') \cdot (x + y' + z) \cdot (x + y' + z') \cdot (x' + y + z) \\ \cdot (x' + y + z') \cdot (x' + y' + z)$$

$$= \Pi M(0,1,2,3,4,5,7)$$

$$\text{of } M_1 F_2 = (x + y + z) \cdot (x + y' + z) \cdot (x + y' + z')$$

$$\text{of } M_2 F_3 = \Pi M(0,2,3)$$

$$F_3 = (x + y + z) \cdot (x + y' + z) \cdot (x' + y + z) \cdot (x' + y' + z) \\ = \Pi M(0,2,6,7)$$

Conversion of Canonical Forms

Sum-of-Minterms \Rightarrow Product-of-Maxterms

- Rewrite Minterm shorthand using Maxterm shorthand.
- Replace Minterm indices with indices not already used.

For example, $F_1(x,y,z) = \Sigma m(6) = \Pi M(0,1,2,3,4,5,7)$

Product-of-Maxterms \Rightarrow Sum-of-Minterms

- Rewrite Maxterm shorthand using Minterm shorthand.
- Replace Maxterm indices with indices not already used.

For example, $F_2(x,y,z) = \Pi M(0,2,3) = \Sigma m(1,4,5,6,7)$

Sometimes, you are given the reduced expression for any Boolean expression. In this case, you need to find Minterms or Maxterms present in the expression. To convert from a general expression to a Minterm or Maxterm expression, you can use either the truth table or the algebraic manipulation method.

For example, suppose you wish to find all the Minterm expansions of $F = AB' + A'C$.

The truth table for the expression is represented in Table 1.10:

Table 1.10 Truth Table

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

From the Table 1.10,

$$F = A' \cdot B' \cdot C + A' \cdot B \cdot C + A \cdot B' \cdot C' + A \cdot B' \cdot C$$

$$= \Sigma m(1, 3, 4, 5)$$

Using Algebraic Manipulation

Use $X + X' = 1$ to introduce the missing variables in each term; this addition will not change the overall expression value. Therefore, for the Boolean expression $F = AB' + A'C$, the missing variable in the first term is C and in the second term is B. So, the missing variable can be introduced as follows:

$$\begin{aligned} &= A \cdot B' \cdot (C + C') + A' \cdot C \cdot (B + B') \\ &= A \cdot B' \cdot C + A \cdot B' \cdot C' + A' \cdot B \cdot C + A' \cdot B' \cdot C \\ &\equiv m_5 + m_4 + m_3 + m_1 \\ &= \Sigma m(1, 3, 4, 5) \end{aligned}$$

Similarly, you can find all the Maxterms for reduced expressions. Find the Maxterms expansion of $F = (A + B)(A' + C)$

Using Algebraic Expression: In this case, $XX' = 0$ is used to introduce missing variables in each term.

Therefore, $F = (A + B' + CC')(A' + C + BB')$

Assuming that $(A + B') = X$ and $C \cdot C' = YZ$, you can use the expression rule

$$\begin{aligned} &= X \cdot YZ = (X + Y)(X + Z) \\ &\text{using } F = (A + B' + C)(A + B' + C')(A' + B + C)(A' + B' + C) \\ &= \prod(2, 3, 4, 6) \end{aligned}$$

Using the Truth Table

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

Here,

$$F(A, B, C) = \prod(2, 3, 4, 6)$$

Check Your Progress

1. Write one characteristic of basic NOT gate.
2. What is a Boolean expression?
3. What is a Boolean function?
4. What does a truth table contain?
5. What is a literal?

1.5 LOGIC CIRCUITS

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

- Combinational Circuit
- Sequential Circuit

1.5.1 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 1.6.



Fig. 1.6 Block Diagram of a Combinational Circuit

Analysis of a Combinational Circuit

An analysis of the function of a combinational circuit is shown in Figure 1.7.

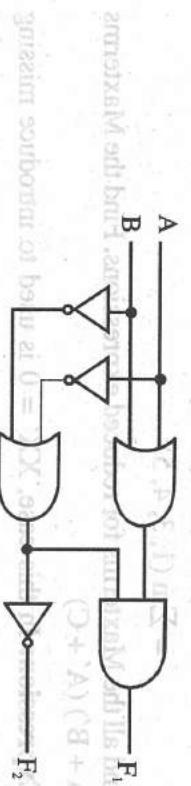


Fig. 1.7 A Simple Combinational Circuit

The steps for analysis are as follows:

Step 1: Label the inputs and outputs

Step 2: Obtain the functions of intermediate points and the outputs

Step 3: Draw the truth table

Step 4: Deduce the functionality of the circuit

Figure 1.8 illustrates a combinational circuit.

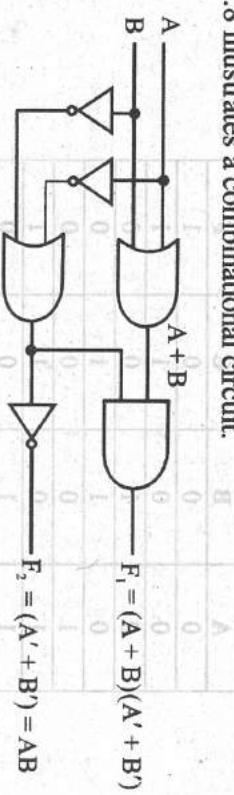


Fig. 1.8 A Combinational Circuit with Label

Then we form the truth table for the system. Table 1.11 summarizes the truth table for the circuit shown in Figure 1.8.

Table 1.11 Truth Table for the Circuit shown in Figure 1.8

A	B	(A + B)	(A' + B')	F ₁	F ₂
0	0	0	1	0	0
0	1	1	0	1	0
1	0	1	1	1	0
1	1	1	0	0	1

1.5.2 Sequential 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 1.9.

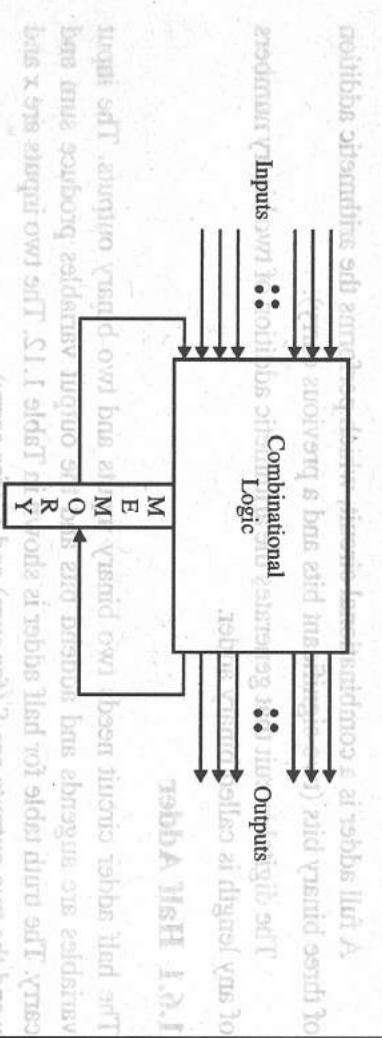


Fig. 1.9 Block Diagram of a Sequential Circuit

In combinational logic, the output depends only on the current inputs. However, in sequential logic, the output depends not only on the current inputs but also on the past input values. It needs some type of memory to remember the past input values.

A sequential circuit consists of a combinational logic and storage elements (refer Figure 1.10).

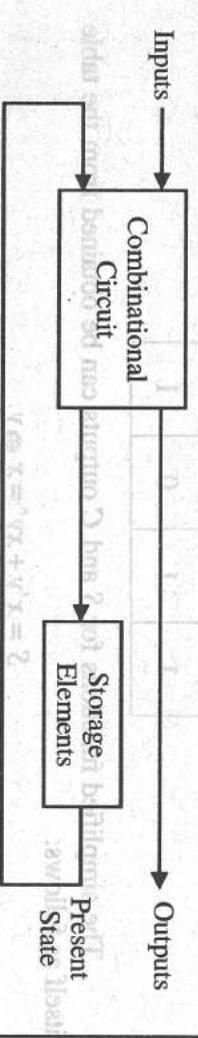


Fig. 1.10 Block Diagram of Sequential System

A sequential circuit can be classified into the following two types:

- Synchronous
- Asynchronous

NOTES

Synchronous Sequential Circuit: A system whose behaviour can be defined from the knowledge of its signal at a discrete instant of time. A practical synchronous sequential logic system uses fixed amplitudes, such as voltage levels for the binary signals. Synchronization is achieved by a timing device called master clock generator.

NOTES

Asynchronous Sequential Circuit: In this, the behaviour of sequential circuit depends on the order in which its input signals change and can be affected at any moment of time. In this type of circuit, generally a master clock generator is not required.

1.6 ADDERS

An adder is a combinational circuit, which performs the addition of two numbers to produce sum and carry as output.

A half adder is a combinational circuit, which performs the arithmetic addition of two binary bits.

A full adder is a combinational circuit, which performs the arithmetic addition of three binary bits (two significant bits and a previous carry).

The digital circuit that generates the arithmetic addition of two binary numbers of any length is called binary adder.

1.6.1 Half Adder

The half adder circuit needs two binary inputs and two binary outputs. The input variables are augends and addend bits and the output variables produce sum and carry. The truth table for half adder is shown in Table 1.12. The two inputs are x and y and the two outputs are S (for sum) and C (for carry).

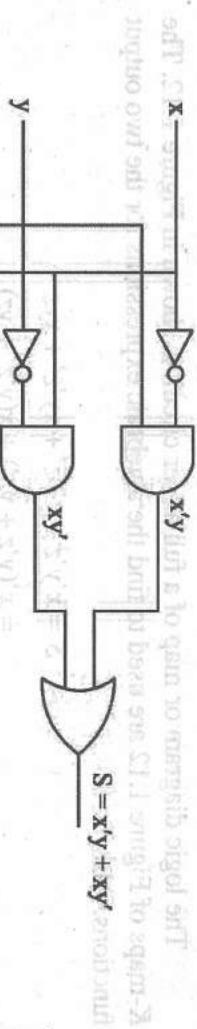
Table 1.12 Truth Table for Half Adder

x	y	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

The simplified functions for S and C outputs can be obtained from the table itself as follows:

$$S = x'y + xy' = x \oplus y$$

The expressions $S = x'y + xy'$ and $C = x \cdot y$ are in sum-of-products form. The logic circuit for this implementation is shown in Figure 1.11(a). Also, $S = x'y + xy'$ is equivalent to $S = x \cdot y$. The logic circuit for $S = x \cdot y$ and $C = xy$ is shown in Figure 1.11(b). The block diagram for half adder combinational circuit is shown in Figure 1.11(c).



NOTES

It consists of three inputs and two outputs. Two of the input variables, denoted by x and y , represent the two significant bits to be added. The third input, z , represents the carry from the previous lower significant position. Two outputs are denoted by symbols S (for sum) and C (for carry). The truth table for full adder is shown in Table 1.13.

Table 1.13 Truth Table for Full Adder

x	y	z	S	C
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

be described using Boolean expression or using Karnaugh map.

The functions S and C obtained from the truth table are as follows:

$$S = x'y'z + x'y'z' + xy'z + xyz$$

$$C = x'y + xy'z + xyz' + xyz$$

It is observed that the function $C = x'y + xy'z + xyz'$ is redundant since it is included in the function $S = x'y'z + x'y'z' + xy'z + xyz$.

The logic diagram or map of a full adder circuit is shown in Figure 1.12. The K-maps of Figure 1.12 are used to find the algebraic expressions for the two output functions. Also,

$$\begin{aligned}
 S &= x'y'z + x'yz' + xy'z' + xyz \\
 &= x'(y'z + yz') + x(y'z' + yz) \\
 &= x'(y \oplus z) + x(y \oplus z)' \\
 &= x'(y \oplus z) = x \oplus y \oplus z
 \end{aligned}$$

Thus, $S = x \oplus (y \oplus z)$ and $C = xy + yz + xz$.

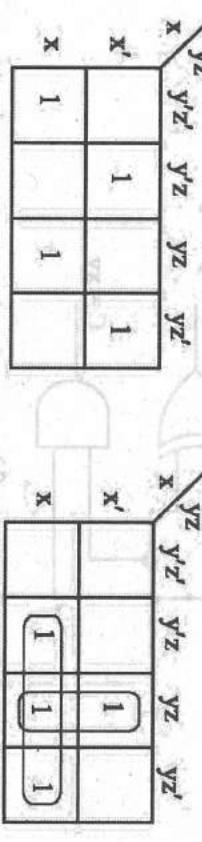


Fig. 1.12 Maps for Full Adder

Figure 1.13 illustrates the implementation of a full adder circuit.

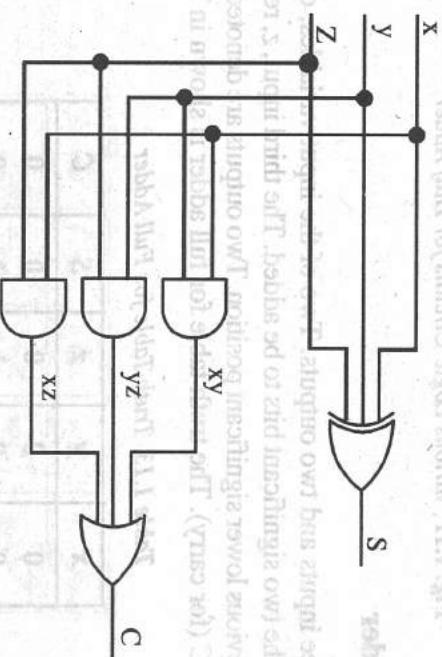


Fig. 1.13 Implementation of a Full Adder Circuit

Check Your Progress

6. Name the two broad categories of logic circuits.
7. What does a practical synchronous sequential logic system uses for the binary signals?
8. What is an adder?
9. What is a binary adder?

1.7 BOOLEAN SIMPLIFICATION

Boolean expressions can be manipulated into many forms. Hence some standardized forms are required for Boolean expressions to simplify statements of the expressions. Boolean algebra is typically used in the simplification of logic circuits. If a logic circuit function is translated into symbolic (Boolean) form and certain algebraic rules are applied to the resulting equation to reduce the number of terms and/or arithmetic operations, then the simplified equation may be translated back into circuit form for a

NOTES

logic circuit performing the same function with fewer components. There are several rules of Boolean algebra that can be used in reducing expressions to their simplest forms.

(any) basic logic gate = (gate) basic logic gate

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

$$F = (x+y)(x+y')(x'+z) \quad (6 \text{ literals})$$

using the theorems of Boolean algebra

$$= (x+x)(y+y') + 0 \cdot (x'+z) \quad (\text{Idempotency, Associative, } 0 \cdot x = 0)$$

$$= (x+x)(x'+z) \quad (\text{Complement})$$

$$= (x+x+0)(x'+z) \quad (\text{Identity 1})$$

$$= (x)(x'+z) \quad (\text{Idempotency, Identity 0})$$

$$= (x \cdot x' + x \cdot z) \quad (\text{Associative})$$

$$= (0 + x \cdot z) \quad (\text{Complement})$$

$$= x \cdot z \quad (\text{Identity 0})$$

Number of literals reduced from 6 to 2.

For example,

1. Finding the minimal SOP and POS expressions of :

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

$$= x'y \cdot z + x'y \cdot y' \cdot x + y'z \quad (\text{Distributive})$$

$$= x'y \cdot z + 0 + y'z \quad (\text{Complement, Null element 0})$$

$$= x'y \cdot z + y'z = \cancel{d's} + \cancel{d's} \quad (\text{Identity 0})$$

$$= x'z + y'z = \cancel{d's} + \cancel{d's} \quad (\text{Absorption})$$

$$= (x' + y')z \quad (\text{Distributive})$$

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:

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

$$= a.b.c + \underline{a.b.d} + a'.b.c' + c.d + \underline{b.d'} \quad (\text{Absorption on underlined terms})$$

$$= a.b.c + a.b + \underline{a.b.c'} + c.d + b.d' \quad (\text{Absorption on underlined terms})$$

$$= \underline{a.b.c} + \underline{a.b} + b.c' + c.d + b.d' \quad (\text{Absorption on underlined terms})$$

NOTES

Alternative 1

OR

$a \{$	0	b
	ab'	ab

m_0	m_1
m_2	m_3

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

Alternative 2

OR

$a \{$	0	b
	$a'b'$	ab'

m_0	m_2
m_1	m_3

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

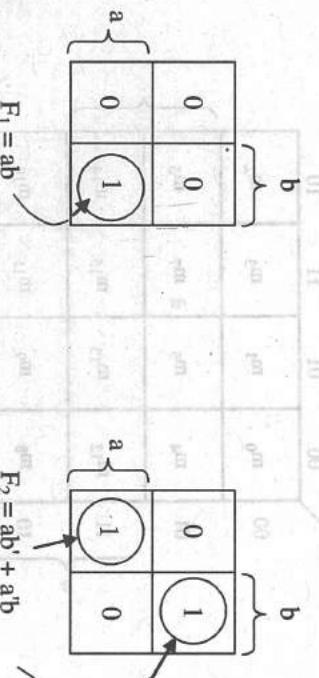
Equivalent Labelling



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 $F_1 = ab$ 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.

NOTES

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

NOTES

		bc	00	01	11	10
		a	$a'b'c'$	$a'b'c$	$a'bc$	$a'bc'$
a	0	0				
	1	1	$ab'c'$	$ab'c$	abc	abc'

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

		bc	00	01	11	10
		a	m_0	m_1	m_3	m_2
a	0	0				
	1	1	m_4	m_5	m_7	m_6

Also there is wrap-around in the K-map as follows:

- $a'b'c'$ (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:

		yz	00	01	11	10
		wx	m_0	m_1	m_3	m_2
w	0	00				
	1	01	m_4	m_5	m_7	m_6

		yz	00	01	11	10
		wx	m_8	m_9	m_{11}	m_{10}
w	0	10				
	1	11	m_{12}	m_{13}	m_{15}	m_{14}

Figure 1 is an example of Boolean expression $f = \overline{w} \cdot \overline{x} \cdot \overline{y} \cdot \overline{z} + \overline{w} \cdot \overline{x} \cdot \overline{y} \cdot z + w \cdot \overline{x} \cdot \overline{y} \cdot z + w \cdot x \cdot \overline{y} \cdot z$

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.

Steps for Forming Karnaugh Map

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_5 .

NOTES

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 1.1: Draw the K-map for the following function F:

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

Solution: Find all the Minterms.

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

Rearranging the terms with the MSB first and then the next bit up to the LSB, and removing repeated Minterms, we obtain:

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

The following is the K-map:

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

Example 1.2: The K-map of a 3-variable function F is as follows.

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

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

NOTES

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,

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

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:

$$A + A' = 1$$

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 1.3: Find the reduced expression for the function given as follows:

$$\begin{aligned} F(w, x, y, z) &= w'x'y'z' + w'x'y'z + w'x'y'z' + w'x'y'z \\ &= \Sigma m(4, 5, 10, 11, 14, 15) \end{aligned}$$

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

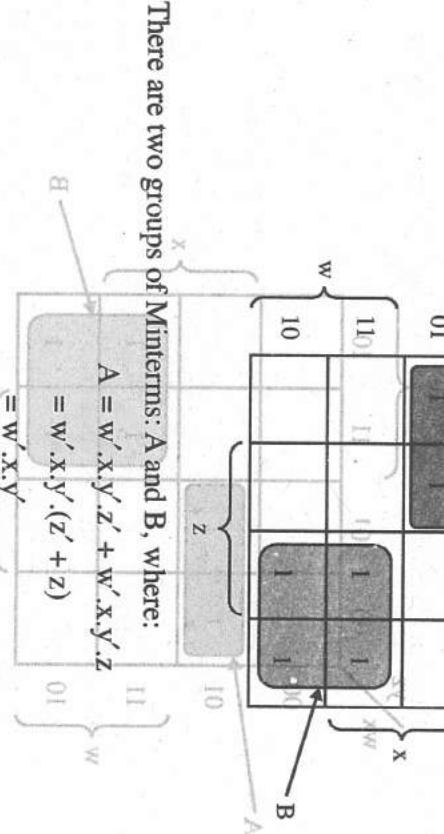
		yz		wx			
		00	01	10	11		
		w\y	0	1	0	1	
	w\y	00	01	10	11		
0	0	1	1	1	0	0	1
1	0	0	0	1	1	1	0
	w\y	00	01	10	11		

Example 1.3: Draw K-map of 4-variable function.

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

arrangement of minterms in K-map is same as arrangement of minterms in Boolean expression. In K-map, minterms are arranged in binary order. In Boolean expression, minterms are arranged in powers of 2's. So, the Boolean expression $w'x'y'z' + w'x'yz + wxy'z' + wxyz$ is equivalent to $w'x'y'z' + w'x'yz + wxy'z' + wxyz$.

NOTES



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

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

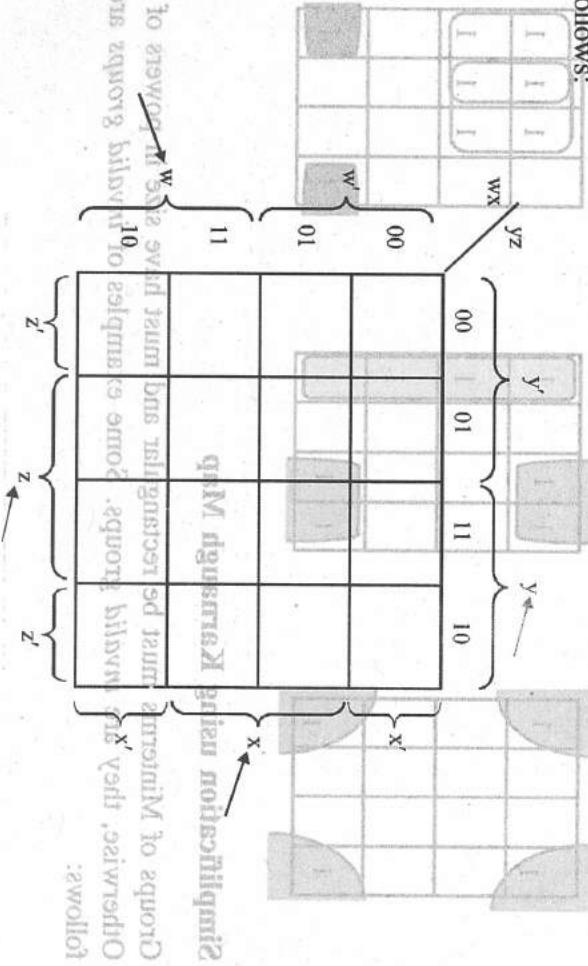
$$B = w'x'y'z' + w'x'y.z + w.x.y'z' + w.x.yz$$

The minterm $w'x'y'z'$ is common to both groups A and B. So, the Boolean expression $F(w,x,y,z) = A + B = w'x'y'z' + w'x'yz + wxy'z' + wxyz$ is equivalent to $w'x'y'z' + w'x'yz + wxy'z' + wxyz$.

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* terms, which represents all groups of the Minterms of the function.

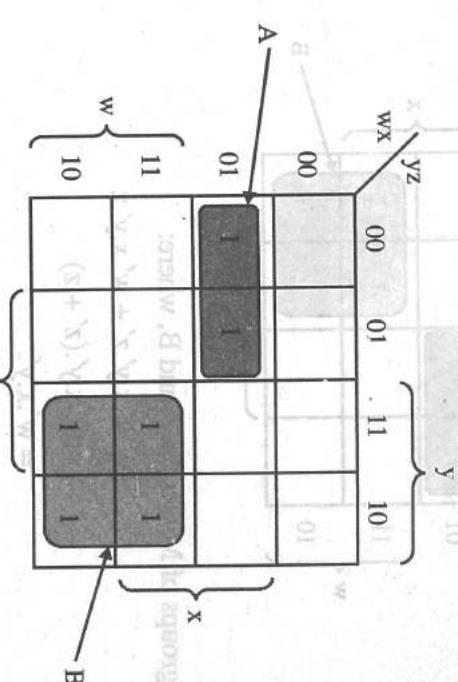
$F(w,x,y,z) = A + B = w'x'y'z' + w'x'yz + wxy'z' + wxyz$

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

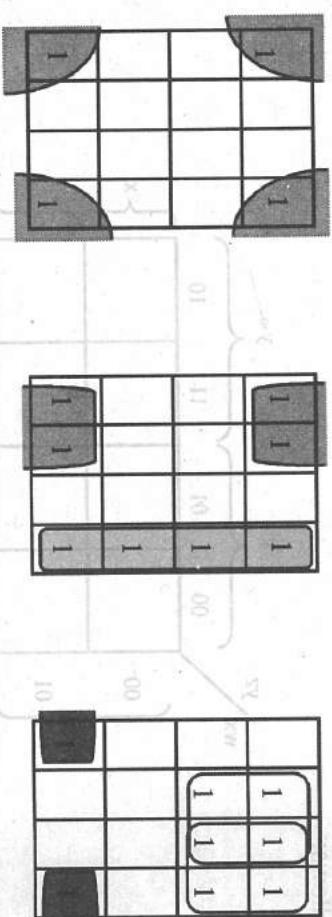


$$\Sigma w'y'z'w + \Sigma w'yz'w + \Sigma w'y'zw + \Sigma wz'w = \Sigma w'yz'w$$

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:

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

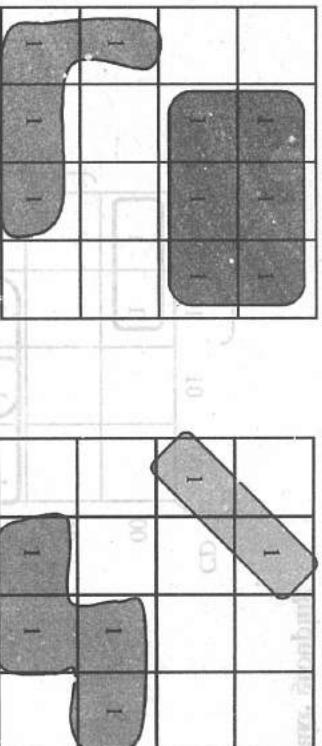
Larger groups correspond to product terms of fewer literals. In the case of a 4-variable K-map, if you have 1 cell, then you have 4 literals; if you have 2 cells, then 3 literals; if 4 cells, then 2 literals; if 8 cells, then 1 literal and at last, if 16 cells, then no literal. Also, some other possible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



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:

NOTES



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

Example 1.4: Find the grouping for the $F(A,B,C,D) = A(C+D)'(B'+D') + C(B+C'+A'D')$.

Solution: ~~restricting the grouping to rectangular regions~~ ~~using the minimum number of cells~~

Step 1: Find the SOP expression for the function.

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

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

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

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

$$\begin{aligned} &= AB'C'D' + ABC'D' + ABC + A'BC + A'CD \\ &= AB'C'D' + ABC'D' + ABC \cdot (D+D') + A'BC \cdot (D+D') \end{aligned}$$

After removing the repeated term $A'BCD$,

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

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

Step 3: Draw K-map entries.

		AB		CD			
		00	01	10	11		
		00					
		01					
		1					
			1				
				1			
					1		

*→ Low to quay là ai và
tất cả các minterm đều
đều phải có 2 minterm
kết nối với nhau*

Step 4: Make grouping.

NOTES			
CD	AB	A	
00	00	01	11
01	11	1	1
10	1	1	1
11	1	1	1
Using K-maps to find the SOP expression			
Step 1: Find the Minterms of the function			
Step 2: Use the power of 2 rule			
Step 3: Using the Bools rule to find the prime implicants			
Step 4: Using the area intersection concept to find the product terms			
$F(A, B, C, D) = A \cdot \bar{C} \cdot \bar{D} + B \cdot \bar{C} + A \cdot \bar{C} \cdot \bar{D}$			
$= A \cdot \bar{B} \cdot \bar{C} \cdot \bar{D} + A \cdot \bar{C} \cdot \bar{D} + B \cdot \bar{C} + C \cdot \bar{B} \cdot \bar{C} + A \cdot \bar{C} \cdot \bar{D}$			

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

$$F(A, B, C, D) = A \cdot \bar{C} \cdot \bar{D} + B \cdot \bar{C} + A \cdot \bar{C} \cdot \bar{D}$$

Simplest SOP Expressions

To find the simplest possible Sum-Of-Products (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:

After intersection			
After intersection			
Step 3: Draw K-map (Initial)	1	1	
	1	1	
		1	
			1

X

(Wrong)

After intersection			
After intersection			
Step 3: Draw K-map (Initial)	1	1	
	1	1	
		1	
			1

✓

(Correct)

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.

NOTES



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 1.5: Identify the prime implicants and the essential prime implicants of the following K-map:

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

Solution: Prime implicants are the covering of Minterms $\{m_0, m_1, m_3\}$, $\{m_1, m_2, m_3\}$ and $\{m_2, m_3\}$.

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 $\{m_1, m_2\}$ and $\{m_2, m_3\}$.

How to Find the Simplest SOP Expression?

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

Step 1: Circle all prime implicants on the K-map.

Step 2: Identify and select all essential prime implicants for the cover.

Step 3: 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 1.6: 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.

Step 2: Find minimum sum of products. After identifying the essential prime implicants, draw the remaining prime implicants to cover the remaining 1's.

NOTES

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

Step 2: Find prime implicants.

Essential Prime Implicants

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

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

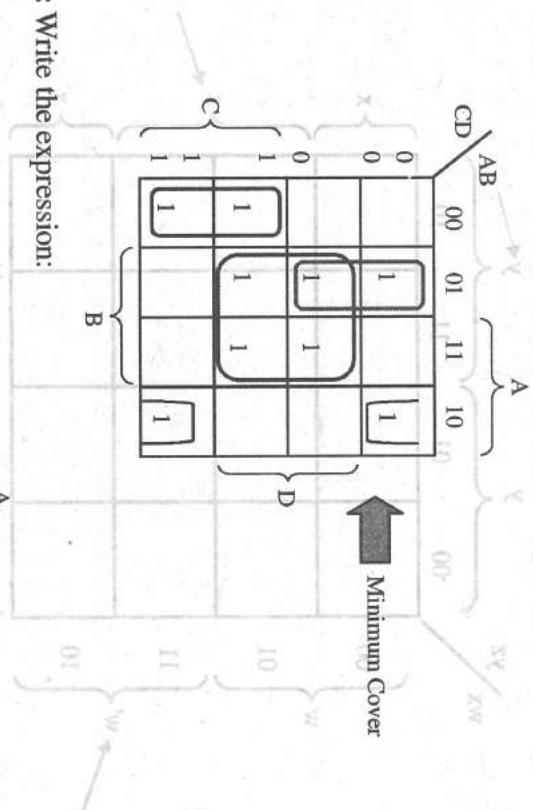
Some of the prime implicants are redundant. They can be removed after finding the essential prime implicant.

Step 3: Find essential prime implicants.

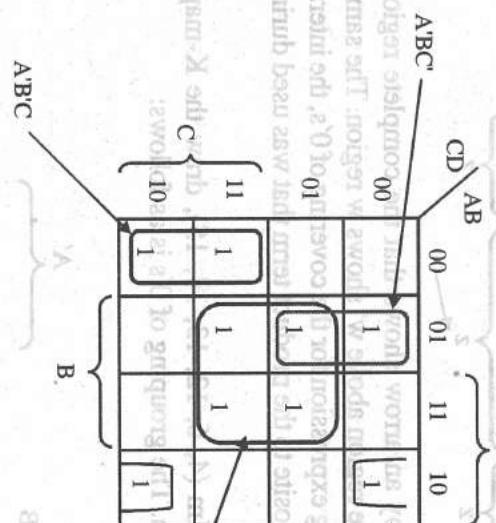
Essential Prime Implicants

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

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.

NOTES

Step 5: Write the expression:



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

Example 1.7: 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) $F = A' \cdot B \cdot C' + A' \cdot B \cdot D + A \cdot B' \cdot D' + A' \cdot B \cdot C$
- (ii) $F = A' \cdot B \cdot C' + A' \cdot C \cdot D + A \cdot B' \cdot D' + A' \cdot B \cdot C$

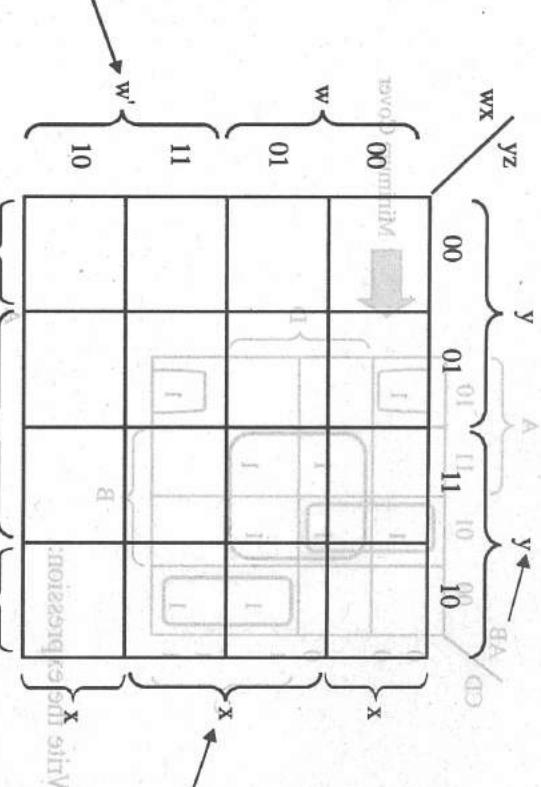
Example 1.8: $F(A,B,C,D) = A \cdot B \cdot C + B' \cdot C \cdot D' + A \cdot D + B' \cdot C' \cdot D'$, find the reduced expression using K-map.

Solution: $F(A,B,C,D) = A \cdot D + A \cdot C + B' \cdot D'$

Getting POS Expressions

Simplified POS expressions can be obtained by grouping the Maxterms (i.e., 0) 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:

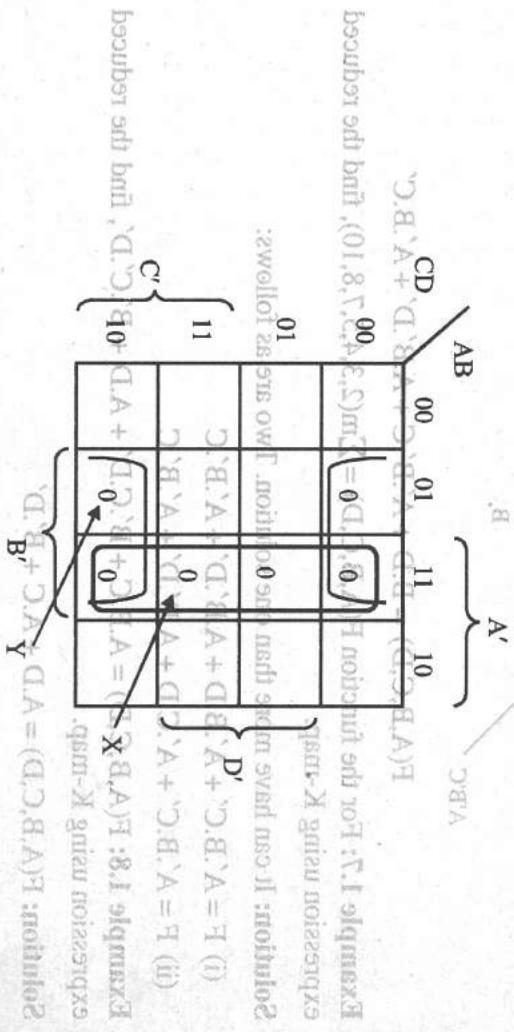
NOTES



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 1.9: 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.

will be as follows:

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

Digital Logic

Full adder is covered in Page 1.4.
A 2×4 decoder is shown in Page 1.4.

NOTES

AB	CD	00	01	11	10
		00	01	11	10
00	00	0	1	1	0
00	01	0	0	1	0
01	00	0	0	0	0
01	01	0	1	0	0
10	C0	0	0	0	0
10	C1	1	1	0	0

A 2×4 decoder is a circuit which takes two binary inputs and produces four binary outputs. The first three outputs are called F_0, F_1, F_2 and the fourth output is called F_3 . The inputs are A and B . The outputs are F_0, F_1, F_2, F_3 . The truth table for a 2×4 decoder is given below:

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' as follows:

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

Example 1.10: 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.

1.8 DECODER

Codes are frequently used to represent entities. For example, your name is a code to denote yourself (an entity). These codes can be identified (or decoded) using a decoder.

Convert binary information from n input lines to (Maximum of) 2^n output lines. Known as n -to- m line decoder or simply n : m or $n \times m$ decoder ($m \leq 2^n$), it may be used to generate 2^n (or fewer) minterms of n input variables. For example, if $n = 2$ and $m = 4$, then we have the 2×4 decoder (refer Figure 1.15).

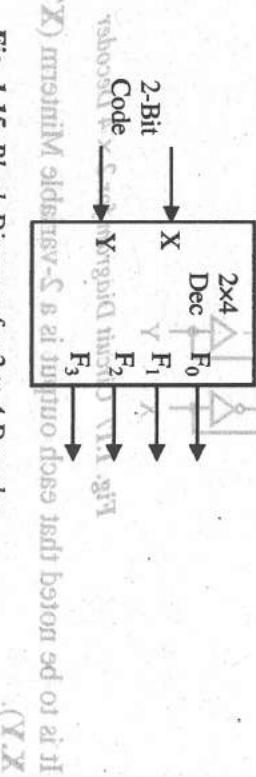


Fig. 1.15 Block Diagram for 2×4 Decoder

Further, if codes 00, 01, 10, 11 are used to identify four light bulbs, then we may use a 2×4 decoder as shown in Figure 1.16.



Fig. 1.16 2×4 Decoder used to Control Bulb Circuitry

Basic 2×4 Decoder

A 2×4 decoder is a circuit which has two input lines and generates 4 output lines which represent four codewords or Minterms. The truth table for the same is given in Table 1.14.

Table 1.14 Truth Table 2×4 Decoder

X	Y	F ₀	F ₁	F ₂	F ₃
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

2×4 decoder gives output high to only one output at a time. In Table 1.14, when $X = Y = 0$, $F_0 = 1$ and the rest of output lines are 0. The same can be written in Boolean expression as:

$$F_0 = X'Y'; F_1 = X'Y; F_2 = X.Y'; F_3 = X.Y$$

The circuit diagram for the same is given in Figure 1.17.

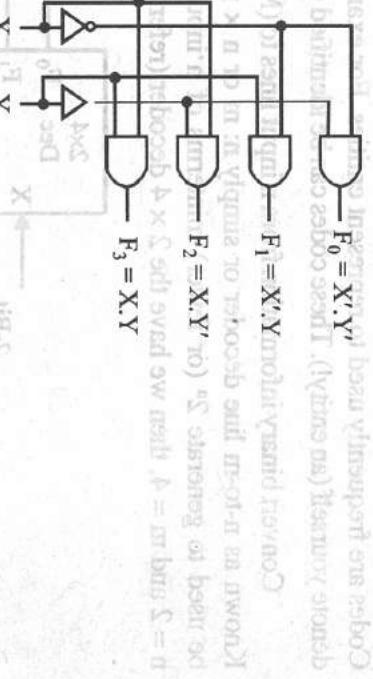


Fig. 1.17 Circuit Diagram for 2×4 Decoder

It is to be noted that each output is a 2-variable Minterm ($X'Y'$, $X'Y$, $X.Y'$, $X.Y$) or $X.Y$.

3 × 8 Decoder

It is a circuit, which has three input lines and generates 8 output lines to represent eight code words or Minterms. The truth table and the circuit diagram, for 3 × 8 decoder are given in Table 1.15 and Figure 1.18, respectively.

Table 1.15 Truth Table for 3 × 8 Decoder

x	y	z	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7
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 1.18 illustrates the circuit diagram for 3 × 8 decoder.

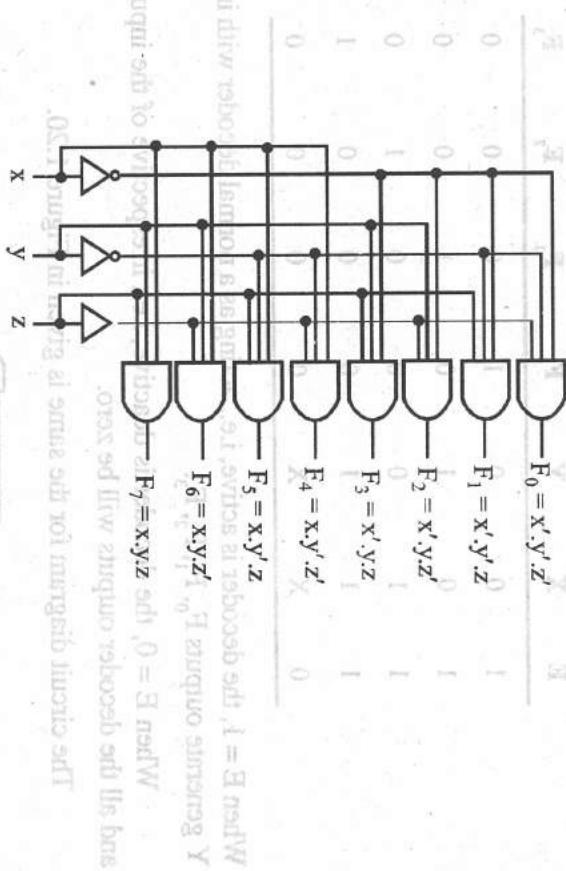


Fig. 1.18 Circuit Diagram for 3 × 8 Decoder

A decoder can be used to implement Boolean functions. A Boolean function represented in the Sum-of-Minterms form can use decoder outputs to generate the minterms and an OR gate to form the sum. Any combinational circuit with n inputs and m outputs can be implemented with an $n:2^n$ decoder with m OR gates. It is good when the circuit has many outputs and each function is expressed with few minterms.

Example 1.11: Realize a full adder sum and carry using 3 × 8 decoder.

Solution: We have $S(X, Y, Z) = \Sigma m(1, 2, 4, 7)$ and $C(X, Y, Z) = \Sigma m(3, 5, 6, 7)$. A 3×8 decoder generates all the possible Minterms and the Minterms can be given to

NOTES

the OR gate to find the Sum-of-Minterms for getting S and C. The same is given in Figure 1.19.

If it is a circuit, which has three inputs and three outputs, then we have to implement 8 \times 8 logic code words or Minterms. The truth table for the same is given in Table 1.19. The decoder uses enable signal E to select the appropriate minterm. The sum of Minterms is given by the OR gate.

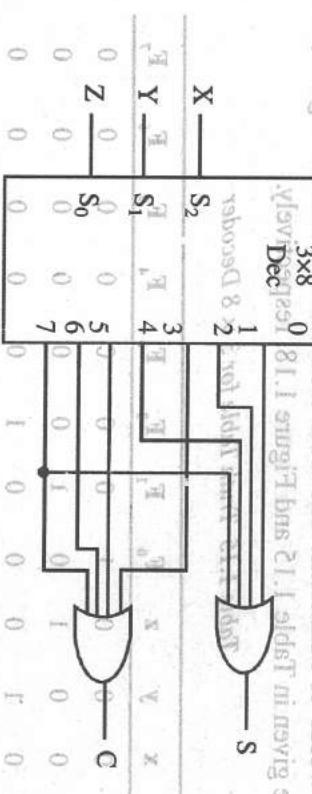


Fig. 1.19 Realization of a Full Adder using 3 \times 8 Decoder and OR Gate

Decoders with an Enable Signal

Decoders often come with an enable signal, so that the device is only activated when the enable E = 1. The truth table for the same is given in Table 1.16.

Table 1.16 Truth Table for 2 \times 4 Decoder with Enable E

E	X	Y	F ₀	F ₁	F ₂	F ₃
1	0	0	0	0	0	0
1	0	1	1	0	0	0
1	1	0	0	1	0	0
1	1	1	0	0	1	0
0	X	X	0	0	0	0

When E = 1, the decoder is active, i.e., acting as a normal decoder with inputs X and Y generate outputs F₀, F₁, F₂, F₃.

When E = 0, the decoder is inactive, i.e., irrespective of the inputs X and Y, and all the decoder outputs will be zero.

The circuit diagram for the same is given in Figure 1.20.

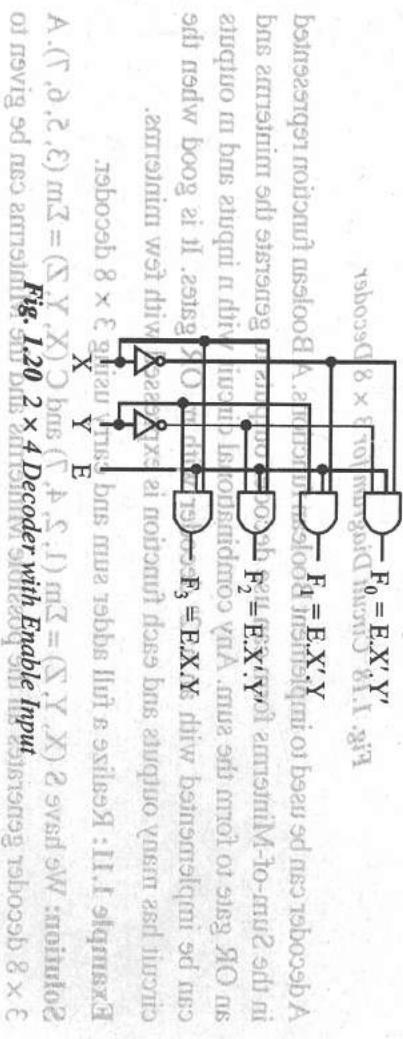


Fig. 1.20 2 \times 4 Decoder with Enable Input

The decoder has a one enable signal, that is, the decoder is enabled with $E = 1$. In most Medium Scale Integration or MSI decoders, enable signal is zero-enable usually denoted by E' (or E). The decoder is enabled when the signal is zero.

NOTES

Table 1.17 Truth Table of 2×4 Decoder with Enable E'

8×3 binary inputs of X to Y		E'	X	Y	F_0	F_1	F_2	F_3
0	0	0	1	0	0	0	0	0
0	0	1	0	1	1	0	0	0
0	1	0	0	0	0	1	0	0
0	1	1	1	0	0	0	1	0
1	0	0	0	1	1	0	0	0
1	0	1	1	0	0	0	0	1
1	1	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0

Larger Decoders

Larger decoders can be constructed from smaller ones. For example, a $3\text{-to}-8$ decoder can be constructed from two $2\text{-to}-4$ decoders with one enable as shown in Figure 1.21.

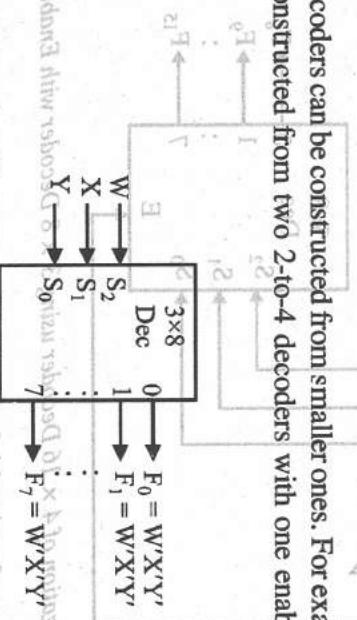


Fig. 1.21 Block Diagram of 3×8 Decoder

The above decoder can be realized using two 2×4 decoder with enable (refer Figure 1.22). The above decoder can be realized using two 2×4 decoder with enable (refer Figure 1.22).

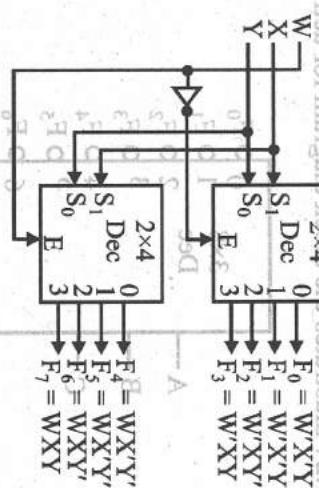


Fig. 1.22 Realization of 3×8 Decoder using 2×4 Decoder with Enable

Working: Since we have to realize 3×8 decoder with input line as WXY where W is the MSB and Y is the LSB. Giving the MSB bit W to enable of first 2×4 decoder will realize decoder outputs as Minterms from $X'Y'$, $X'Y$, XY' and XY . This can be treated as Minterms from $W \cdot X \cdot Y'$, $W \cdot X \cdot Y$, $W \cdot X \cdot Y'$ and $W \cdot X \cdot Y$, since $W = 0$ given enable input of the first decoder as high which makes it active. Now when $W = 1$, the first decoder is inactive and the second decoder will become

NOTES

Solution: Figure 1.23 represents the realization of 4×16 decoder using 3×8 decoder with enable.

active and realize decoder outputs as Minterms from $X'Y'$, $X'Y$, XY' and XY with $W = 1$. This can be treated as Minterms from $W\bar{X}'\bar{Y}'$, $W\bar{X}'Y$, $WX'\bar{Y}$ and $WX\bar{Y}$.

Example 1.12: Construct a 4×16 decoder from two 3×8 decoders with active high enable.

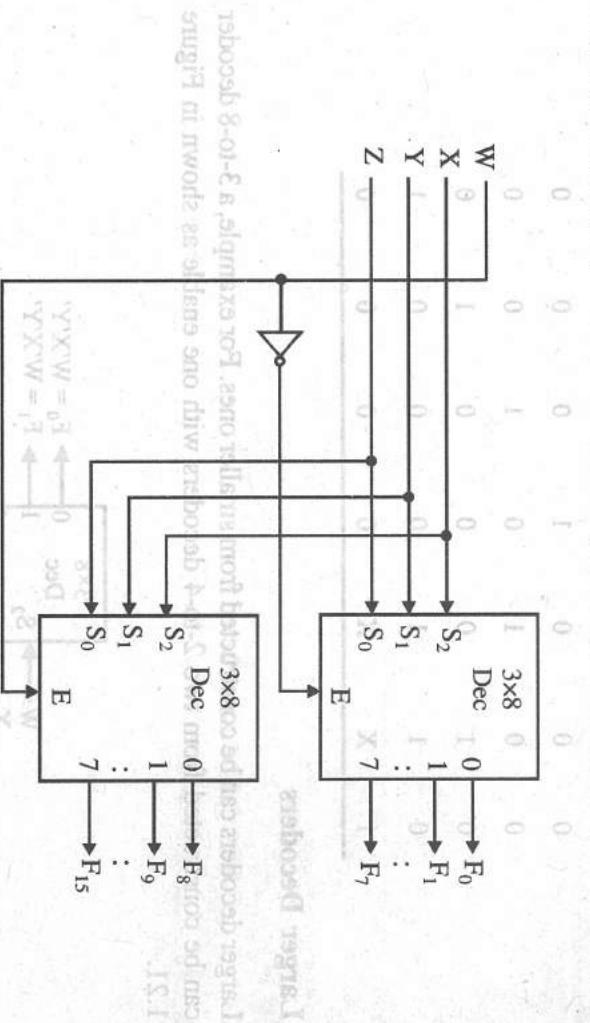


Fig. 1.23 Realization of 4×16 Decoder using 3×8 Decoder with Enable

A decoder can be of active high or active low type. The decoders discussed above are active high since output high represents the code. Sometime, we can have active low decoder where output low represents the code, only one output line is low corresponding to the desired input and the rest are high. Active low decoders are represented with bubble at each output of decoder. The same is being produced here for reference. Figure 1.24 illustrates the block diagram for active low decoder.

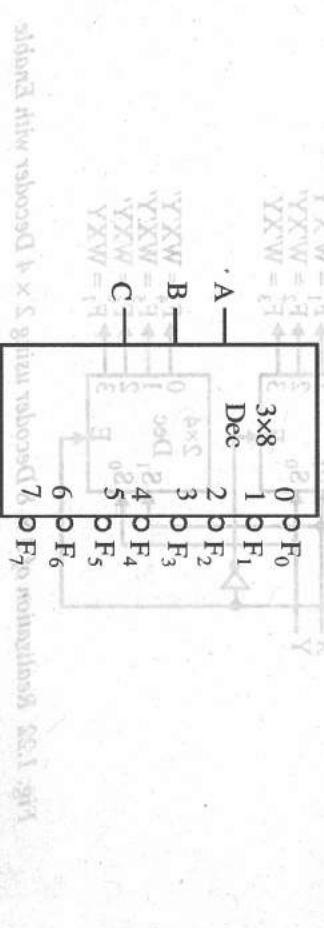


Fig. 1.24 Block Diagram for Active Low Decoder

While realizing any Boolean function in minterm or maxterm form, then depending on the type of decoder, i.e., whether it is active high or active low, we have to use either AND/OR or NAND/NOR gate to get the sum or product terms. The following examples illustrate it clearly.

Let us implement the following logic function using decoders and logic gates:

$$F(Q, X, P) = \sum m(0, 1, 4, 6, 7) = \prod m(2, 3, 5)$$

We may implement the function in several ways as follows:

- (a) Use a decoder with active high outputs with an OR gate. This will give minterm realization (refer Figure 1.25).

- $F(Q, X, P) = m_0 + m_1 + m_4 + m_6 + m_7$

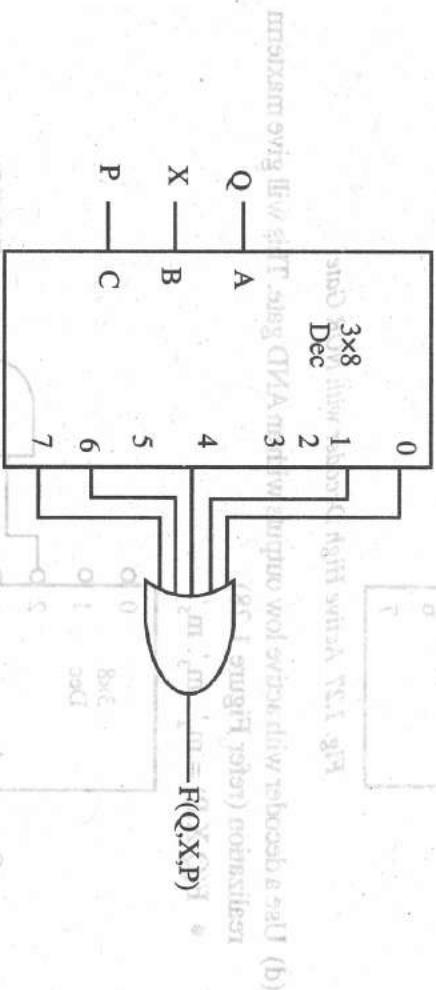


Fig. 1.25 Active High Decoder with OR Gate

- (b) Use a decoder with active low outputs with a NAND gate. This will give Minterm realization (refer Figure 1.26).

- $F(Q, X, P) = (m_0'. m_1'. m_4'. m_6'. m_7')$

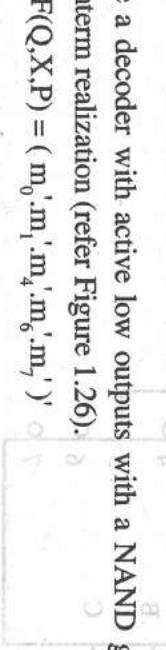


Fig. 1.26 Active Low Decoder with NAND Gate

- (c) Use a decoder with active high outputs with a NOR gate. This will give maxterm realization (refer Figure 1.27).

- $F(Q, X, P) = (m_2 + m_3 + m_5)'$



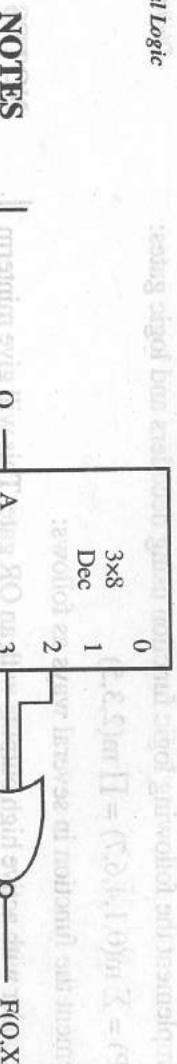


Fig. 1.27 Active High Decoder with NOR Gate

- (d) Use a decoder with active low outputs with an AND gate. This will give maxterm realization (refer Figure 1.28).

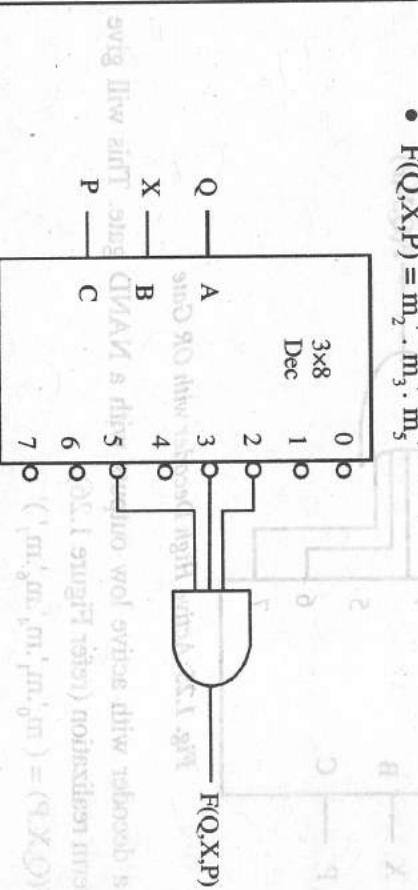
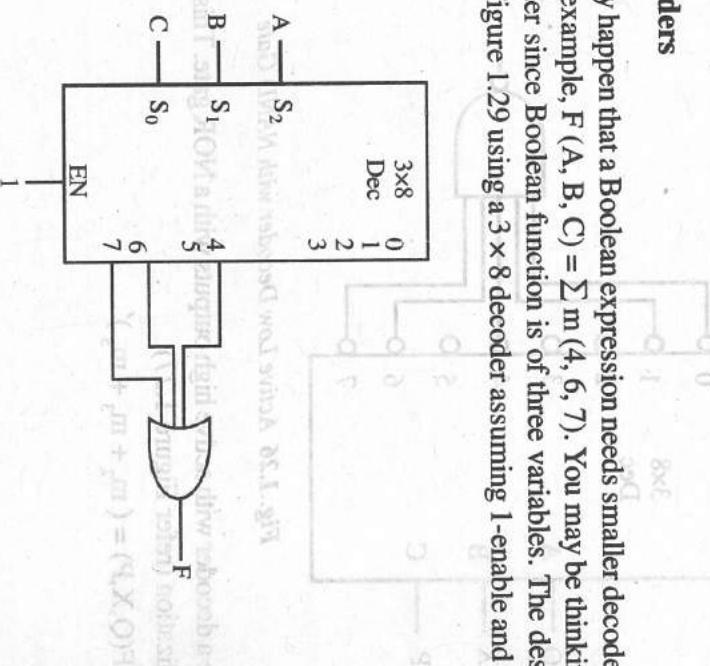


Fig. 1.28 Active Low Decoder with AND Gate

Reducing Decoders

Sometimes, it may happen that a Boolean expression needs smaller decoder to realize the function. For example, $F(A, B, C) = \sum m(4, 6, 7)$. You may be thinking that you need 3×8 decoder since Boolean function is of three variables. The design for the same is given in Figure 1.29 using a 3×8 decoder assuming 1-enable and active high outputs.

Fig. 1.29 Realization of $F(A, B, C)$ using 3×8 Decoder

If one of the input, say A, is connected as enable input, then the decoder is active when A = 1 and it is inactive when A = 0. Since the Minterms required for the above expression involves variable A = 1, so we can use 2×4 decoder with A as enable input (refer Figure 1.30).

NOTES

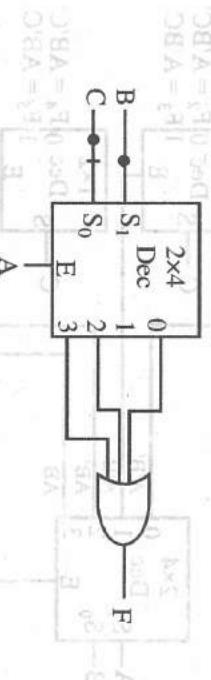


Fig. 1.30 Realization of $F(A, B, C)$ using 2×4 Decoder with Enable

Operation: When A = 1, then the decoder will generate all the Minterms for which A = 1, i.e., Minterms m_4, m_5, m_6 and m_7 , since decoder is active. Now for the desired Boolean expression, only three Minterms are needed and hence, they are given to the OR gate inputs.

We have seen that a decoder may be constructed from smaller decoders. Now some more realizations of the 3×8 decoder are shown with 2×4 decoder and 1×2 decoder.

- (i) Using two 2×4 decoders and one 1×2 decoder (refer Figure 1.31).

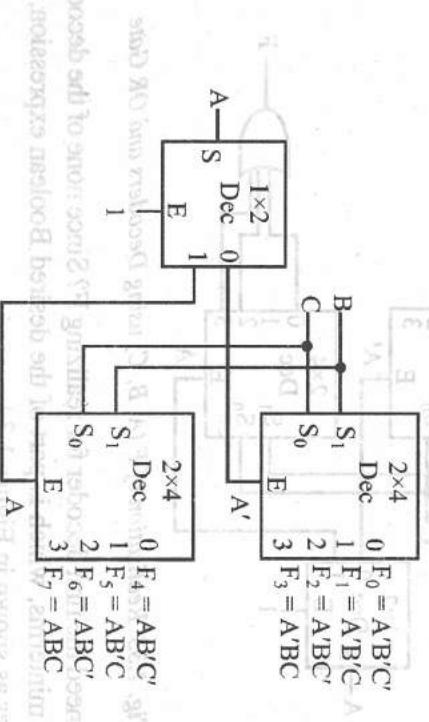


Fig. 1.31 Realizations of 3×8 Decoder using Two 2×4 Decoders and One 1×2 Decoder

- (ii) Using four 1×2 decoders and one 2×4 decoder (refer Figure 1.32).



svitca zí vloženého 2 bitu adresu, který ještěm se bavíme. V tomto obecném případě můžeme říct, že A je tříbitová adresa a B je jednobitový výstup. Tento výstup je nazýván $F_0 = ABC'$. $A' = A$ nebo $B = 1$ je vložená do druhého dekóduče, jehož výstup je nazýván $F_1 = A'B'C$.

NOTES

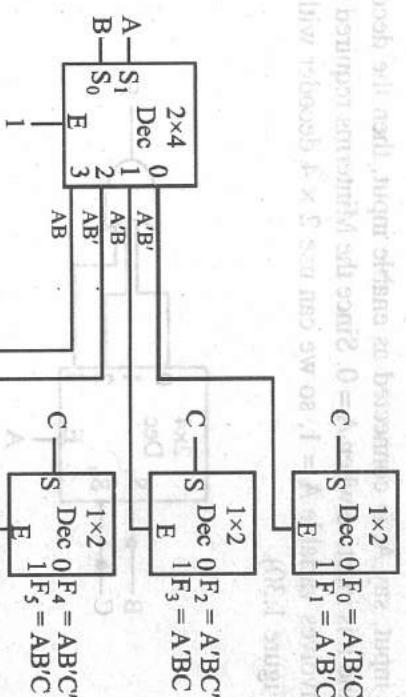


Fig. 1.32 Realization of 3×8 Decoder using One 2×4 Decoder and Four 1×2 Decoders

Using smaller decoders, sometimes we may be able to save some decoders, for example, $F(A, B, C) = \sum m(4, 6, 7)$ as shown in Figure 1.33.

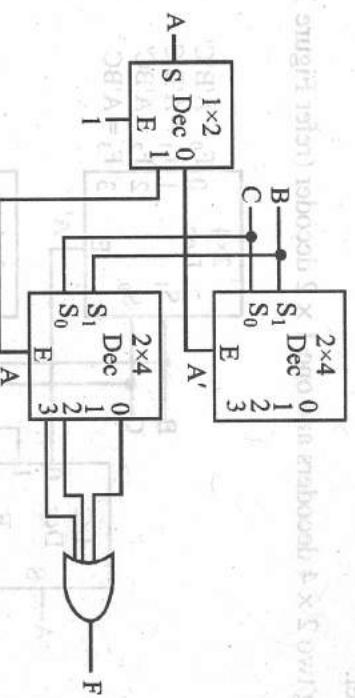


Fig. 1.33 Realization of $F(A, B, C)$ using Decoders and OR Gate

Do we really need the first decoder for realizing F ? Since none of the decoder outputs is giving any minterms, which is part of the desired Boolean expression, so we can save a decoder as shown in Figure 1.34.

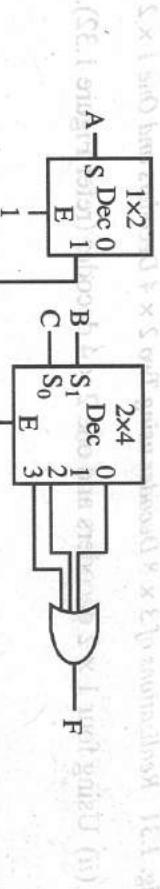


Fig. 1.34 Realization of $F(A, B, C)$ with Reduced Number of Decoders

Example 1.13: $F(A, B, C) = \sum m(0, 1, 2, 3, 6)$. Using 2×4 and 1×2 decoders realize the above Boolean function.

Solution: At the first instance, Figure 1.35 shows the realization which illustrates the representation of $F(A, B, C)$ using decoders and OR gate.

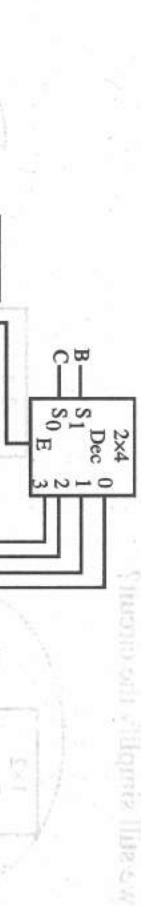


Fig. 1.35 Realization of $F(A, B, C)$ using Decoders and OR Gate

It can be seen that all the outputs of the first decoder are part of minterms for the function, so is this decoder needed? Can we do something about this? Yes, we may remove the top 2×4 decoder and connect the appropriate output from the 1×2 decoder directly to the OR gate. The same is shown in Figure 1.36.

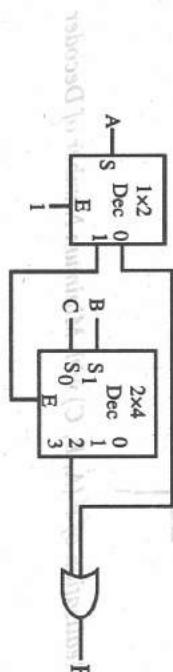


Fig. 1.36 Realization of $F(A, B, C)$ with Reduced Number of Decoders

For example, realization of $F(A, B, C) = \sum m(0, 3, 4, 7)$ is shown in Figure 1.37.

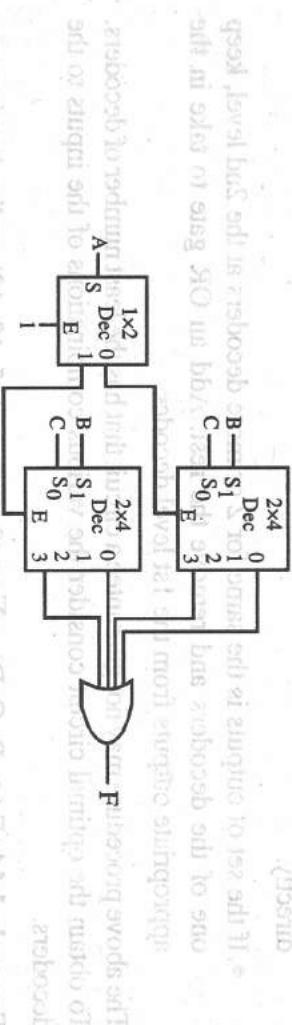


Fig. 1.37 Realization of $F(A, B, C)$ using Decoders and OR Gate

We have the same pattern of outputs from the 2 decoders, i.e., we take the first and fourth outputs from each decoder. Can we do something about it?

If we have the same pattern of outputs from 2 or more decoders at the second level, we may keep one decoder and use an OR gate on the corresponding outputs from the first-level decoder. The same is shown in Figure 1.38.

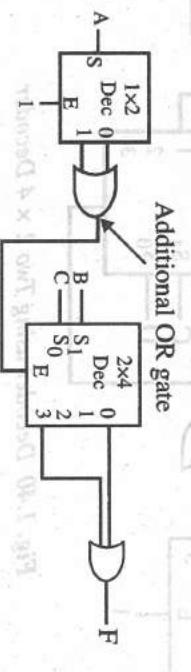
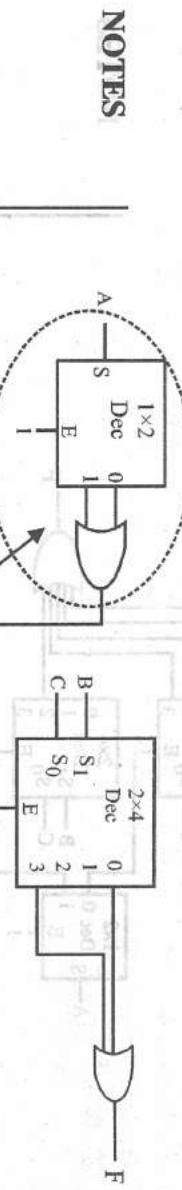


Fig. 1.38 Realization of $F(A, B, C)$ with Reduced Number of Decoders

NOTES

Complex logic functions	can be realized by connecting multiple decoders in parallel.
Inputs	are fed to the inputs of the decoders.
Outputs	of the decoders are connected to the inputs of an OR gate.
Decoders	are selected based on the required minterms.



The final reduced circuit diagram is shown in Figure 1.39.

Fig. 1.39 Realization of $F(A, B, C)$ with Minimum Number of Decoder

To sum up:

- If no outputs are needed from a 2nd level decoder, just remove the decoder.
- If all outputs are needed from a 2nd level decoder, remove the decoder and connect the corresponding output from the 1st level decoder to the OR gate directly.
- If the set of outputs is the same for 2 or more decoders at the 2nd level, keep one of the decoders and remove the rest. Add an OR gate to take in the appropriate outputs from the 1st level decoder.

The above procedure may not guarantee a circuit that has the least number of decoders. To obtain the optimal circuit consider the various combinations of the inputs to the decoders.

Example 1.14: $F(A, B, C, D) = \sum m(0, 1, 2, 3, 4, 5, 12, 13)$ realize it using two 2×4 decoders.

Solution: Figure 1.40 illustrates the decoder using two 2×4 decoder.

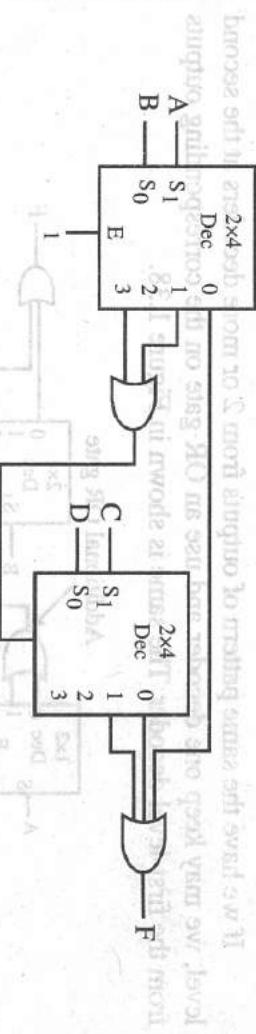


Fig. 1.40 Decoder using Two 2×4 Decoder

Check Your Progress

10. What is the aim of algebraic simplification?
11. What is a Karnaugh map?
12. What is a 2×4 decoder?
13. How are active low decoders represented?

1.9 ENCODER

Encoding is the converse of decoding. Given a set of input lines, where one has been selected, provide a code corresponding to that line. It contains 2^n (or fewer) input lines and n output lines. It is implemented with OR gates. Figure 1.41 illustrates the block diagram for 4×2 encoder.

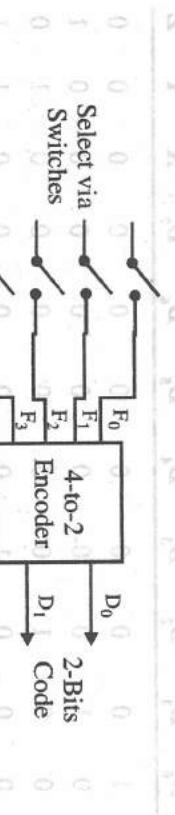


Fig. 1.41 Block Diagram for 4×2 Encoder

The truth table for the 4×2 encoder is shown in Table 1.18.

Table 1.18 Truth Table for 4×2 Encoder

S. Line	F ₀	F ₁	F ₂	F ₃	D ₁	D ₀
1	0	0	0	0	0	0
0	1	0	0	0	0	1
0	0	1	0	0	1	0
0	0	0	1	0	1	1
0	0	0	0	0	X	X
1	0	1	0	1	X	X
1	1	0	0	1	X	X
1	0	1	1	0	X	X
1	1	0	1	0	X	X
1	1	1	0	1	X	X
1	1	1	1	0	X	X
1	1	1	1	1	X	X

With the help of a K-map and don't care conditions, we can obtain:

$$D_0 = F_1 + F_3$$

$$D_1 = F_2 + F_3$$

The circuit diagram is given in Figure 1.42.

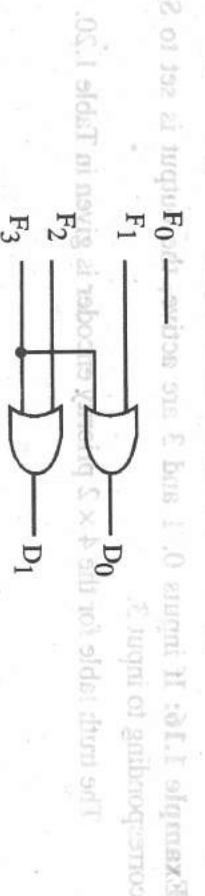


Fig. 1.42 Circuit Diagram for 4×2 Encoder

NOTES

Example 1.15: Octal-to-binary encoder. At any one time, only one input line has a value of 1. The truth table for the same is given in Table 1.19. Draw a circuit diagram for octal to binary converter.

Table 1.19 Truth Table for Octal to Binary Conversion

Inputs								Outputs		
D ₀	D ₁	D ₂	D ₃	D ₄	D ₅	D ₆	D ₇	X	Y	Z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	1	0	0	1	0	0
0	0	0	0	0	0	1	0	1	0	1
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	1	1

Solution: The Boolean expression for X, Y and Z can be directly written by observation. In Table 1.19, the following expression will be obtained for X, Y and Z.

$$X = D_4 + D_5 + D_6 + D_7$$

$$Y = D_2 + D_3 + D_6 + D_7$$

$$Z = D_1 + D_3 + D_5 + D_7$$

The circuit diagram for octal to binary converter is shown in Figure 1.43.

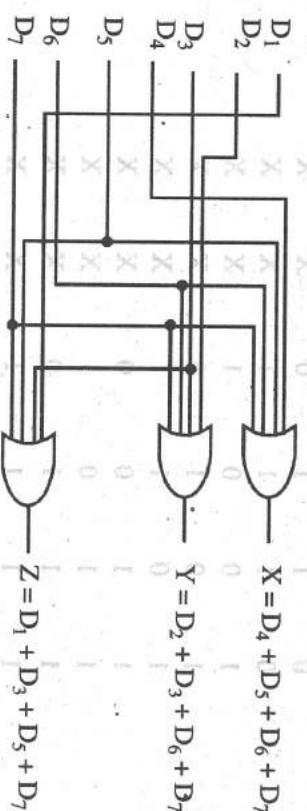


Fig. 1.43 Circuit Diagram for Octal to Binary Converter

Priority Encoder

When more than one input can be active, the priority encoder must be used. The output is set to correspond to the highest active input.

Example 1.16: If inputs 0, 1 and 3 are active, the output is set to $S_1S_0 = 11$, corresponding to input 3.

The truth table for the 4×2 priority encoder is given in Table 1.20.

Table 1.20 Truth Table for 4×2 Priority Encoder

Digital Logic

D₀	D₁	D₂	D₃	Inputs	Outputs
0	Y ₀	0	Y ₁	0, X	X
1	0	0	0	0, X	0
X	0	0	0	0, X	1
X	0	1	0	0, X	0
X	0	X	1	0, X	1
X	X	0	0	1, X	0
X	X	X	X	1, X	1

Solution: Solving the expression for Y_1 and Y_0 we have $Y_1 = D_2 + D_3$ and $Y_0 = D_3 + D_2' D_1$ (refer Figure 1.44).

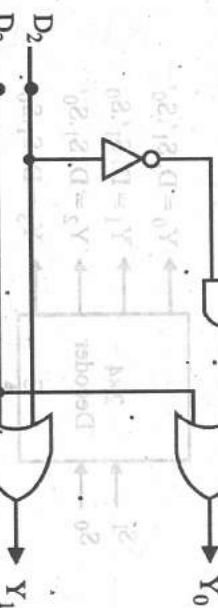


Fig. 1.44 Circuit Diagram for 4×2 Priority Encoder, with D_3 having the Highest Priority

1.10 DEMULTIPLEXERS

A demultiplexer or demux is a device that takes a single input signal and selects one of many data output lines, which is connected to the single input. A multiplexer is often used with a complementary demultiplexer on the receiving end. The data consists of n lines (for n-bit words). Data input lines provide the information to be stored (written) into the memory while data output lines carry the information out (read) from the memory.

Given is an input line and a set of selection lines, a demultiplexer directs data from input to a selected output line. An example of a 1-to-4 demultiplexer is shown in Figure 1.45.

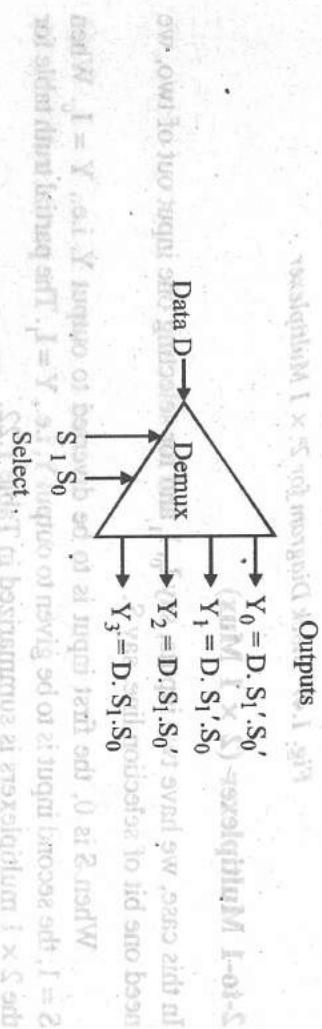


Fig. 1.45 Block Diagram for 1×4 Demultiplexer

The truth table given in Table 1.21 shows that depending on the selection signal values, input signal D is diverted to one of the output lines.

Table 1.21 Truth Table for 1×4 Demultiplexer

X	S ₁	S ₀	Y ₀	Y ₁	Y ₂	Y ₃
0	0	0	D	0	0	0
1	0	1	0	D	0	0
0	1	0	0	0	1	0
1	1	1	0	0	0	D

It can be observed that a demultiplexer is actually identical to a decoder with enable, as illustrated in Figure 1.46 with diagram just rotated, i.e., data line in the demultiplexer is enable input of the decoder and selection lines of the demultiplexer is input of the decoder. The same is shown below in Figure 1.46.

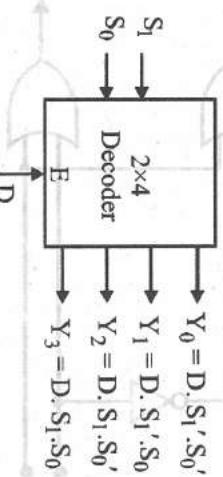


Fig. 1.46 Block Diagram of 2×4 Decoder with Enable Input

1.11 MULTIPLEXERS

A multiplexer is a circuit, which has a number of *input* lines and *selection* lines with one *output* line. It steers one of 2^n inputs to a single output line using n selection lines (refer Figure 1.47). It is also known as a *data selector*.

Inputs
⋮
Multiplexer
Select
Output

Fig. 1.47 Block Diagram for $2^n \times 1$ Multiplexer

Fig. 1.47 Block Diagram for $2^n \times 1$ Multiplexer

2-to-1 Multiplexer (2 × 1 Mux)

In this case, we have two inputs, say I₀, I₁ and for selecting one input out of two, we need one bit of selection line, say S.

When S is 0, the first input is to be diverted to output Y, i.e., Y = I₀. When S = 1, the second input is to be given to output Y, i.e., Y = I₁. The partial truth table for the 2 × 1 multiplexers is summarized in Table 1.22.

Table 1.22 Partial Truth Table for 2×1 Multiplexers

Inputs	Selection(S)	Output Y
I ₀	0	I ₀
I ₁	1	I ₁

NOTES

We can expand the truth table as shown in Table 1.23.

Table 1.23 Complete Truth Table for 2×1 Multiplexers

I ₁	I ₀	S	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Solving expression for Y in terms of I₀, I₁ and S, we have the following Boolean expression:

$$Y = I_0.S' + I_1.S$$

The circuit diagram for the 2×1 multiplexer is shown in Figure 1.48.

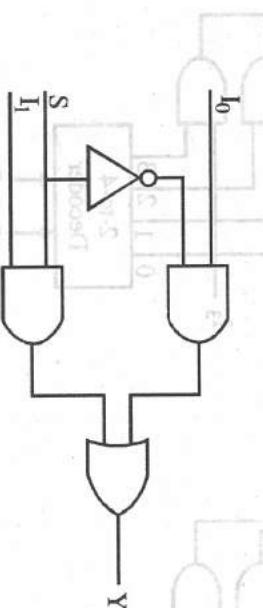


Fig. 1.48 Circuit Diagram for 2×1 Multiplexer

4-to-1 Multiplexer (4 × 1 Mux)

In this case, we have four inputs, say I₀, I₁, I₂, I₃ and for selecting one input out of four, we need two bits of selection line, say S₁ and S₀ (refer Figure 1.49).

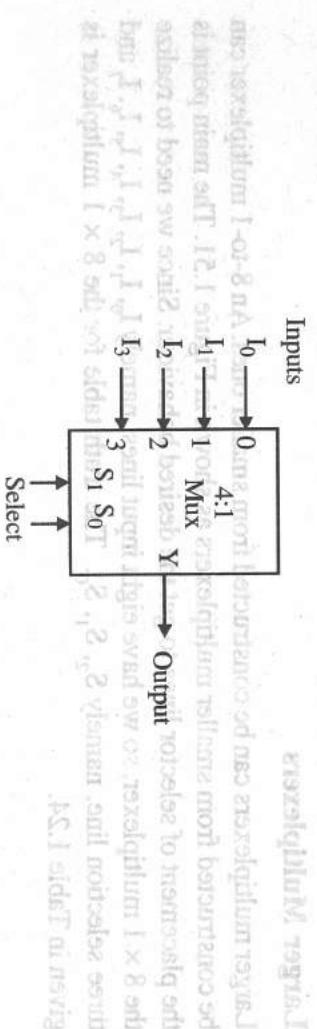


Fig. 1.49 Block Diagram for 4×1 Multiplexer

Table 1.23 Truth Table for 4×1 Multiplexer

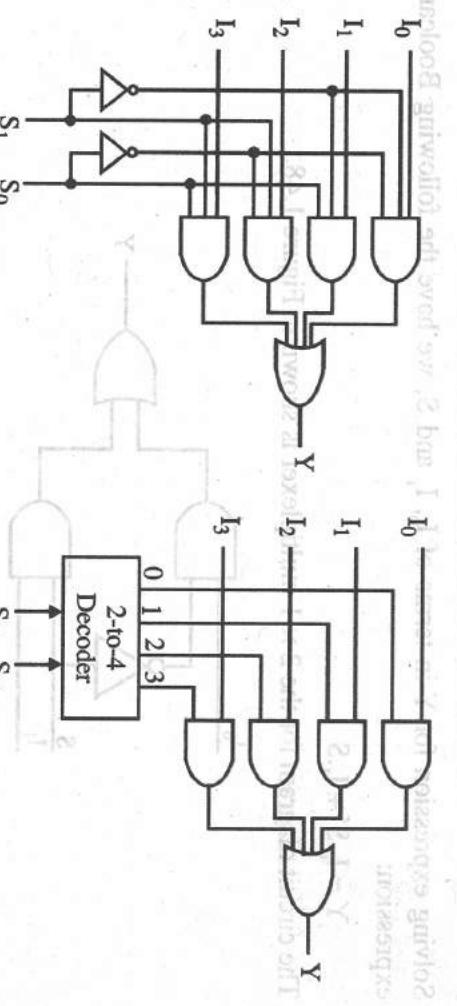
$I_0 = I_1$	I_2	I_3	S_1	S_0	Y
d_0	d_1	d_2	d_3	0	0
d_0	d_1	d_2	d_3	0	1
d_0	d_1	$\Sigma d_2 + d_3$	1	0	$d_2 + d_3$
d_0	d_1	$d_2 \times d_3$	1	1	$d_2 \times d_3$

Table 1.23 summarizes the truth table for 4×1 multiplexer.

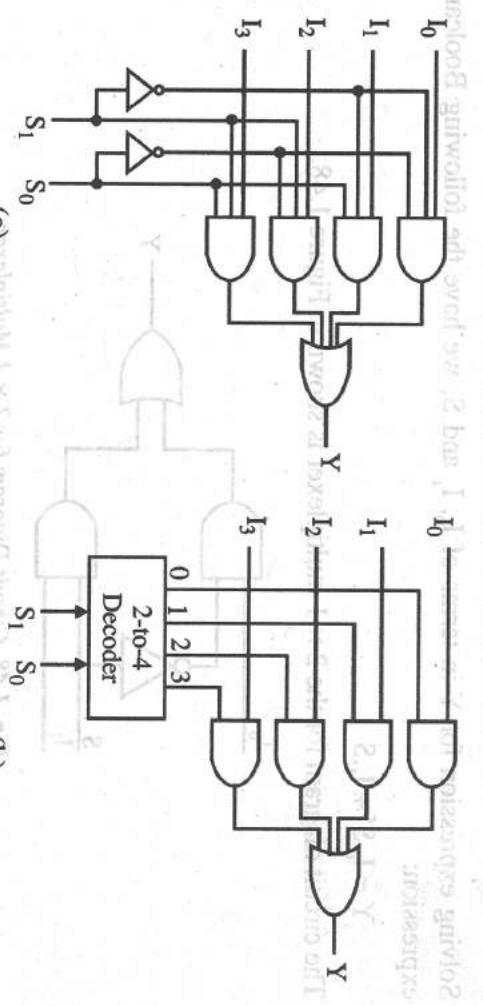
The Boolean expression for the 4×1 multiplexer can be derived by the inspection method. As we know that output Y is equal to I_0 when $S_1 S_0 = 00$, then we can write the following expression for output, $Y = I_0 S_1' S_0'$. Similarly, the output can be written for the other terms also and finally adding all the terms will give the overall result for output Y. The same is given as follows:

$$Y = I_0(S_1' S_0') + I_1(S_1' S_0) + I_2(S_1 S_0') + I_3(S_1 S_0)$$

The circuit diagram and realization using 2×4 decoder of 4×1 Mux for the above expression is given in Figures 1.50 (a) and (b), respectively.



(a)



(b)

Fig. 1.50 (a) Circuit Diagram (b) Realization using 2×4 Decoder of 4×1 Multiplexer

It is to be noted that A 2^n -to-1 line multiplexer or simply $2^n:1$ Mux is made from an $n:2^n$ decoder by adding to it 2^n input lines, one to each AND gate.

Larger Multiplexers

Larger multiplexers can be constructed from smaller ones. An 8-to-1 multiplexer can be constructed from smaller multiplexers as shown in Figure 1.51. The main point is the placement of selector lines to get the desired behaviour. Since we need to realize the 8×1 multiplexer, so we have eight input lines, namely $I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7$ and three selection line, namely S_2, S_1, S_0 . The truth table for the 8×1 multiplexer is given in Table 1.24.

Table 1.24 Truth Table for the 8 × 1 Multiplexer

Digital Logic

S_2	S_1	S_0	Y
0	0	0	I ₀
0	0	1	I ₁
0	1	0	I ₂
0	1	1	I ₃
1	0	0	I ₄
1	0	1	I ₅
1	1	0	I ₆
1	1	1	I ₇

Realization of 8×1 multiplexer using two 4×1 multiplexer and one 2×1 multiplexer is shown in Figure 1.51.

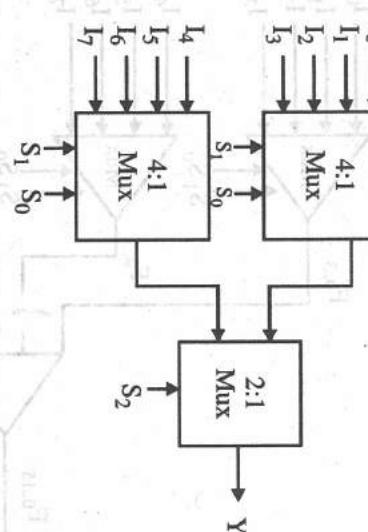


Fig. 1.51 Realization of 8×1 Multiplexer using Smaller Multiplexers

Operation: When S_2 is 0, we know that 8×1 multiplexer will give outputs from I_0, I_1, I_2, I_3 . Now using 4×1 multiplexer with input lines as I_0, I_1, I_2, I_3 and selection lines as S_1, S_0 , the same can be achieved.

Similarly, for S_2 is 1, we know that 8×1 multiplexer will give outputs from I_4, I_5, I_6, I_7 . Now using 4×1 multiplexer with input lines as I_4, I_5, I_6, I_7 and selection lines as S_1, S_0 , the same can be achieved.

Now for the overall output of 8×1 multiplexer, the outputs of the above two

4×1 multiplexer should be given to inputs of 2×1 multiplexer so that depending on the selection S_2 , we can get the behaviour of 8×1 multiplexer.

Checking: When selection bit S_2, S_1, S_0 is 001, then the first 4×1 multiplexer will give output I_1 and the second will give output I_5 . Now since $S_2 = 0$, the output of 2×1 multiplexer will be the first input, i.e., I_1 .

Another implementation of an 8-to-1 multiplexer using smaller multiplexers is shown in Figure 1.52.

$$\text{Sel} \oplus I_1 : E \oplus \text{Sel} \oplus \text{Sum}-Q = Y, B'C + ABC + AB'C + ABC'$$

$$Z = \text{Sum}(I_1, S_2, 0)$$

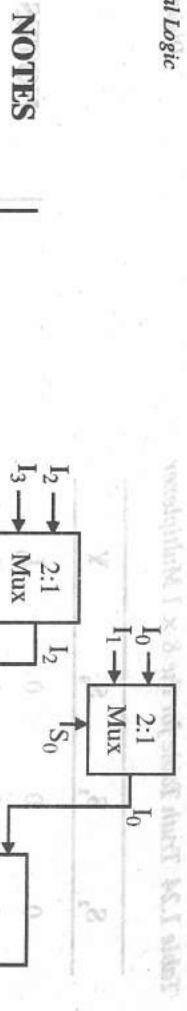


Fig. 1.52 Realization of 8×1 Multiplexer using Smaller Multiplexers

For example, a 16-to-1 multiplexer can be constructed from five 4-to-1 multiplexers. Figure 1.53 illustrates the realization of 16×1 multiplexer using smaller multiplexers.

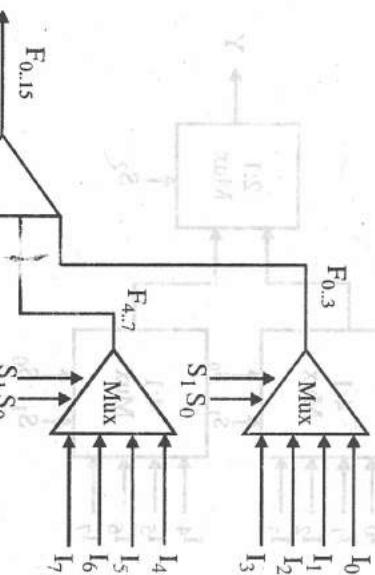


Fig. 1.53 Realization of 16×1 Multiplexer using Smaller Multiplexers

Implementing Boolean Functions using Multiplexers

A Boolean function can be implemented using multiplexers. A 2^n -to-1 multiplexer can implement a Boolean function of n input variables as follows:

Step 1: Express in the Sum-Of-Minterms form. For example,

$$F(A, B, C) = A'B'C + A'BC + AB'C + ABC'$$

$$= \Sigma m(1, 3, 5, 6)$$

Step 2: Connect n variables to the n selection lines.

Step 3: Put a '1' on a data line if it is a Minterm of the function, otherwise put '0'.

Supplying '1' to I_1, I_3, I_5, I_6 and '0' to the rest input of 8×1 multiplexer with selection input as A, B, C will result in output $F = m_1 + m_3 + m_5 + m_6$.

NOTES

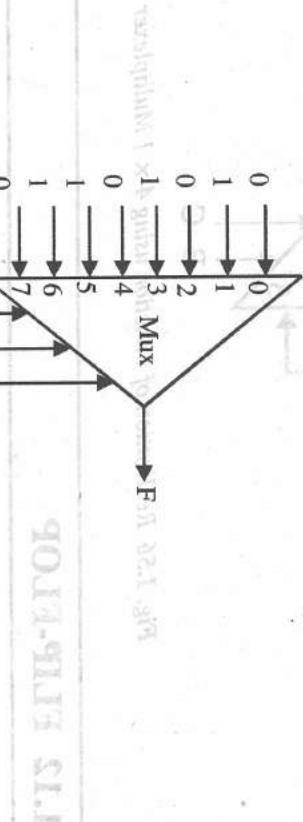


Fig. 1.54 Realization of Function using 8×1 Multiplexer

Figure 1.54 illustrates the realization of function using 8×1 multiplexer. It is also possible to realize the Boolean function by using smaller multiplexers. Earlier, we have seen how a 2^n -to-1 multiplexer could be used to implement any Boolean function of n input variables. However, we can use a single smaller $2^{(n-1)}$ -to-1 multiplexer to implement any Boolean function of n input variables. In this method, one of the function variables is given to multiplexer inputs.

Let us look at this example: $F(A,B,C) = \sum m(0,1,3,6) = A'B'C' + A'B'C + A'BC + ABC'$

If A and B are selection bit for S_1 and S_0 , then depending on the value of A and B, the inputs of the multiplexer will be as follows:

If $A = 0$ and $B = 0$, then $F(A,B,C) = C' + C + 0 + 0 = 1$ and we know that input at I_0 pin will be the output, so we have to connect 1 to I_0 pin of Mux.

If $A = 0$ and $B = 1$, then $F(A,B,C) = 0 + 0 + C + 0 = C$ and we know that input at I_1 pin will be the output, so we have to connect C to I_1 pin of Mux. Similarly we can verify the rest of the connection. Figure 1.55 illustrates the realization of function using 4×1 multiplexer.

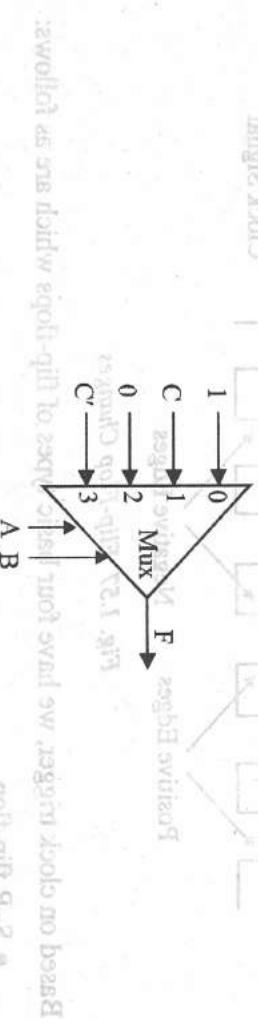
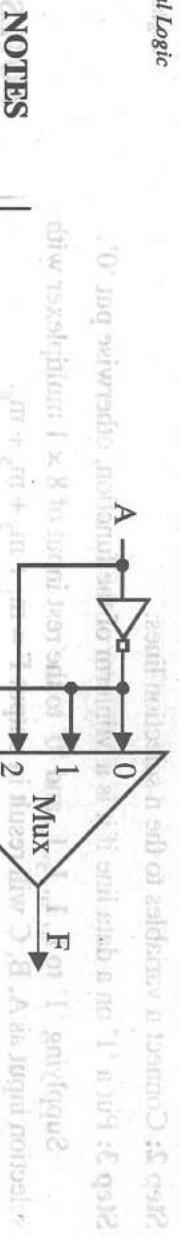


Fig. 1.55 Realization of Function using 4×1 Multiplexer

What happens when we use A for input lines and B, C for selector lines? When B = 0 and C = 0, F = A', connect A' to I_0 pin of multiplexer. The rest of the inputs of multiplexer can be verified easily. Figure 1.56 illustrates the realization of function using 4×1 multiplexer.

Fig. 1.56 Realization of Function using 4×1 Multiplexer

• J-K flip-flop
• T flip-flop
• D flip-flop
• S-R flip-flop

NOTES

1.12 FLIP-FLOP

In digital electronics, a flip-flop or latch is a circuit that has two stable states and can be used to store state information. The circuit can be made to change state by signals applied to one or more control inputs and will have one or two outputs. It is the basic storage component in sequential logic. Flip-flops and latches are used as data storage elements. Such data storage can be used for storage of state and such a circuit is described as sequential logic. Flip-flops can be either simple (transparent or opaque) or clocked (synchronous or edge-triggered); the simple ones are commonly called latches. The word latch is mainly used for storage elements, while clocked devices are described as flip-flops. Flip-flops can be either simple (transparent or asynchronous) or clocked (synchronous). The transparent ones are commonly called latches. The word latch is mainly used for storage elements, while clocked devices are described as flip-flops.

Remember that a latch circuit is not suitable in the synchronous circuit design because of its transparency nature. This problem can be solved by applying timing signal like clock, which will restrict the time at which memory will change state. This leads to designing of flip-flop circuit, which is an edge-triggered memory element. The state of the flip-flop changes as a specified point on a trigger input (refer Figure 1.57). State can change either at positive transition, i.e., when clock is changing from 0 to 1 or at negative transition, i.e., when clock is changing from 1 to 0.



Fig. 1.57 Flip-Flop Changes

Check Your Progress

14. What is encoding?
15. When is priority encoder used?
16. What is a demultiplexer?
17. What is a multiplexer?

Based on clock trigger, we have four basic types of flip-flops which are as follows:

- S-R flip-flop

*Ans. 1.52 Realization of Priority encoder using 4 × 1 multiplexer
Ans. 1.53 Realization of Demultiplexer using 1-to-4 decoder
Ans. 1.54 Realization of J-K flip-flop using 1-to-2 decoder
Ans. 1.55 Realization of T flip-flop using 1-to-2 decoder
Ans. 1.56 Realization of D flip-flop using 1-to-2 decoder
Ans. 1.57 Realization of S-R flip-flop using 1-to-2 decoder
Ans. 1.58 Realization of J-K flip-flop using 1-to-2 decoder
Ans. 1.59 Realization of T flip-flop using 1-to-2 decoder
Ans. 1.60 Realization of D flip-flop using 1-to-2 decoder*

The symbols for the D, S-R and J-K flip-flops are shown in Figure 1.58.

Digital Logic

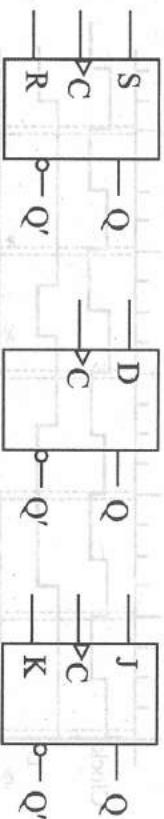


Fig. 1.58(a) Positive Edge-Triggered Flip-Flop

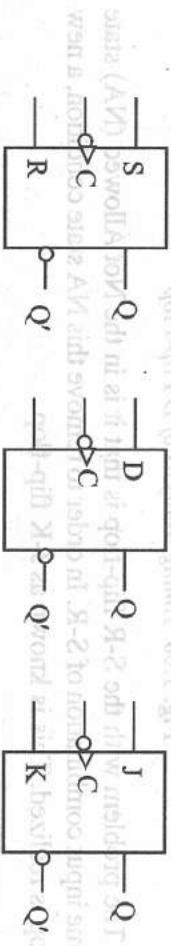


Fig. 1.58(b) Negative Edge-Triggered Flip-Flop

Circle marks at the clock shows negative triggered flip-flop (refer Figure 1.58(b)). The working of S-R flip-flop is same as that of the basic latch but states change only at the edges. In between, two transition states of the flip-flop remain unchanged. The characteristic table of the S-R flip-flop is given in Table 1.25.

Table 1.25 Characteristic Table for Positive-Triggered S-R Flip-Flop

Clock	S	R	Q(n+1)
↑	0	0	NC
↑	0	1	0
↑	1	0	1
↓	X	X	NC/Hold

The upward arrow shows positive transition on the clock. Whenever there is a positive transition on a clock, depending on the values of the S-R input, the next state is decided. For negative transition as shown by the downward arrow, there is no change in flip-flop, i.e., hold state.

1.12.1 D Flip-Flop

It has single input D. The block diagram and characteristic table of D flip-flop is shown in Figure 1.59.



Fig. 1.59 (a), (b) Block Diagram and Characteristic Table for D Flip-Flop

NOTES

Figure 1.60 illustrates timing analysis of D flip-flop using S-R, Q and D as inputs.

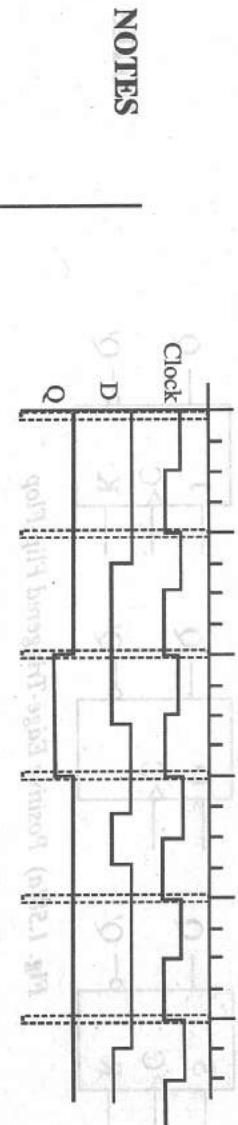


Fig. 1.60 Timing Analysis of D Flip-Flop

The problem with the S-R flip-flop is that it is in the Not Allowed (NA) state for some input combination of S-R. In order to remove this NA state condition, a new flip-flop is realized. This is known as J-K flip-flop.

1.12.2 J-K Flip-Flop

It is same as that of S-R flip-flop but it is having feedback Q and Q' to the NAND gate to which S and R inputs are connected along with clock pulse. The internal circuit diagram of J-K flip-flop is shown in Figure 1.61.

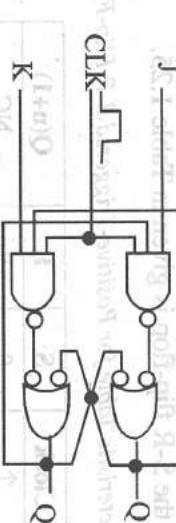
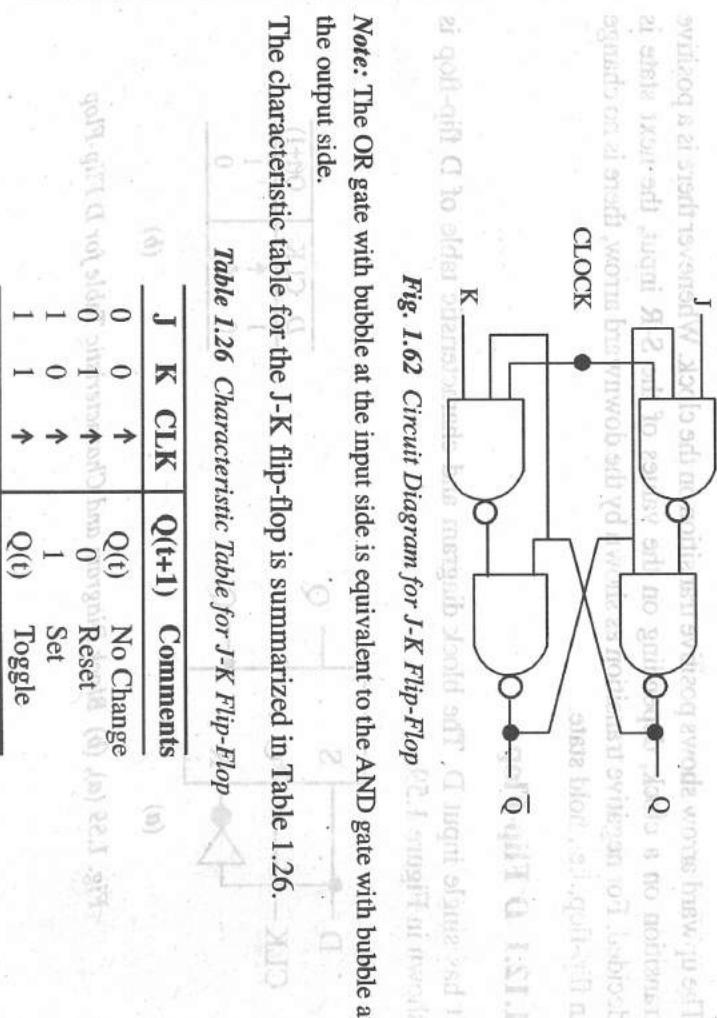


Fig. 1.61 Circuit Diagram for JK Flip-Flop

Note that Q' is feedback to the NAND gate with J and $clock$ as other inputs. Similarly, Q is feedback to the NAND gate with K and $clock$ as other inputs. The alternate circuit diagram with the NAND gate is shown in Figure 1.62.



Note: The OR gate with bubble at the input side is equivalent to the AND gate with bubble at the output side.

The characteristic table for the J-K flip-flop is summarized in Table 1.26.

Table 1.26 Characteristic Table for J-K Flip-Flop

J	K	CLK	Q(t+1)	Comments
0	0	↑	Q(t)	No Change
0	1	↑	0	Reset
1	0	↑	1	Set
1	1	↑	Q(t)	Toggle

In this case, for $J = K = 1$, the next state is the complement of the present state. The same can be verified from the circuit diagram. This is known as toggle state also. The expanded truth table is given in Table 1.27.

Table 1.27 Truth Table for J-K Flip-Flop

NOTES

Q	J	K	Q(t+1)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Table 1.28 Truth Table for T Flip-Flop

Q	T	Q(t+1)
0	0	0
0	1	1
1	0	1
1	1	0

Figure 1.63 illustrates timing analysis with different values of J , K and Clock.

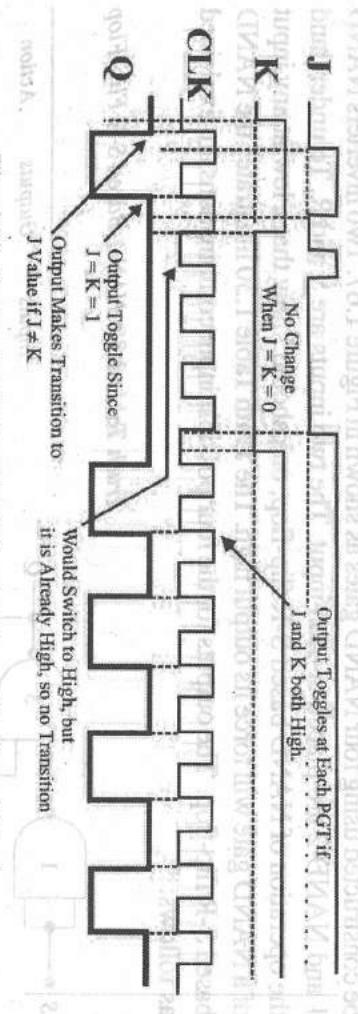


Fig. 1.63 Timing Analysis with Different Values of J, K and Clock

1.12.3 T Flip-Flop

It is a single input version of J-K flip-flop formed by tying both the inputs of J-K. The circuit and block diagrams are shown in Figures 1.64 and 1.65.

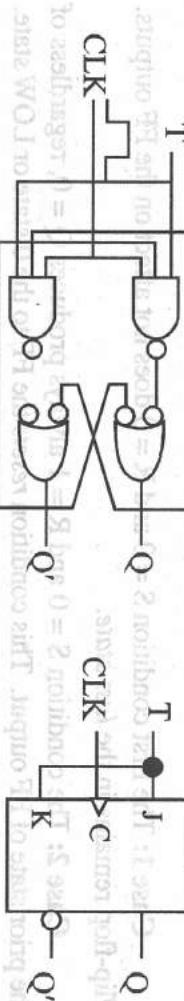


Fig. 1.64 T Flip-Flop Circuit Diagram

The truth table and characteristic table is summarized in Table 1.28 and Table 1.29, respectively. In this section, we will discuss the basic characteristics of T flip-flops.

Table 1.28 Truth Table for T Flip-Flop *Table 1.29 Characteristic Table for T Flip-Flop*

Q	T	Q(t+1)	T	CLK	Q(t+1)	Comments
0	0	0	0	0	0	No change
0	1	1	1	0	1	Q(t) = 0 → Q(t+1) = 1
1	0	1	1	1	0	Q(t) = 1 → Q(t+1) = 0
1	1	0	1	1	0	Q(t) = 1 → Q(t+1) = 0

When $T = 0$, this will cause $J = 1$ and $K = 1$, i.e., No Change state. When $T = 1$, this will cause $J = 0$ and $K = 0$, i.e., Toggle state condition. Figure 1.66 illustrates the timing analysis of T flip-flop.

NOTES

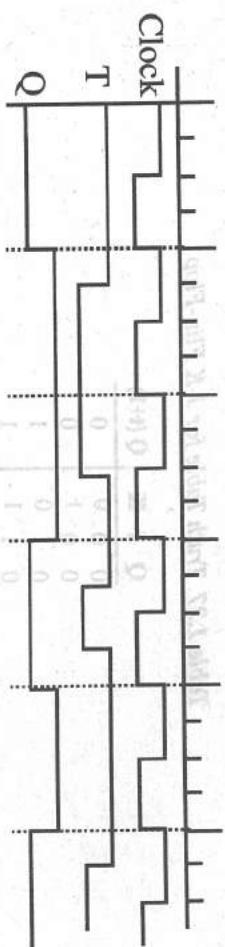


Fig. 1.66 Timing Analysis of T Flip-Flop

1.12.4 S-R Flip-Flop

A flip-flop is an electronic circuit which has two stable states. The S-R flip-flop can be constructed using four NAND gates as shown in Figure 1.67. Two inverters NAND 1 and NAND 2 are placed at each input. The two inputs are S and R. To understand the operation of NAND based S-R flip-flop, one should know that a low at any input of a NAND gate will force its output high. The Truth Table 1.30 illustrates the NAND based S-R flip-flop. The outputs for the four possible input combinations are explained as follows:

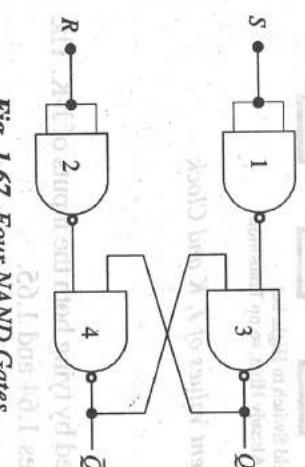


Fig. 1.67 Four NAND Gates

Case 1: The first condition $S = 0$ and $R = 0$ does not affect on the FF outputs. Flip-flop remains in the *last state*.

Case 2: The condition $S = 0$ and $R = 1$ always produces $Q = 0$, regardless of the prior state of FF output. This condition *resets* the FF to the 0 state or LOW state.

Case 3: The condition $S = 1$ and $R = 0$ always produces $Q = 1$ regardless of the present state of the FF output. This condition sets the FF to the 1 state or HIGH state.

Case 4: The last condition $S = 1$ and $R = 1$ is ambiguous and should not be used. When both inputs are HIGH, both outputs are also HIGH. This goes against the fact that output Q and \bar{Q} harmonize with each other.

1.12.5 Asynchronous Flip-Flop

The normal data inputs to a flip-flop (D, S and R, J and K, T) are referred to as *synchronous* inputs because they affect the outputs (Q and \bar{Q}) with the clock signal transitions. These types of the flip-flops are known as synchronous flip-flop. Another category of the flip-flops is asynchronous flip-flop. In these types of flip-flops, we

Truth Table 1.30 NAND-Based S-R Flip-Flop

Inputs	Outputs	Action
0 0	1 0	0
0 1	0 1	Last State
1 0	0 1	Reset
1 1	1 0	Set
1 1	1 1	Forbidden

have some extra inputs to control the states of flip-flop without waiting for the clock to appear. These extra inputs are called *asynchronous inputs* because they can set or reset the flip-flop, regardless of the status of the clock signal. Typically, they are called *Preset* and *Clear*. Asynchronous inputs, just like synchronous inputs, can be engineered to be active high or active low.

Asynchronous J-K Flip-Flop

The circuit diagram shown in Figure 1.68 shows asynchronous J-K flip-flop.

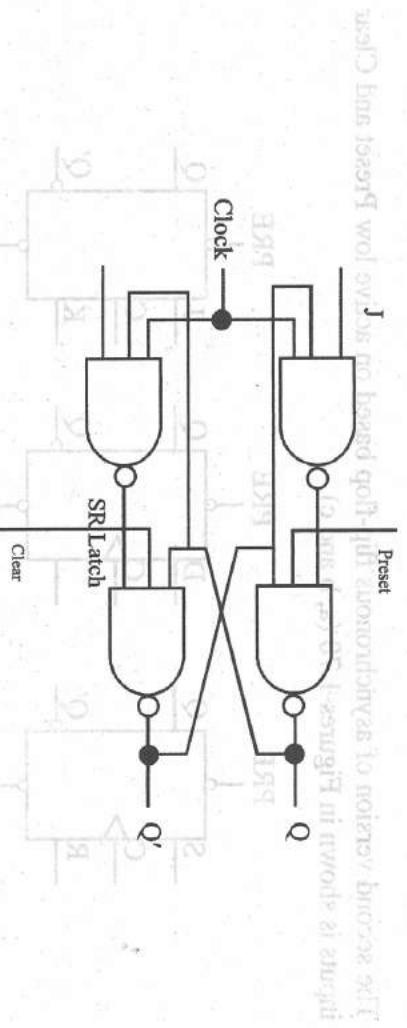


Fig. 1.68 Circuit Diagram for Asynchronous J-K Flip-Flop

This circuit is same as that of J-K flip-flops but with two extra input signals, Preset and Clear. These signals are connected to the second-stage NAND gates as shown in Figure 1.68. The characteristic table is shown in Table 1.31.

Table 1.31 Characteristic Table for Asynchronous J-K Flip-Flop with Active High Preset/Clear

Target	Preset	Clear	Clock	J	K	Q(n+1)
0	0	0	X	X	X	NA
0	1	0	X	X	X	1
1	0	X	X	X	X	0
1	1	X	0	0	NC	NA
1	1	↑	0	1	0	1
1	1	↑	1	0	1	1
1	1	↑	1	1	Q/Toggle	0
1	1	↓	X	X	NC/Hold	0

Preset and Clear should not be 0 at the same time; otherwise, both the outputs will be 1, which is known as invalid state. When Preset = 0, Clear = 1, then independent on clock transition and values of J and K, output Q (n + 1) will be 1. Similarly, When Preset = 1, Clear = 0, then independent on clock transition and values of J and K, output Q (n + 1) will be 0. When Preset = 1, Clear = 1, then the flip-flop will be working as normal flip-flop and its working will depend on clock transition and values of J and K. The above type of asynchronous Preset/Reset is known as active high.

The S-R, D and J-K flip-flops with asynchronous inputs can be represented in the block form and is shown in Figures 1.69 (a, b and c).

NOTES

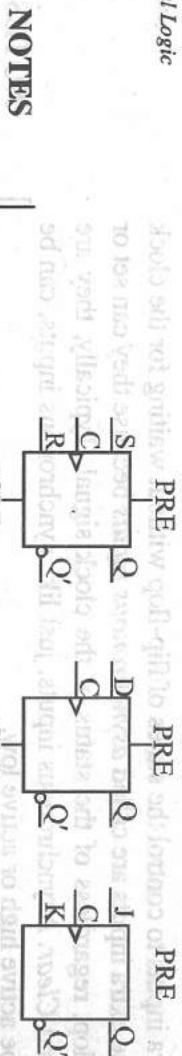


Fig. 1.69 (a), (b), (c) Block Diagram for Asynchronous S-R, D, J-K Flip-Flops with Active High Preset/Clear

The second version of asynchronous flip-flop based on active low Preset and Clear inputs is shown in Figures 1.70 (a, b and c).

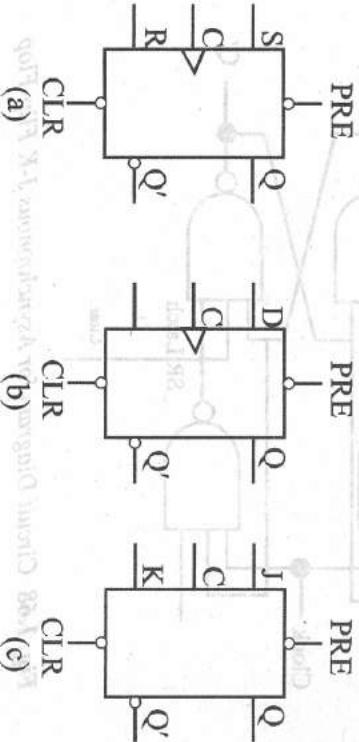


Fig. 1.70 (a), (b), (c) Block Diagram for Asynchronous S-R, D, J-K Flip-Flops with Active Low Preset/Reset and Active Low Clear

Bubble at the Reset and Clear pin shows negation. The characteristic table for the same is given in Table 1.32. The above circuit shows that when Reset = 0 or 1, the NAND gate receives logic 1 and 0, respectively, at the second stage of the NAND gate, near output Q. Similarly, when Clear = 0 or 1, the NAND gate receives logic 1 and 0, respectively, at the second stage of the NAND gate, near output Q'. In this case, Reset = 1 will give Q = 1 and Clear = 1 will give Q = 0 and hence justifies the name of Preset and Clear.

Table 1.32 Characteristic Table for Asynchronous J-K Flip-Flop with Active Low Preset/Reset

Preset	Clear	Clock	J	K	Q(n+1)
0	0	X	X	X	NA
0	1	X	X	X	1
1	0	X	X	X	0
1	1	0	0	0	NC
1	1	1	0	1	0
1	1	1	1	0	1
1	1	1	1	1	Q' Toggle
1	1	1	X	X	NC/Hold

1.13 REGISTERS

An n-bit register has a group of n flip-flops and some logic gates. It is capable of storing n bits of information. The flip-flops store information while the control circuitry decides when and how new information is transferred into the register. Some functions of the register are listed as follows:

- Retrieve data from register.
- Store/Load new data into register either serially or parallelly.
- Shift the data within register, either in the left or right direction.

Simple Registers

It does not have any external gate. It just transfers the input data to the output side. A 4-bit parallel register is shown in Figure 1.71, which transfers input data on each flip-flop to the respective output. Generally, D flip-flop is used in registers realization.

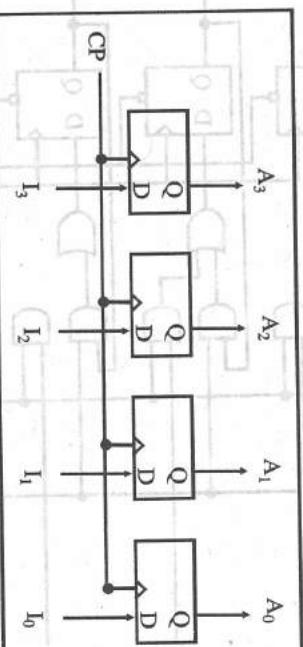


Fig. 1.71 4-Bit Parallel Register

Working: Inputs lines I₀, I₁, I₂, I₃ are connected to the D flip-flop inputs lines. When the clock is having positive transitions, the respective input values are transferred from the input side to the respective output side. This type of register is known as Parallel-In-Parallel-Out (PIPO).

Registers with Parallel Load

Instead of loading the register at every clock pulse, we may require to control the loading of the register by using some control signal. Loading a register means transferring new information into the register and it requires a load control input. Sometimes, we need parallel loading, i.e., transferring all the bits simultaneously.

For example, we can design a 4-bit register with control signal load with parallel loading feature. Assuming that when Load = 1, parallel loading is to be done and when Load = 0, the old value is to be retained.

Design: Assuming that we have four input signals, I₀, I₁, I₂ and I₃.

When Load = 1, the inputs should be reaching the inputs of the respective D flip-flop inputs of the register and on the clock transition, it should be transferred to the output of the respective flip-flop. When Load = 0, the circuit should be maintaining the previous output or value of register. So we need some circuitry to handle the input of D flip-flop to satisfy the above requirement. It can be seen that D inputs should be $D_I = L \cdot I_i + Q_r \cdot L'$.

NOTES

The circuit diagram for the same is shown in Figure 1.72.

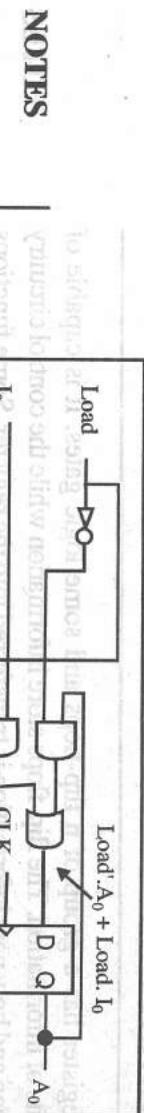


Fig. 1.72 Circuit Diagram for 1-Bit Register with Parallel Loading Feature

The complete circuit diagram for the same is given in Figure 1.73 with the feature of clear signal for clearing the register with 0 initially and the negative trigger on clock for the transfer of data from input to output.

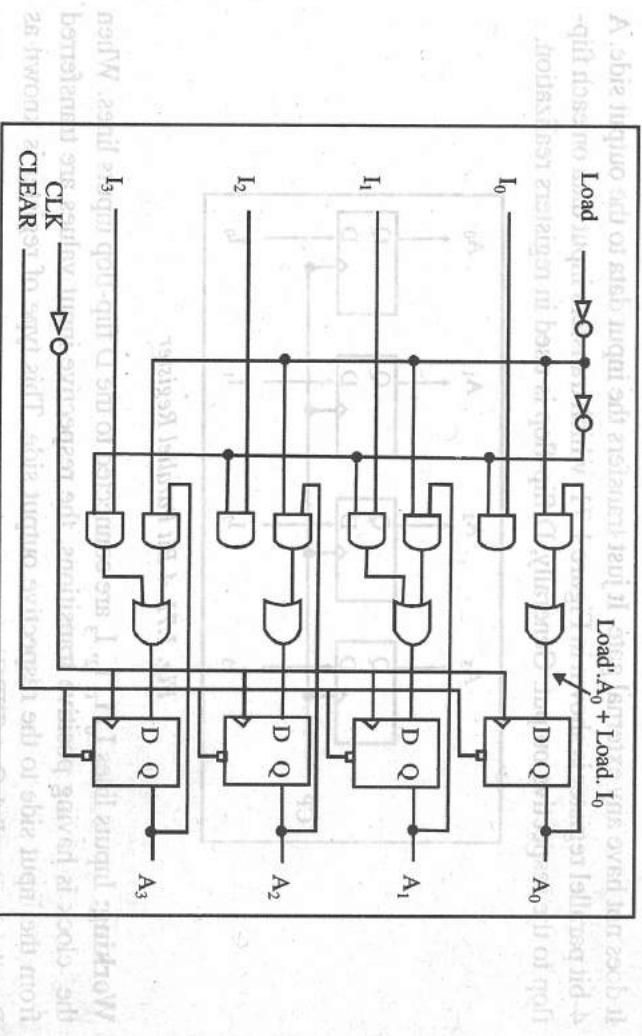


Fig. 1.73 Complete Circuit Diagram for 4-Bit Register with Parallel Loading Feature

1.13.1 Parallel Registers

In this type of register, data inputs can be shifted either in or out of the register in parallel. Also, in this register, there is no interconnection between successive flip-flops 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. A simple 4-bit parallel-in-parallel-out shift register using D flip-flops is shown in Figure 1.74. Here the parallel inputs to be entered should be applied at A, B, C and D inputs which are directly connected to delay (D) inputs of respective flip-flops. Now on applying a clock pulse, these inputs are entered into the register and are immediately available at the outputs Q₁, Q₂, Q₃, and Q₄.

IC 74195—4-Bit Serial/Parallel-In and Serial/Parallel-Out Shift Registers

Integrated Circuit or IC 74195 is a 4-bit Transistor-Transistor Logic Medium Scale Integration or TTL MSI having both serial/parallel-in and serial/parallel-out capability.

The pinout diagram of IC 74195 is shown in Figure 1.75. Since this IC also has a serial input, it can be used for serial-in-serial-out and serial-in-parallel-out operation. This IC can be used for parallel-in-serial-out operation by using Q_D as the output.

NOTES

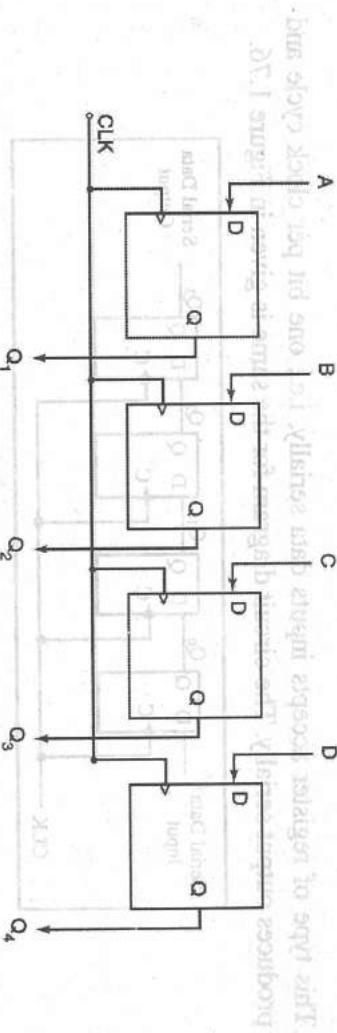


Fig. 1.74 A 4-Bit Parallel-In-Parallel-Out Shift Register

When the SH/LD input is LOW, the data on the parallel inputs, i.e., A, B, C and D, are entered synchronously on the positive transition of the clock. When SH/LD is HIGH, stored data will shift right (Q_A to Q_D) synchronously with the clock. Let J and K be the serial data inputs to the first stage of the register (Q_A); Q_D can be used for getting a serial output data. The active LOW clear is asynchronous.

There are a number of 4-bit, parallel-input-parallel-output shift registers available since they can be conveniently packaged in a 16-pin DIP. An 8-bit register can be created by connecting two 4-bit registers in series. ICs 74174, 74178, 74198 and 7495 are parallel-in-parallel-out registers.

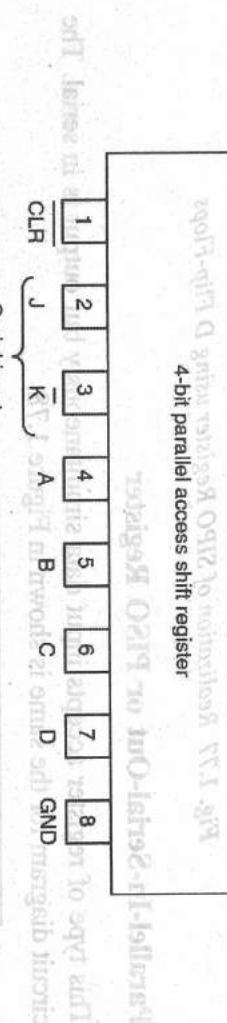
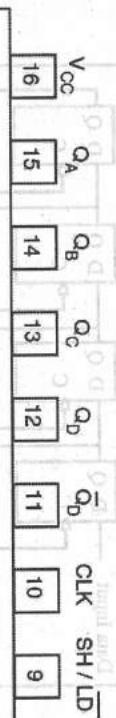


Fig. 1.75 Pinout Diagram of IC 74195 Shift Register

1.13.2 Shift Registers

A register can also be used to provide data movements. Each stage in a shift register represents one bit of storage and it has shifting capability. There are four basic types of shift registers. They are as follows:

- Serial-In-Serial-Out (SISO)
- Serial-In-Parallel-Out (SIPO)
- Parallel-In-Serial-Out (PISO)
- Parallel-In-Parallel-Out (PIPO)

The 16-bit parallel-in-parallel-out IC 74195 is shown in Figure 1.75.

In addition, we can have shift right, shift left, circular right and circular left. The details for the above mentioned shift registers are given as follows:

Serial-In-Serial-Out or SISO Registers

NOTES

This type of register accepts inputs data serially, i.e., one bit per clock cycle and produces output serially. The circuit diagram for the same is given in Figure 1.76.

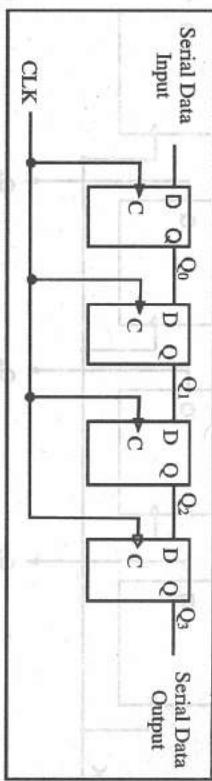


Fig. 1.76 Realization of SISO Register using D Flip-Flops

Working: Input is connected to the first flip-flop D input. On each clock transition, values on the D input get transferred to the respective outputs. After 4 clock transition, the input data starts appearing from the last flip-flop output.

Serial-In-Parallel-Out or SIPO Register

This type of register accepts inputs data serially, i.e., one bit per clock cycle and outputs of all the stages are available simultaneously. The circuit diagram for the same is shown in Figure 1.77.

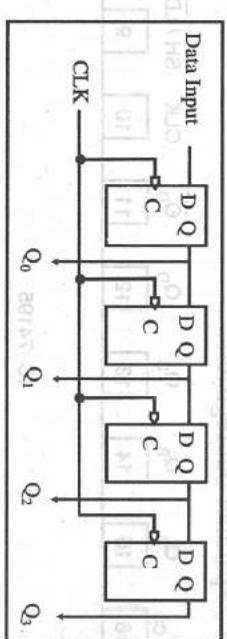


Fig. 1.77 Realization of SIPO Register using D Flip-Flops

Parallel-In-Serial-Out or PISO Register

This type of register accepts input data simultaneously but output is in serial. The circuit diagram for the same is shown in Figure 1.78.

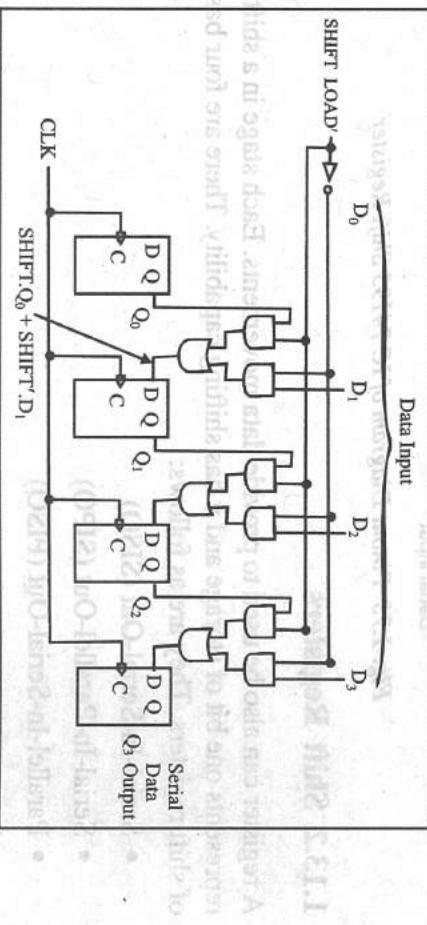


Fig. 1.78 Realization of PISO Register using D Flip-Flops and Basic Logic Gates

Working: First all the input data are loaded simultaneously with the help of control signal **Shift/Load'**. When **Shift/Load' = 0**, the input data D_0, D_1, D_2, D_3 are loaded to the respective D flip-flop inputs. When **Shift/Load' = 1**, it is acting as a shift register, which transfers the previously loaded data to the next stage serially and on each clock cycle, the input data starts coming out from the last flip-flop stage.

Parallel-In-Parallel-Out or PIFO Register

This type of register accepts inputs data simultaneously and output is also coming out parallel. The circuit diagram for the same is shown in Figure 1.79.

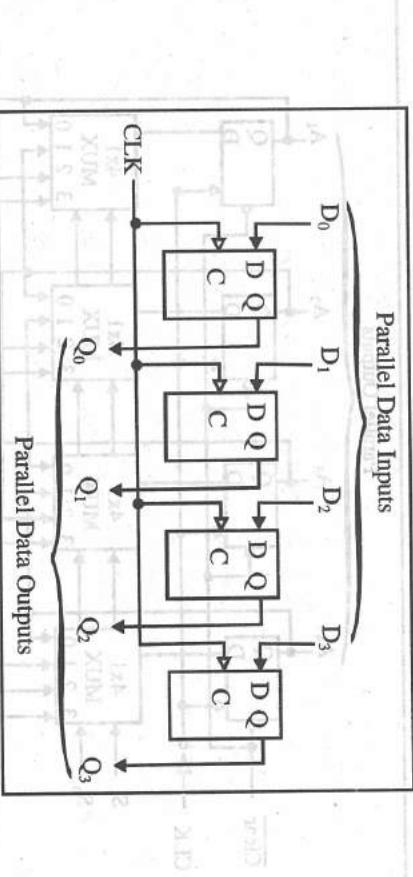


Fig. 1.79 Realization of PIFO Register using D Flip-Flops

Bi-Directional Shift Register

In this type of register, data can be shifted in either right or left direction by using control signal. The circuit diagram is shown in Figure 1.80.

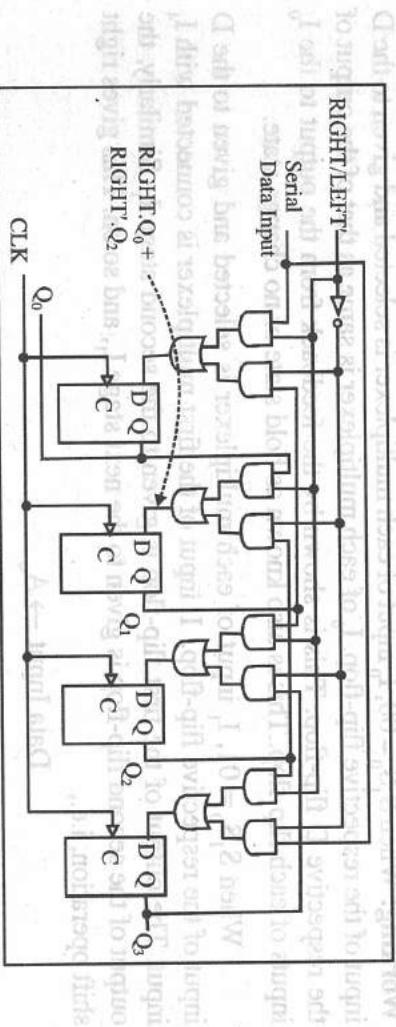


Fig. 1.80 Bi-Directional Shift Register

Working:

When **Right/Left' = 0**, means data should be transferred to the left direction.

When **Right/Left' = 1**, means data should be transferred to the right direction.

A bi-directional shift register can also be designed with parallel load. The following feature can be incorporated:

NOTES

Mode	Control	Register Operation
$S_1 = 1, S_0 = 0$	$S_0 = 0$	No Change
$S_1 = 0, S_0 = 0$	$S_0 = 0$	Shift Right
$S_1 = 1, S_0 = 1$	$S_0 = 1$	Shift Left
$S_1 = 0, S_0 = 1$	$S_0 = 0$	Parallel Load

The complete circuit diagram is shown in Figure 1.81.

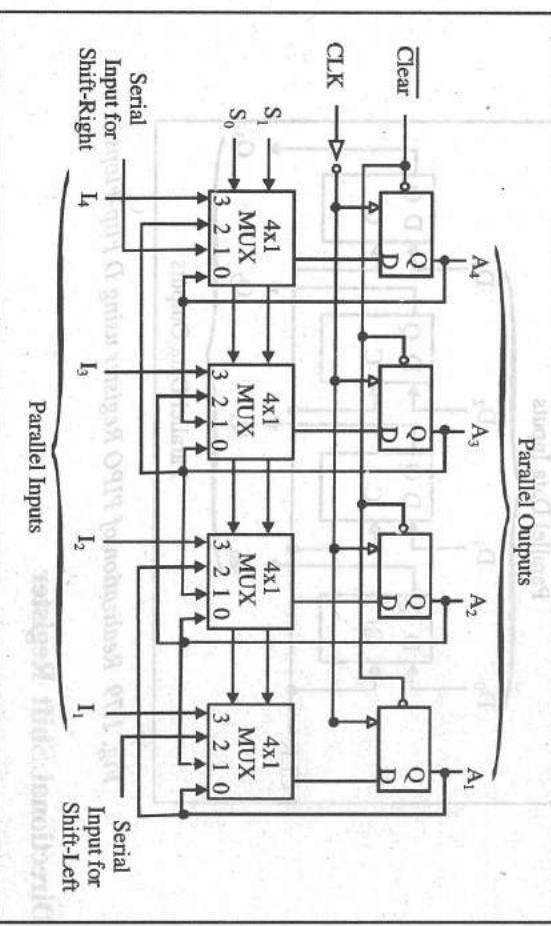


Fig. 1.81 Bi-Directional Shift Register with Parallel Loading Feature

Working: When $S_1 S_0 = 00$, I_0 input of each multiplexer is selected and given to the D input of the respective flip-flop. I_0 of each multiplexer is same as that of the output of the respective D flip-flop. This is shown by the feedback from the output to the I_0 inputs of each flip-flop. This is also known as hold state or no change state.

When $S_1 S_0 = 01$, I_1 input of each multiplexer is selected and given to the D input of the respective flip-flop. I_1 input of the first multiplexer is connected with I_4 input. The output of the first flip-flop is given to the second stage I_1 . Similarly, the output of the second flip-flop is given to the next stage I_1 , and so on. This gives right shift operation, i.e.,

Data Input $\rightarrow A_4$

$A_4 \rightarrow A_3$

$A_3 \rightarrow A_2$

$A_2 \rightarrow A_1$

When $S_1 S_0 = 10$, I_2 input of each multiplexer is selected and given to the D input of the respective flip-flop. I_2 input of the first multiplexer is connected with A_3 output. Similarly, I_2 input of the second multiplexer is connected with A_2 output and so on. This gives left shift operation, i.e.,

Data Input $\rightarrow A_1$

$A_1 \rightarrow A_2$

Digital logic design: Using logic and sequential circuits

$A_2 \rightarrow A_3$, $A_3 \rightarrow A_4$, $A_4 \rightarrow A_5$, $A_5 \rightarrow A_6$

When $S_1 S_0 = 11$, I_3 input of each multiplexer is selected and given to the D input of the respective flip-flop. Since external inputs are connected at I_3 pin of each multiplexer, so on clock transition, these inputs will be loaded to the flip-flop outputs. This gives parallel loading operation.

0	0	0	0	0	0	0
1	0	0	0	1	0	0
0	1	0	0	0	1	0
0	0	1	0	0	0	1
1	1	1	0	1	1	0
1	0	0	1	0	0	1
1	1	0	1	0	1	0

1.14 COUNTERS

It is a sequential circuit that cycles through a sequence of states. For example, a 3-bit binary counter means that it follows the binary sequence from 000 to 111 and then back to 000. An n-bit binary counter counts from 0 to 2^{n-1} and it uses n flip-flop.

Figure 1.82 shows the state diagram and Table 1.33 summarizes the state table for a 3-bit binary counter.



Fig. 1.82 State Diagram for a 3-Bit Binary Counter

Table 1.33 State Table for a 3-Bit Binary Counter

Present State	Next State
$A_2 A_1 A_0$	$A_2 A_1 A_0$
0 0 0	0 0 0
0 0 1	0 1 0
0 1 0	1 0 0
0 1 1	1 1 0
1 0 0	0 0 1
1 0 1	0 1 1
1 1 0	1 0 1
1 1 1	0 0 0

NOTES

Design with T Flip-Flop: The complete state table with the flip-flop inputs is given in Table 1.34.

NOTES

Present State			Next State			Flip-Flop Inputs		
A_2	A_1	A_0	A_2^*	A_1^*	A_0^*	TA_2	TA_1	TA_0
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	1	1
1	1	1	0	0	0	1	1	1

Use K-map to find the Boolean expression for the flip-flop inputs:

$$\begin{aligned}
 A_2 &= \overbrace{\begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 0 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array}}^{\text{A}_0 = 0, \text{A}_1 = 1} \\
 A_2 &= \overbrace{\begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 0 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array}}^{\text{A}_0 = 1, \text{A}_1 = 0} \\
 TA_2 &= A_1, A_0 \\
 TA_1 &= A_0 \\
 TA_0 &= 1
 \end{aligned}$$

Circuit Diagram

Figure 1.83 illustrates the realization of a 3-bit counter using T flip-flop.

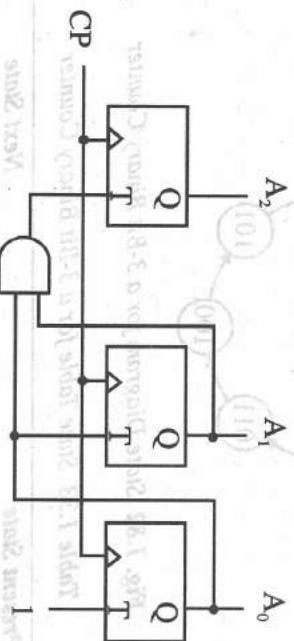


Fig. 1.83 Realization of a 3-Bit Counter using T Flip-Flop

Example 1.77: Design a counter for counting non-binary sequence given as follows using J-K flip-flops.

$000 \rightarrow 001 \rightarrow 010 \rightarrow 100 \rightarrow 101 \rightarrow 110$ and back to 000

Solution: Figure 1.84 illustrates state diagram and Table 1.35 summarizes state table for the given example.

Present State	Flip-Flop Inputs			NOTES									
	A	B	C										
000	0	0	0	0	0	1	0	x	0	x	1	x	x
001	0	0	1	0	1	0	0	0	x	1	x	x	1
010	0	1	0	1	0	0	1	x	x	1	0	x	x
011	0	1	1	0	0	1	1	0	1	x	1	0	x
100	1	0	0	0	0	0	0	0	0	0	0	0	x
101	1	0	1	0	0	1	0	1	x	x	1	0	x
110	1	1	0	0	1	0	0	1	x	x	1	0	x
111	1	1	1	0	0	1	1	0	0	0	0	0	x

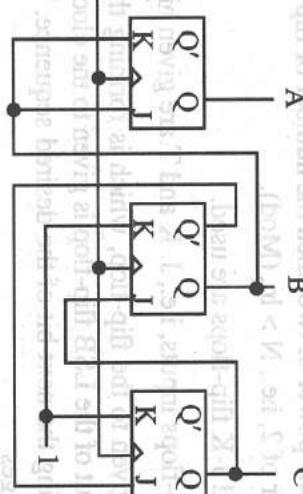
Fig. 1.84 State Diagram

Figure 1.84 shows the state diagram for a three-bit binary counter using J-K flip-flops. The states are represented by three-bit binary numbers: 000, 001, 010, 011, 100, 101, 110, and 111. Transitions are labeled with the corresponding J-K flip-flop inputs (A*, B*, C*) required to move from one state to another.

Note: In the above counting sequence, unused states 011 and 111 are treated as *don't care*. Therefore, the next state transition and the entries for the J-K flip-flops are also treated as *don't care* for these states.

Circuit Diagram

Figure 1.85 illustrates the realization of counting sequence using J-K flip-flop.

**Fig. 1.85 Realization of Counting Sequence using J-K Flip-Flop**

Since the sequence consists of two unused states, namely, 011 and 111, so self-correcting realization leads to the state diagram as shown in Figure 1.86.

Figure 1.86 shows the state diagram for a three-bit binary counter using J-K flip-flops. The states are 000, 001, 010, 011, 100, 101, 110, and 111. Transitions are labeled with the corresponding J-K flip-flop inputs (J and K) required to move from one state to another. The sequence starts at 000 and goes through 001, 010, 011, 100, 101, 110, and ends at 111. The state 011 is a don't care state, and the state 111 is a self-correcting state.

Fig. 1.86 State Diagram

There are two types of counters:

- Synchronous parallel counter
- Asynchronous ripple counter

In the first type of counter, clock signal is applied to all the flip-flops simultaneously, so it is called parallel counter. In a ripple counter, some flip-flop outputs are used as source of clock signal for other flip-flops.

The method for the synchronous counter is same as that of state diagram realization. The counters discussed above are synchronous counter.

NOTES

1.14.1 Ripple Counters

In this type of counter, the flip-flops do not change states at exactly the same time, as they do not have a common clock pulse. Also known as ripple counters, the input clock pulse ‘ripples’ through the counter and hence, it has cumulative delay, which is the drawback with the ripple counter.

In general, a ripple counter is realized by cascading flip-flops. If we want to count from 0 to 3, i.e., from 00 to 11 having four states, then the number of flip-flops will be 2. For counter with 2^N states, we need N flip-flops to be cascaded. The number of state through which the counter goes through is also known as MOD number. Certain point must be kept in mind while designing an asynchronous counter. They are as follows:

Point 1: Decide the number of flip-flops. If MOD is of the power of two, then the number of flip-flops will be $N = \ln_2(\text{Mod})$.

When MOD is not the power of two, then the number of flip-flops will be an integer nearest to the power of 2, i.e., $N > \ln_2(\text{Mod})$.

Point 2: Only T and J-K flip-flops are used.

Point 3: All the flip-flops inputs, i.e., J, K and T are given high.

Point 4: Clock is given to the flip-flop, which is forming the LSB of the counting sequence. The output of the LSB flip-flop is given to the clock input of the next flip-flop, which is forming the next bit of the desired sequence. This will repeat for the subsequent next stages.

Example 1.18: 2-bit binary counter, i.e., counting from 00 \rightarrow 01 \rightarrow 10 \rightarrow 11 and back to 00.

Solution:

Step 1: Decide the number of flip-flops. Since the MOD number is 4, so the number of flip-flops required will be 2 (refer Figure 1.87). Take two J-K flip-flops and connect logic 1 to all the J-K inputs of both the flip-flops.

Step 2: Apply clock to first flip-flop, which will form the LSB bit of the counting sequence.

Step 3: Cascade the flip-flop such that the first flip-flop output will be connected to the second stage clock.

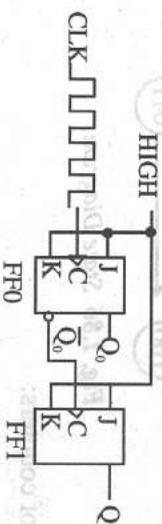


Fig. 1.87 MOD-4 Realization using Positive-Triggered J-K Flip-Flop

Working: Since the flip-flops are positive-triggered, so the first flip-flop, i.e., Q_0 will be changing states every time the clock is having positive transition. The complemented output, Q'_0 of the first flip-flop is given to the next stage flip-flop clock input. Therefore, when Q'_0 is changing from 0 to 1, then the state of the second flip-flop will change.

Assuming that flip-flop is in the initial state 00, i.e., Q_0 and Q_1 are 0. When the first positive clock transition occurs, Q_0 , toggled since $J = K = 1$ and hence, Q'_0 sees transition from 1 to 0. Since Q'_0 is connected to the next stage flip-flop, which sees negative transition, so the state of the next stage flip-flop will not change.

When the clock sees second time positive transition, Q_0 will again toggle its value and will become 0. This will give transition on Q'_0 from 0 to 1, which will be positive clock transition for the next stage flip-flop and this will cause the output state of the next flip-flop to toggle. This will give $Q_1 = 1$.

When the clock sees third time positive transition, Q_0 will again toggle its value and will become 1. This will give transition on Q'_0 from 1 to 0, which will be negative clock transition for the next stage flip-flop and hence, the output state of the next flip-flop will not see any change. This will maintain the value of $Q_1 = 1$.

When the clock sees fourth time positive transition, Q_0 will again toggle its value and will become 0. This will give transition on Q'_0 from 0 to 1, which will be positive clock transition for the next stage flip-flop and this will cause the output state of next flip-flop to toggle. Since $Q_1 = 1$, so the next state will be $Q_1 = 0$.

So after four clock transition, the circuit is again back to the original starting state, i.e., $Q_1Q_0 = 00$. The waveform presentation is shown in Figure 1.88.

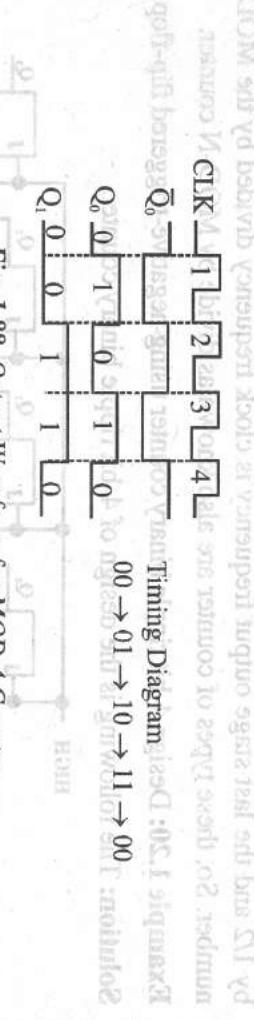


Fig. 1.88 Output Waveform for MOD-4 Counter

It can be seen that the LSB bit of state is changing on each positive clock transition since its clock is directly being connected to the external clock. However, the state of the second-stage flip-flop will change only when Q'_0 is changing from 0 to 1.

Example 1.19: What is 3-bit ripple binary counter?

Solution: This is also known as MOD-8 ripple counter (refer Figure 1.89).

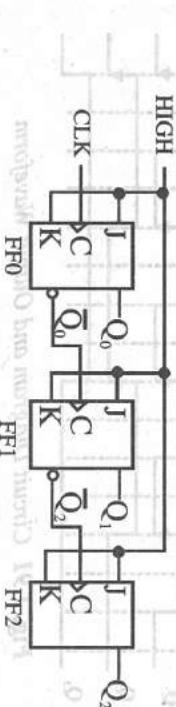
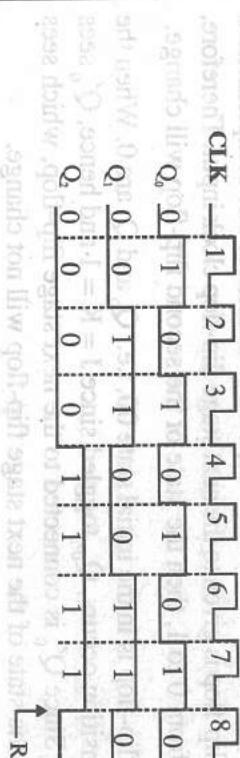


Fig. 1.89 Circuit Diagram for MOD-8 Counter. Q₀: indicates the least significant bit of state. Q₁: indicates the middle significant bit of state. Q₂: indicates the most significant bit of state. The circuit has three stages. The first stage is a J-K flip-flop with J=K=1. The second stage is a J-K flip-flop with J=K=1. The third stage is a J-K flip-flop with J=K=1. The outputs of the three stages are Q₀, Q₁, and Q₂ respectively.

NOTES

NOTES**Fig. 1.90 Output Waveform for MOD-8 Ripple Counter**

Explanation: Q_0 will change state on each positive transition of clock. Q_1 will change state when Q_0 is changing from 1 to 0 since this will cause Q'_0 to change from 0 to 1. Similarly, Q_2 will change state when Q_1 is changing from 1 to 0 since this will cause Q'_1 to change from 0 to 1.

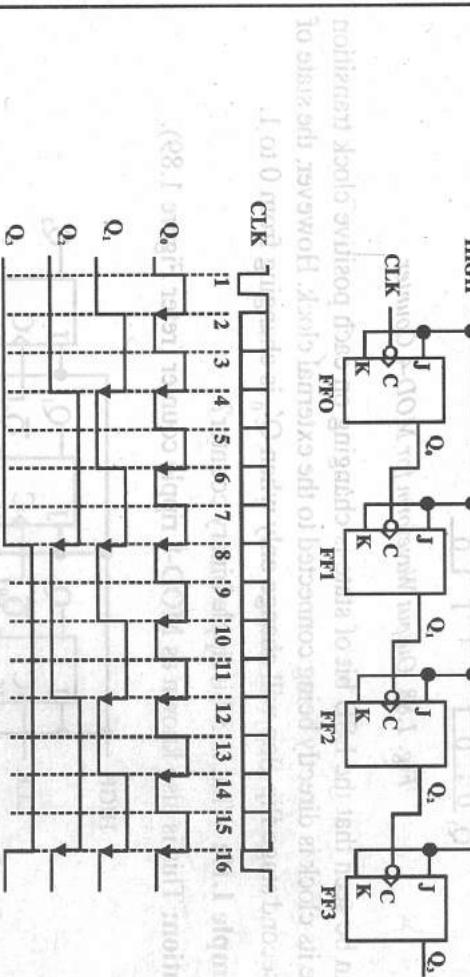
From the waveform as shown in Figure 1.90, it can be seen that Q_0 has half the frequency of clock signal, CLK. Similarly, Q_1 has half the frequency of Q_0 signal, which acts as a clock for the flip-flop Q_1 . In other words, we can say that Q_1 has frequency equal to 1/4 clock that is connected to the first stage.

Similarly Q_2 has frequency 1/2 with respect to Q_0 and hence 1/8 times clock signal that is connected to the first stage.

In other words, we can say that each stage divides the clock signal frequency by 1/2 and the last stage output frequency is clock frequency divided by the MOD number. So, these types of counter are also known as divide by MOD-N counter.

Example 1.20: Design 4-bit ripple binary counter using negative-triggered flip-flop.

Solution: The following is the design of 4-bit ripple binary counter.

**Fig. 1.91 Circuit Diagram and Output Waveform**

Explanation: Q_0 will change state on each negative transition of clock. Q_1 will change state when Q_0 is changing from 1 to 0 since this will cause negative transition on the next stage flip-flop (refer Figure 1.91). Similarly, Q_2 will change state when Q_1 is changing from 1 to 0 and so does Q_3 .

1.14.2 Synchronous Counters

All the flip-flops are clocked at the same time by a common clock pulse. We can design these counters using the sequential logic design process discussed already. For example, 2-bit synchronous binary counter using T flip-flop is shown in Figure 1.92.

NOTES

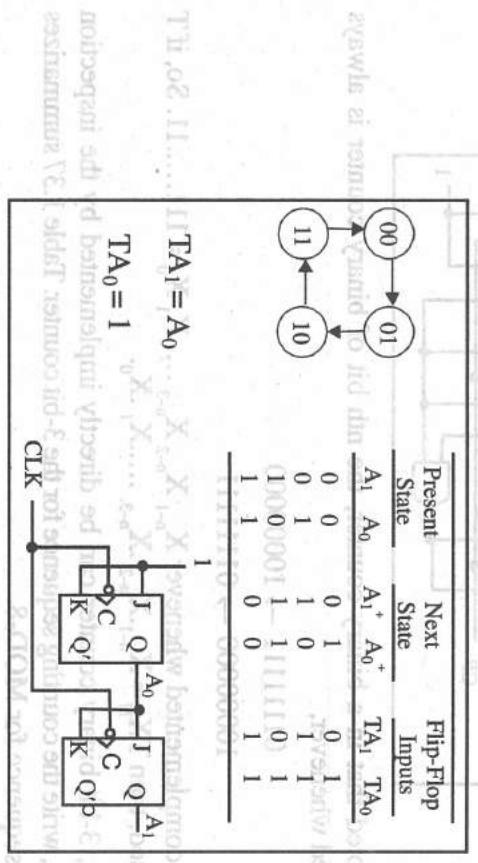


Fig. 1.92 MOD-4 State Diagram, State Table, Input Equation and Circuit Diagram

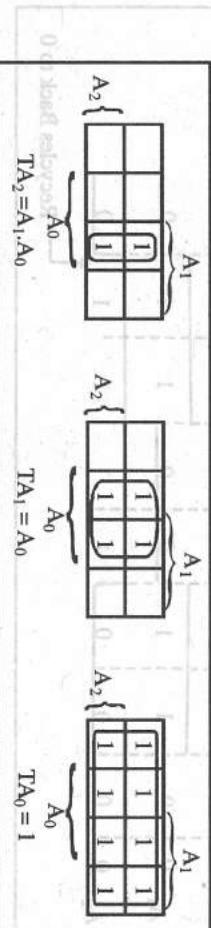
Example 1.21: Draw a state table for 3-bit binary counter, i.e., counting from 000 to 111 and then back to 000.

Solution: Table 1.36 summarizes the state table for 3-bit binary counter.

Table 1.36 State Table

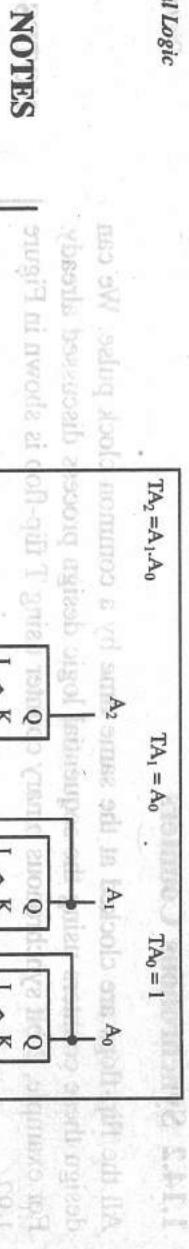
Present State	Next State				Flip-Flop Inputs				
	A ₂	A ₁	A ₀	A ₂ ⁺	A ₁ ⁺	A ₀ ⁺	TA ₂	TA ₁	TA ₀
000	0	0	0	0	0	1	0	0	1
001	0	0	1	0	1	0	0	1	1
010	0	1	0	0	1	1	0	0	1
011	0	1	1	0	0	1	1	1	1
100	1	0	0	1	0	0	0	0	1
101	1	0	1	1	0	1	1	0	1
110	1	1	0	0	1	0	0	1	1
111	1	1	1	1	1	0	1	1	0
000	0	0	0	0	1	0	0	1	1

Design: Find the Boolean expression of flip-flop inputs.



Circuit Diagram: Here T flip-flop is realized with shorting J-K input of J-K flip-flop.

base 0 = \bar{A}_0 andt gungnato ai \bar{A} to state 011 .mientan kook das no gungnato seko binerai $(\bar{I} = {}_0 A \wedge I = {}_1 A)$ to $(\bar{I} = {}_0 A$



$T_{A_2} = A_1 \cdot A_0$ $T_{A_1} = A_0$ $T_{A_0} = 1$

Hence, X_n is complemented whenever $X_{n-1} X_{n-2} X_{n-3} \dots X_1 X_0 = 111\dots11$. So, if T flip-flop is used, then $T_{X_N} = X_{n-1} \cdot X_{n-2} \cdot X_{n-3} \dots \cdot X_1 \cdot X_0$.

For example, 3-bit binary counter can be directly implemented by the inspection method. First, write the counting sequence for the 3-bit counter. Table 1.37 summarizes the counting sequence for MOD-8.

Table 1.37 Counting Sequence for MOD-8

Present State			Next State							
A ₂	A ₁	A ₀	Q ₀	Q ₁	Q ₂	Q ₃	Q ₄	Q ₅	Q ₆	Q ₇
0	0	0	0	1	0	0	1	0	0	1
0	0	1	0	0	1	0	0	1	0	0
0	1	0	1	0	0	1	0	0	1	0
0	1	1	1	0	1	0	0	1	0	0
1	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	0	0	0	0	0	0
1	1	0	1	1	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0

The waveform for the counting sequence is shown in Figure 1.93.

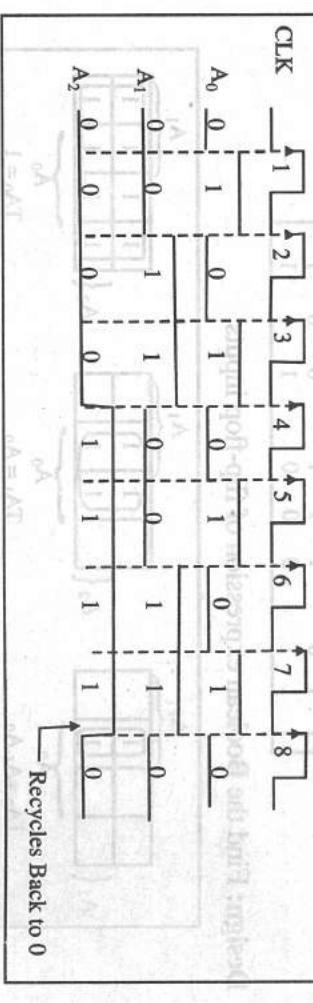


Fig. 1.93 Output Waveform for MOD-8 Counting Sequence

Now from the above waveform, it can be seen that the state of the LSB, A_0 is changing on each clock transition. The state of A_1 is changing when ($A_2 = 0$ and $A_0 = 1$) or ($A_2 = 1$ and $A_0 = 1$) and clock goes for the second positive transition.

NOTES

It is to be noted that in a binary counter, the nth bit of binary counter is always complemented whenever,

$$0111111 \rightarrow 1000000$$

$$1000000 \rightarrow 0111111$$

Similarly, the state of A_2 is changing when both A_0 and A_1 are 1 and clock goes for the second positive transition. T is being set at rising edge of the clock.

So the flip-flop input for the first T flip-flop will be $T_0 = 1$. The T_1 for the second flip-flop will be $= A'_2 \cdot A_0 + A_2 \cdot A_0 = A_0$, i.e., the output of the first flip-flop. Similarly, T_2 for the third flip-flop will be the AND of outputs of the first flip-flop, i.e., $A_0 \cdot A_1$, since when both A_0 and A_1 are 1, then the state of A_2 is changing on clock transition. So we can write:

$$TA_0 = 1 \quad TA_1 = A_0 \quad TA_2 = A_1 \cdot A_0$$

Example 1.22: Design a 4-bit synchronous binary counter using T flip-flop.

Solution: The inputs of flip-flop, assuming that A_3 is the MSB and A_0 is the LSB will be:

$$\begin{aligned} TA_3 &= A_2 \cdot A_1 \cdot A_0 \\ TA_2 &= A_1 \cdot A_0 \\ TA_1 &= A_0 \\ TA_0 &= 1 \end{aligned}$$

Circuit Diagram

Figure 1.94 illustrates the realization of MOD-16 synchronous binary counter.

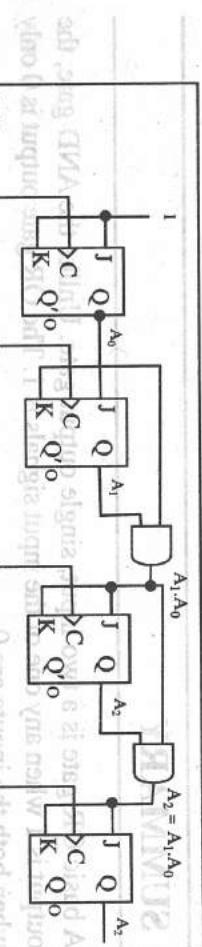


Fig. 1.94 Realization of MOD-16 Synchronous Binary Counter

Table 1.38 summarizes the state table for MOD-10 counter.

Table 1.38 Counting Sequence for MOD-10 Synchronous Counter

Clock Pulse	Q_3	Q_2	Q_1	Q_0
Initially	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	1	0	0
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	1	0	1	0
8	1	0	0	0
9	1	0	0	1
10 (Recycle)	0	0	0	0

as before. It can be seen that the state of Q_0 is changing on each clock transition, so the input of the first flip-flop should be 1, i.e., $T_0 = 1$. The state of Q_1 is changing when $\{Q_3 = 0, Q_2 = 0, Q_0 = 1\}$ or $\{Q_3 = 0, Q_2 = 1, Q_0 = 1\}$.

NOTES

On combining the conditions, we can have the $Q'_3 Q'_2 Q'_0 + Q'_3 Q'_2 Q_0 = Q'_3 Q_0$. This gives you the value of logic that is to be applied at T_1 input of flip-flop.

The state of Q_2 is changing when $Q_1 = 1, Q_0 = 1$, so Q_1, Q_0 gives you the value of logic that is to be applied at T_2 input of flip-flop.

The state of Q_3 is changing when $Q_2 = 1, Q_1 = 1, Q_0 = 1$; so the Boolean expression $Q_2 Q_1 Q_0 + Q_3 Q_0$ gives you the value of logic that is to be applied at T_3 input of flip-flop.

$$T_0 = 1 \quad T_1 = Q'_3 Q_0 \quad T_2 = Q_1 Q_0 \quad T_3 = Q_2 Q_1 Q_0 + Q_3 Q_0$$

The circuit diagram for the same is shown in Figure 1.95.

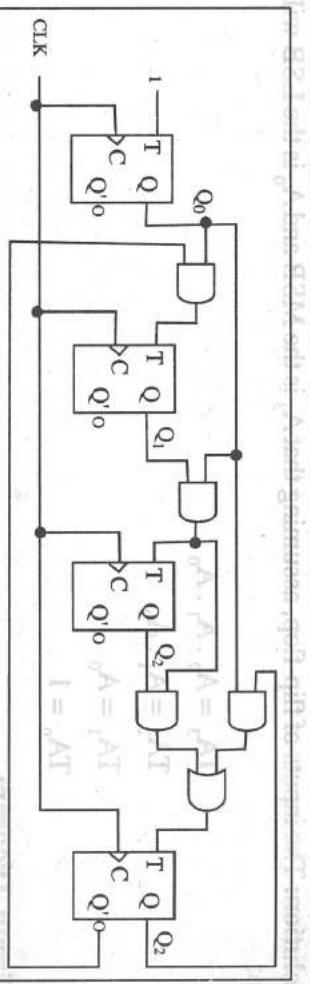


Fig. 1.95 Realization of MOD-10 Synchronous Binary Counter

1.15 SUMMARY

- 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.
 - A Boolean function is an expression formed with binary variables, the two binary operators OR and AND, the unary operator NOT, and the equal and parenthesis signs. Its result is also a binary value. The general usage is ‘.’ for AND, ‘+’ for OR and ‘’ for NOT.
 - For a function F, the complement of this function F' is obtained by interchanging 1 with 0 and vice versa in the function's output values.
 - Product-Of-Sum or POS expression is a sum term or a logical product (AND) of several sum terms.
 - In a combinational circuit, each output depends entirely on the immediate (present) inputs to the circuit.
 - An adder is a combinational circuit which performs the addition of two numbers to produce sum and carry as output. A half adder is a combinational circuit which performs the arithmetic addition of two binary bits. A full adder is a combinational circuit which performs the arithmetic addition of three binary bits (two significant bits and a previous carry).
 - A Karnaugh map or K-map is an abstract form of Venn diagram organized as a matrix of squares where each square represents a Minterm.
- Check Your Progress**
18. What is a T flip-flop?
 19. Name the inverters which are placed at each input in S-R flip-flop.
 20. State the meaning of ‘loading a register’.
 21. Name the types of parallel-in-parallel-out registers.
 22. Name the two types of counters.
 23. What is a MOD number?

- Codes are frequently used to represent entities. These codes can be identified or decoded using a decoder. A 2×4 decoder is a circuit, which has two input lines and generates 4 output lines, represent four codewords or Minterms.
- Encoding is the converse of decoding. It is implemented with OR gates.

NOTES

- A demultiplexer or demux is a device that takes a single input signal and selects one of many data output lines which is connected to the single input. A multiplexer is often used with a complementary demultiplexer on the receiving end.
- A multiplexer is a circuit which has a number of input lines and selection lines with one output line. It is also known as a data selector.
- J-K flip-flop is same as that of S-R flip-flop but it has feedback Q and Q' to the NAND gate to which S and R inputs are connected along with clock pulse.
- The flip-flops store information while the control circuitry decides when and how new information is transferred into the register.
- Loading a register means transferring new information into the register and it requires a load control input.
- A register can also be used to provide data movements. Each stage in a shift register represents one bit of storage and it has shifting capability.
- In ripple counters, the input clock pulse ‘ripples’ through the counter and hence, it has cumulative delay.

1.16 KEY TERMS

- **NOT gate:** Logic gate which has only one input and one output
- **Boolean function:** An expression formed with binary variables
- **Truth table:** A mathematical table used in logic specifically in connection with Boolean algebra and Boolean functions
- **Product term:** A logical product (AND) of several literals
- **Sequential logic:** A type of logic circuit whose output depends not only on the present input but also on the history of the input
- **Binary adder:** The digital circuit that generates the arithmetic addition of two binary numbers of any length
- **Karnaugh-map:** A method to simplify Boolean algebra expressions
- **Decoder:** A device which does the reverse operation of an encoder
- **Encoder:** A device that converts information from one format or code to another
- **Demultiplexer:** A device that takes a single input signal and selects one of many data output lines which is connected to the single input

1.17 ANSWERS TO ‘CHECK YOUR PROGRESS’

1. The basic NOT gate has only one input and one output. The output is always the opposite or negation of the input.

NOTES

5. A Boolean expression is an algebraic statement containing Boolean variables and operators.
6. A truth table is a mathematical table which includes logical operations and consists of every possible combination of inputs and its corresponding outputs.
7. A variable in its own or in its complemented form is known as a literal.
8. In digital electronics, the two broad categories of logic circuits are as follows:
- Combinational Circuit
 - Sequential Circuit
9. A practical synchronous sequential logic system uses fixed amplitudes, such as voltage levels for the binary signals.
10. An adder is a combinational circuit, which performs the addition of two numbers to produce sum and carry as output.
11. Algebraic simplification aims to minimize the number of literals and terms.
12. A Karnaugh map (K-map) is an abstract form of Venn diagram, organized as a matrix of squares, where each square represents a Minterm.
13. A 2×4 decoder is a circuit, which has two input lines and generates 4 output lines, which represent four codewords or Minterms.
14. Encoding is the converse of decoding.
15. When more than one input can be active, then the priority encoder is used.
16. A demultiplexer or demux is a device that takes a single input signal and selects one of many data output lines which is connected to the single input.
17. A multiplexer is a circuit, which has a number of input lines and selection lines with one output line.
18. It is a single input version of J-K flip-flop formed by tying both the inputs of J-K.
19. Two inverters NAND 1 and NAND 2 are placed at each input in S-R flip-flop.
20. Loading a register means transferring new information into the register and it requires a load control input.
21. ICs 74174, 74178, 74198 and 7495 are the various types of parallel-in-parallel-out registers.
22. The two types of counters are synchronous parallel counter and as synchronous parallel counter.
23. The number of state through which the counter goes through is also known as MOD number.

1.18 QUESTIONS AND EXERCISES

Digital Logic

Short-Answer Questions

1. Why is AND gate used?
2. What is Boolean algebra?
3. What is the significance of precedence of operators?
4. What are Minterms?
5. Write the function of combinational circuit.
6. What is half adder?
7. What is Karnaugh map?
8. What is 3×8 decoder?
9. Write the function of priority encoder.
10. Why is demultiplexer used?
11. Write the function of D flip-flop.
12. Why bubble is used in flip-flops diagrams?
13. What are registers?
14. Name the types of shift registers.
15. What are counters?
16. Why synchronous counters are used?

Long-Answer Questions

1. Explain the logic gates with the help of illustrations.
2. Discuss the concept of Boolean algebra with the help of axioms and examples.
3. How Boolean functions are implemented? Explain with the help of examples.
4. Explain the functions of combinational and sequential circuits with the help of illustrations and examples.
5. Describe adder and its types with the help of examples.
6. Discuss the Boolean simplifications process with the help of Karnaugh map.
7. Reduce the Boolean expression $F(A, B, C, D) = \sum m(2, 3, 4, 5, 7, 8, 10, 13, 15)$.
8. Find the simplest POS expression for the following function:
$$F(A, B, C, D) = A \cdot B \cdot C + B' \cdot C \cdot D' + A \cdot D + B' \cdot C' \cdot D'$$
9. Explain 4×2 encoder with the help of a block diagram.
10. Discuss demultiplexer and its functions with the help of illustrations.
11. Describe the various types of multiplexer with the help of examples and illustrations.
12. What is flip-flop? Explain its various types with the help of examples.
13. Explain J-K flip-flop with the help of truth table and circuit diagram.

NOTES

1. Discuss the concept of Boolean algebra with the help of illustrations.
2. Explain the logic operations with the help of examples.
3. Explain the significance of precedence of operators.
4. Explain the function of combinational circuit.
5. Explain the function of half adder.
6. Explain the function of Karnaugh map.
7. Explain the function of 3×8 decoder.
8. Explain the function of priority encoder.
9. Explain the function of demultiplexer.
10. Explain the function of D flip-flop.
11. Explain the function of bubble.
12. Explain the function of shift register.
13. Explain the function of registers.
14. Explain the function of shift registers.
15. Explain the function of counters.
16. Explain the function of synchronous counters.

14. Discuss the concept of registers with parallel load.
15. What are parallel registers? Explain with the help of examples.
16. Describe IC 74195 4-bit serial/parallel-in and serial/parallel-out shift register with the help of illustrations.

NOTES

17. Explain ripple counters and its functions with the help of circuit diagrams and examples.

1.19 FURTHER READING

- Tanenbaum, Andrew S. 1999. *Structured Computer Organization*, 4th edition. New Jersey: Prentice-Hall Inc.
- Wilkinson. 1996. *Computer Architecture: Design and Performance*, 2nd edition. Hertfordshire: Prentice-Hall.
- Mano, M. Morris. 1993. *Computer System Architecture*, 3rd edition. New Jersey: Prentice-Hall Inc.
- Stalling, William. 2006. *Computer Organization and Architecture*, 7th edition. New Jersey: Prentice-Hall Inc.
- Hamacher, V.C., Z.G. Vranesic and S.G. Zaky. 2002. *Computer Organization*, 5th edition. New York: McGraw-Hill International Edition.
- Conte, T. M. and C. E. Gimarc. 1995. *Fast Simulation of Computer Architecture*. Boston: Kluwer Academic Publishers.
- Gilmore, M. 1996. *Microprocessors: Principles and Applications*, 2nd edition. New York: McGraw-Hill.

UNIT 2 THE COMPUTER SYSTEM

Structure

2.0 Introduction

2.1 Unit Objectives

2.2 Computer Functions and its Interconnection

2.2.1 Computer Functions

2.2.2 Interconnection Structures

2.2.3 Bus Interconnection

2.3 Memory System Design

2.3.1 Memory Hierarchy and SRAM

2.3.2 Advanced DRAM Organization

2.3.3 Interleaved Memory

2.3.4 Associative Memory

2.3.5 Non-Volatile Memory

2.3.6 Redundant Array of Independent Disks or RAID

2.4 Cache Memory

2.4.1 Cache Memory Principles

2.4.2 Elements of Cache Design

2.4.3 Improving Cache Performance

2.5 Input/Output

2.5.1 Peripheral Devices

2.5.2 External Devices

2.5.3 I/O Modules

2.5.4 Programmed I/O

2.5.5 Interrupt-Driven I/O

2.5.6 Direct Memory Access

2.6 I/O Channels and Processors

2.7 Summary

2.8 Key Terms

2.9 Answers to 'Check Your Progress'

2.10 Questions and Exercises

2.11 Further Reading

NOTES

- Write long and full answers.
- Use bullet points to emphasize important features.
- Explain the concepts in simple terms.
- Explain the significance of the following:

2.0 INTRODUCTION

In the previous unit, you learnt about digital logic. In this unit, you will learn about the computer system. A system of interconnected computers shares a central storage

system and various peripheral devices, such as a printers, scanners or routers. Each computer connected to the system can operate independently but also has the ability to communicate with other external devices and computers. The basic function performed by a computer is execution of a program which consists of a set of instructions stored in memory. The processor does the actual work by executing instructions specified in the program. In the simplest form, instruction processing consists of two steps: the processor reads (fetches) instructions from memory one at a time and executes each instruction. Memory hierarchy starts with a small, expensive and relatively fast unit called the cache, followed by a larger, less expensive and relatively slow main memory unit. Cache and main memory are built using solid-

state semiconductor material typically Complementary Metal Oxide Semiconductor or CMOS transistors. You will also learn about cache memory. The cache is a small, fast memory placed between the CPU and the main memory. 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 I/O system of a computer processes the input information by converting them into computer readable binary form and then displaying the final result as output in user readable form. The peripheral devices of a computer system are known as I/O units. They provide an efficient way of communication between the computer system and the outside world. A computer must have a system to receive information (input) and must be able to communicate results (output) to the external world using keyboard, mouse, monitor and printer. Finally, you will learn about I/O channels and processors. Channels are used to handle the I/O operation of computer system. The term channel is also used for I/O processor that manages the processing of I/O operations. The number of channels connected to the computer system is determined on the basis of application running on the system. Each channel handles the processing of one or more I/O devices connected to it.

2.1 UNIT OBJECTIVES

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

- Understand functions of computer and its interconnection mechanism
- Explain memory system design and hierarchy
- Discuss cache memory and its functions
- Describe input/output interface and devices
- Explain the significance of I/O channels and processors

2.2 COMPUTER FUNCTIONS AND ITS INTERCONNECTION

A system of interconnected computers shares a central storage system and various peripheral devices, such as a printers, scanners or routers. Each computer connected to the system can operate independently but has the ability to communicate with other external devices and computers. An instruction cycle consists of an instruction fetch and operands followed by an interrupt check (if interrupts are enabled). The major computer system components such as processor, main memory, Input/Output or I/O modules are interconnected to exchange data and control signals.

2.2.1 Computer Functions

In its simplest form, a computer consists of five functionally independent components: input, output, memory, arithmetic and logic unit, and control unit as shown in Figure 2.1.

input unit which can be an electromechanical device such as a keyboard or from other computers over digital communication lines. The information received by the computer is either stored in the memory for later reference or used immediately by the Arithmetic and Logic Unit (ALU) for performing the desired operations. Finally, the processed

information in the form of results is displayed through an output unit. The control unit controls all activities taking place inside the computer. The ALU unit along with the control unit are collectively known as the Central Processing Unit (CPU) or processor, and the input and output units are collectively known as the I/O unit.

NOTES

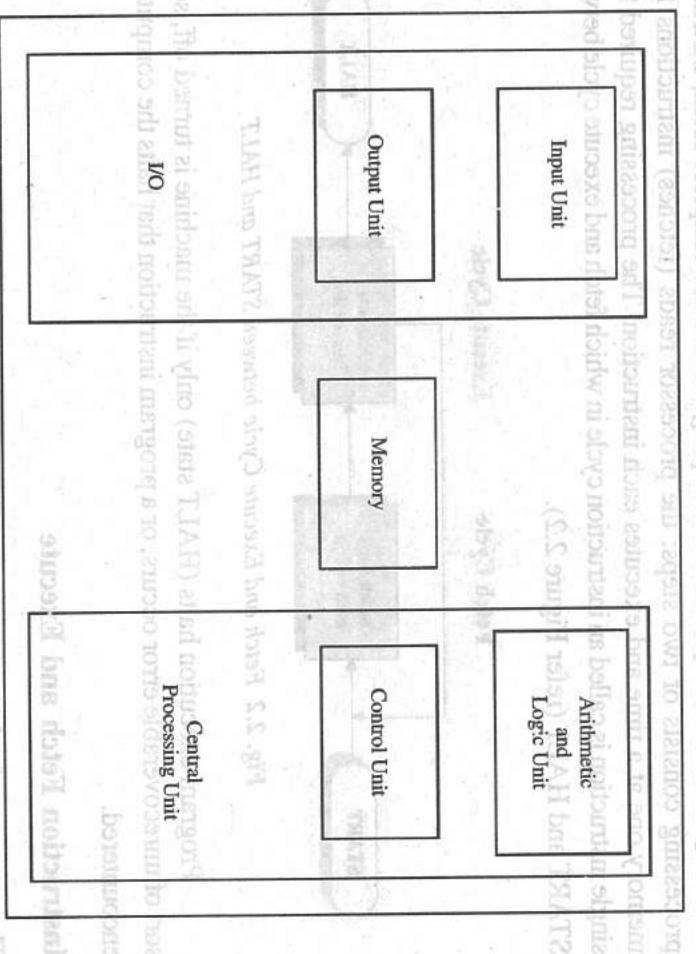


Fig. 2.1 Functional Units of a Computer

- **Input Unit:** A computer accepts input in coded form through an input unit. The keyboard is an input devices. Whenever a key is pressed, the binary code of the corresponding letter or digit is transferred to the memory unit or processor. Other types of input devices are mouse, punch card, joysticks, etc.
- **Memory Unit:** The task of the memory unit is to safely store programs as well as to manage input, output and intermediate data. The two different classes of memory are primary and secondary storage. Primary storage is a very fast memory and contains a large number of semiconductor cells capable of storing one bit of information. A group (of fixed size) of these cells is referred to as words and the number of bits in each word is referred to as word length which typically ranges from 16 to 64 bits. When the memory is accessed, usually one word of data is read or written.
- **Processor Unit:** The processor unit performs arithmetic and other data processing tasks as specified by a program.
- **Control Unit:** It oversees the flow of data among the other units. The control unit retrieves the instructions from a program (one by one) which are safely

at any kept in the memory. For each instruction, the control unit tells the processor to execute the operation marked by the instruction. The control unit supervises the program instructions and the processor manipulates the data as specified by the programs.

NOTES

- **Output Unit:** The output unit receives the result of the computation which is displayed on the screen or printed on paper using a printer.

The basic function performed by a computer is execution of a program which consists of a set of instructions stored in memory. The processor does the actual work by executing instructions specified in the program. In the simplest form, instruction processing consists of two steps: the processor reads (fetches) instructions from memory one at a time and executes each instruction. The processing required for a single instruction is called an instruction cycle in which fetch and execute cycle between START and HALT (refer Figure 2.2).

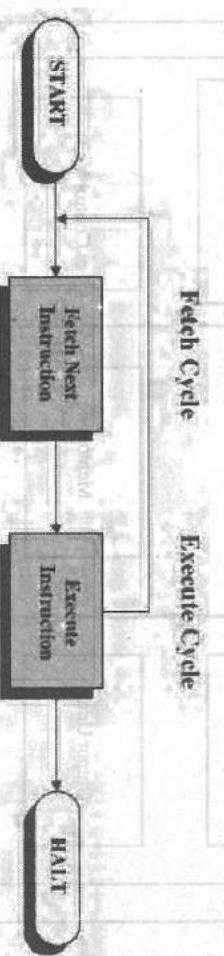


Fig. 2.2 Fetch and Execute Cycle between START and HALT

Program execution halts (HALT state) only if the machine is turned off, some sort of unrecoverable error occurs, or a program instruction that halts the computer is encountered.

Instruction Fetch and Execute

The processor fetches an instruction from memory and the Program Counter (PC) register holds the address of the instruction to be fetched next. The processor increments the PC after each instruction fetch so that it will fetch the next instruction in the sequence. The fetched instruction is loaded into the Instruction Register (IR) in the processor, i.e., the instruction contains bits that specify the action. The processor interprets the instruction and performs the required action. In general, these actions fall into following four categories:

- **Processor Memory:** In this category, data transferred to or from the processor to memory.
- **Processor I/O:** In this category, data transferred to or from a peripheral device by transmitting between the processor and an I/O module.
- **Data Processing:** The processor performs some arithmetic or logic operation on data.
- **Control:** An instruction may specify that the sequence of execution be altered.

2.2.2 Interconnection Structures

The principal characteristic of a multiprocessor is its ability to share a set of main memory and some I/O devices. This sharing is possible through some physical

connections between them which is called the interconnection structure. Some of these schemes are as follows:

The Computer System

- Time-shared common bus
- Multiport memory
- Crossbar switch
- Multistage switching network
- Hypercube interconnection

NOTES

The basic tradeoff in designing the interprocessor communication network is between the speed of communication and the cost: high speed requires lots of connectivity or ‘wires’ and low cost requires few wires. In this section we examine several designs for the network, and evaluate their costs and speeds. For each network, the maximum number of (routing) processors are defined (as diameter) through which a message must pass on its way from the source to the destination. The diameter measures the maximum delay for transmitting a message from one processor to another. We will also give the bisection width, i.e., the largest number of messages which can be sent simultaneously (without needing to use the same wire or routing processor at the same time and so delaying one another), no matter which processors are sending to which other processors. It is also the smallest number of wires you would have to cut to disconnect the network into two equal halves.

The organization and performance of a multiple processor system are greatly influenced by the interconnection network used to connect them.

Time Shared or Common Buses

The simplest interconnection system for multiprocessors is a common communication path connecting all of the functional units. This common path is called as time shared or common bus. We can say a bus is a collection of signal lines that carry module-to-module communication which provide the data highways connecting several digital system elements. Here all the processors and memory are connected to a common bus or buses and memory access is fairly uniform, but not very scalable.

The organization is least complex and easier to configure. Such an interconnection is often a totally passive unit having no active components such as switches. Transfer operations are completely controlled by the bus interfaces of the sending and receiving units. Since the bus is shared, a mechanism must be provided to resolve contention. The conflict resolution methods include static or fixed priorities, First-In-First-Out or FIFO queues and daisy chaining.

A unit that wishes to initiate the transfer must first determine the availability of the status of the bus and then address the destination unit to determine its availability and capability to receive the transfer. A command is also issued to inform the destination unit what operation it is to perform with the data being transferred, after which the data transfer is finally initiated. A receiving unit recognizes its address placed on the bus and responds to the control signals from the sender as shown in Figures 2.3 and 2.4, respectively.

NOTES

Fig. 2.3 Single Bus Multiprocessor Organization

If M3 can communicate with S5 then following conditions can occur:

- M3 sends signals (address) on the bus that causes S5 to respond.
- M3 sends data to S5 or S5 sends data to M3 (determined by the command line).

A multiprocessor with unidirectional buses uses both the buses (refer Figure 2.4).

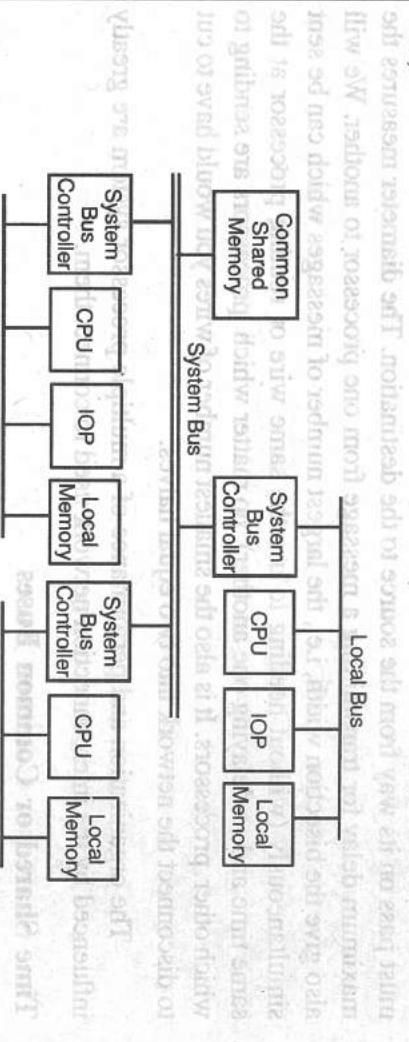


Fig. 2.4 Multi-Bus Multiprocessor Organization

This method increases the complexity. Here, the interconnection subsystem becomes an active device.

Characteristics that Affect the Performance of the Bus

The characteristics that affect the performance of the bus are as follows:

- Number of active devices on the bus
- Bus arbitration algorithm
- Centralization
- Data bandwidth
- Synchronization of data transmission
- Error detection

To validate, when there is a contention on the bus, the bus controller will detect the idle traffic, determine which bus has the highest priority, and then broadcast a message to all the devices on the bus to release the bus.

ANSWER

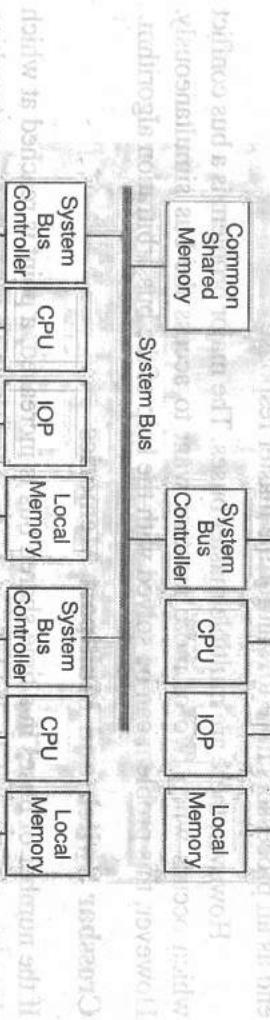


Fig. 2.5 Dynamic Network

Figure 2.5 shows a simple dynamic network. All processors sit on a shared bus (wire) which may be written to by at most one processor at a time, i.e., only one processor can communicate with the main memory. The processor that communicates with the memory has control over the bus. The other processor has to wait for the bus till it becomes free. A command is issued to the destination unit indicating what operations are to be performed. The address of the source is recognized by the destination unit after which the transfer is initiated. As more than one processor can request for the bus, there is a bus controller which resolves the conflict by assigning some priorities. The conflict resolution methods include static or fixed priorities like FIFO queue and daisy chaining. This is the way the CPU, memory and perhaps I/O devices are connected internally in a processor. The diameter is 1, since every processor is directly connected to every other processor. The bisection width is also 1, since only one message may be sent at a time. This is also the way a network of workstations is connected, if there is just one physical medium, like an Ethernet. More sophisticated networks, like Automated Teller Machine or ATMs, consist of buses connected with small crossbars. As only one is communicating, the overall performance is slow. The performance can be increased by increasing the number of buses, but that results in an increase in the complexity and cost of system. The other alternative is to have local buses each connected to its local memory. There are connected to a common memory by a common bus. The common memory also called shared memory and contains global information. While any system is working with the shared memory, the other processor works with the local memory. Such a system is called a multi-bus structure for the multiprocessor (refer Figure 2.6).

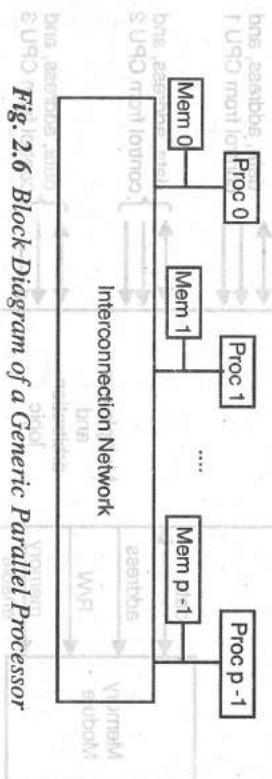


Fig. 2.6 Block Diagram of a Generic Parallel Processor

The following are two possible conditions for a master/slave architecture:

- **Master Device:** Device that initiates and controls the communication.
- **Slave Device:** Responding device.

NOTES

The major drawback of this system is that a bottleneck is created at the master end as all processors have to wait for the master response.

However, there are multiple master buses. The major problem is a bus conflict which occurs when two or more masters want to access the bus simultaneously. However, this problem can be solved with the help of the bus arbitration algorithm.

NOTES

Crossbar Switch and Multiport Memories

If the number of buses in a time shared bus is increased, a point is reached at which there is a separate path available for each memory unit. The interconnection networking is called as a nonblocking crossbar.

The crossbar (or Xbar) is the most expensive dynamic network, with each processor directly connected to every other processor (refer Figure 2.7). It is used in mainframes and smaller ones appear as components in hybrid networks. The memory modules and processor can be visualized as matrix with the memory module representing columns and the processor representing row of the matrix. At each intersection point a switch is placed, which determines the path from the processor to the memory. The processor places the address of the memory to access on the bus. The switch used to resolve the conflict is more than one processor's demand for the same memory module. The switches are a combination of multiplexer and arbitration logic.

Fig. 2.7 Crossbar Switch System for Multiprocessor

Figure 2.8 illustrates the functional structure of cross point in a crossbar network.

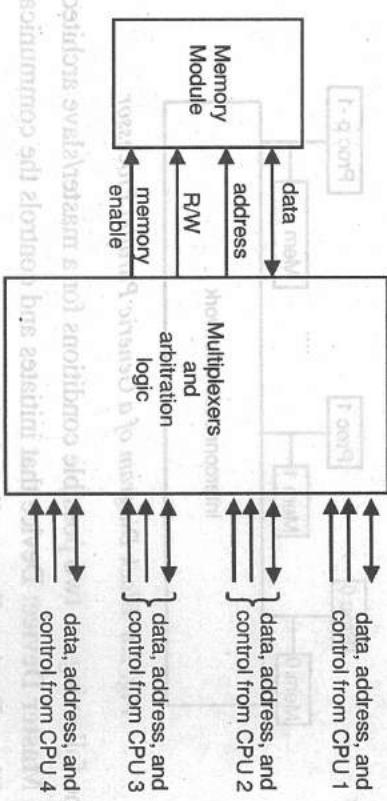


Fig. 2.8 Functional Structure of Cross Point in Crossbar Network

Multistage Networks for Multiprocessors

The Computer System

In order to design multistage networks, the basic principles involved in the construction and control of a simple crossbar has to be understood. A 2×2 switch has the capability of connecting the input A to either the output labelled 0 or to the output labelled 1, depending on the value of some control bit C_A of the input A. If $C_A = 0$ the input is connected to the upper output, and if $C_A = 1$, the connection is made to the lower output. Terminal B of the switches behaves similarly with a control bit C_B . The 2×2 module also has the capability to arbitrate between conflicting requests. If both inputs A and B require the same output terminal, then only one of them will be connected and the other will be blocked or rejected (refer Figures 2.9 and 2.10).



Fig. 2.9 A 2×2 Crossbar Switch

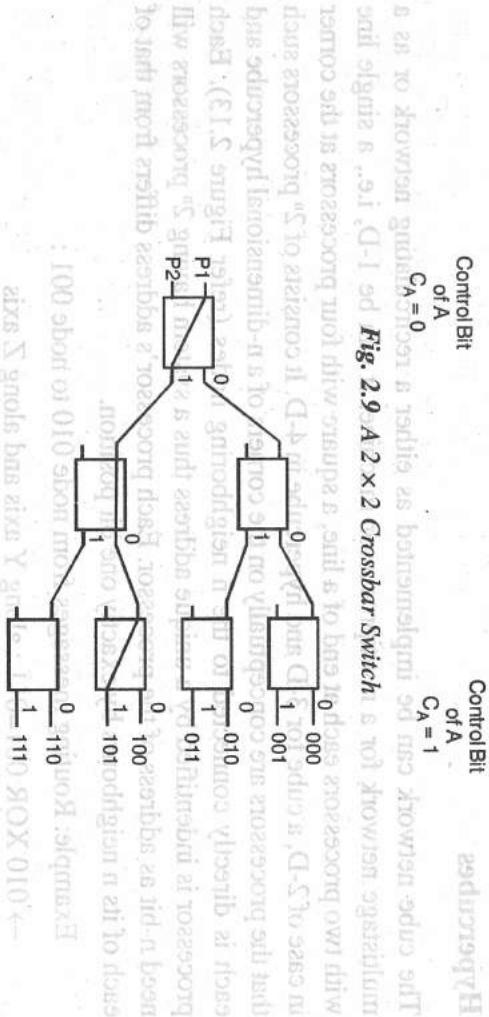


Fig. 2.10 1-by-8 Demultiplexer Implemented with 2×2 Switch Boxes

It is straightforward to construct a 1×2^n demultiplexer using the above 2×2 module.

Fig. 2.11 Interstage Switch

The interstage switch is a 2×2 switch which includes four stages. Stage 1 represents connection of A to 0, Stage 2 represents connection of A to 1, Stage 3 represents connection of B to 0 and Stage 4 represents connection of B to 1. Figure 2.12 illustrates the structure of 8×8 Omega switching network. An 8×8 Omega network is a multistage interconnection network, meaning that Processing

NOTES

Fig. 2.11 represents the structure of interstage switch which includes four stages. Stage 1 represents connection of A to 0, Stage 2 represents connection of A to 1, Stage 3 represents connection of B to 0 and Stage 4 represents connection of B to 1. Figure 2.12 illustrates the structure of 8×8 Omega switching network. An 8×8 Omega network is a multistage interconnection network, meaning that Processing

NOTES

Elements or PEs are connected using multiple stages of switches. Inputs and outputs are given addresses. The outputs from each stage are connected to the inputs of the next stage using a perfect shuffle connection system. In terms of binary representation of the PEs, each stage of the perfect shuffle can be thought of as a cyclic logical left shift; each bit in the address is shifted once to the left with the most significant bit moving to the least significant bit.

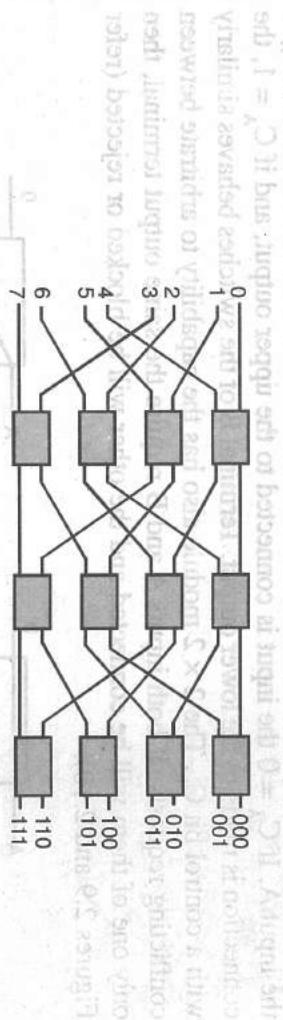


Fig. 2.12 8 × 8 Omega Switching Network

Hypercubes

The cube network can be implemented as either a recirculating network or as a multistage network for a multiprocessor machine. It can be 1-D, i.e., a single line with two processors each at end of a line, a square with four processors at the corner in case of 2-D, a cube for 3-D and hypercube in 4-D. It consists of 2^n processors such that the processors are conceptually on the corners of an n-dimensional hypercube and each is directly connected to the n neighboring nodes (refer Figure 2.13). Each processor is identified by a unique address thus a system having 2^n processors will need n-bit as address of the processor. Each processor's address differs from that of each of its n neighbours by exactly one bit position.

Example: Routing messages from node 010 to node 001 :

→ 010 XOR 001=011 : along Y axis and along Z axis

To design a n-dimension hypercube each processor must be connected to 2^n neighbours. This can also be visualized as the unit (hyper) cube embedded in d-dimensional Euclidean space, with one corner at 0 and lying in the positive orthant. The processors can be thought of as lying at the corners of the cube, with their (x_1, x_2, \dots, x_n) coordinates identical to their processor numbers and connected to their nearest neighbours on the cube. The popular examples where cube topology is used are iPSC, nCUBE and SGI O2K.

Vertical lines connect vertices processors whose address differ in the most significant bit position. Vertices at both ends of the diagonal lines differ in the middle bit position. Horizontal lines differ in the least significant bit position. The unit cube concept can be extended to an n-dimensional unit space called an n cube with n bits per vertex. We use binary sequence to represent the vertex address of the cube. Two processors are neighbours if and only if their binary address differs only in one digit place (refer Figure 2.13).

Omega switching is based on perfect shuffle connection system. It is a 4 stage omega switching network shown in figure 2.12. It shows 8 input nodes and 8 output nodes. The connections are as follows:

\emptyset $\emptyset = \{r <= 8 \wedge s \neq \emptyset\}$

$N = 1, n = \emptyset$

$N = 2, n = 1$

$N = 4, n = 2$

$N = 8, n = 3$

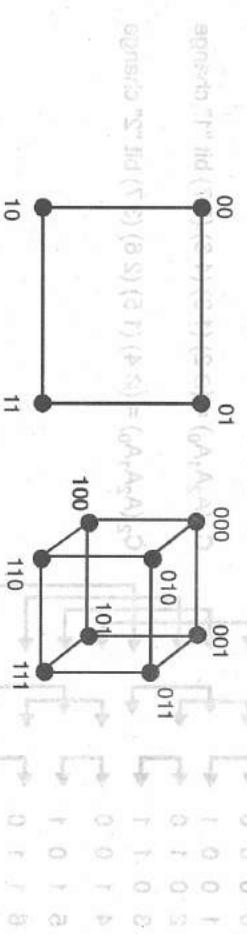
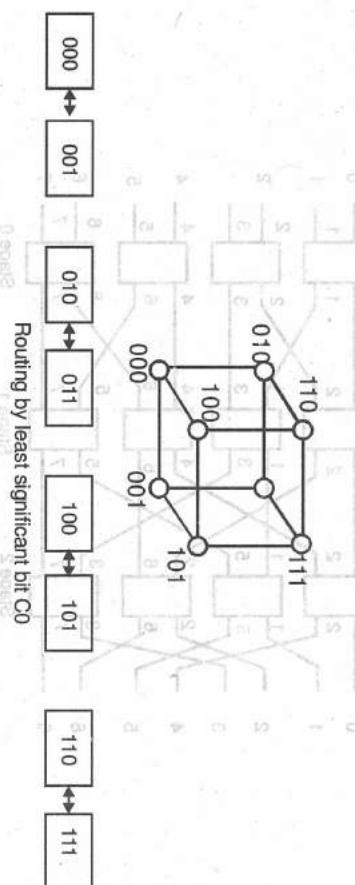


Fig. 2.13 Neighbouring Processors

In an n -dimension cube each processor located at the corner is directly connected to n neighbours. The addresses of neighbouring processor differ in exactly one bit position. Pease's binary n cube is the type of flip-flop network used in programmable switching network proposed for Phoenix is example of cube networks.

When the processor addresses are considered as the corners of an m -dimensional cube this network connects each processor to its m neighbours. The interconnections of the processors corresponding to the three routing function C_0 , C_1 and C_2 are shown in Figure 2.14.



Routing by least significant bit C_0

Routing by middle bit C_1

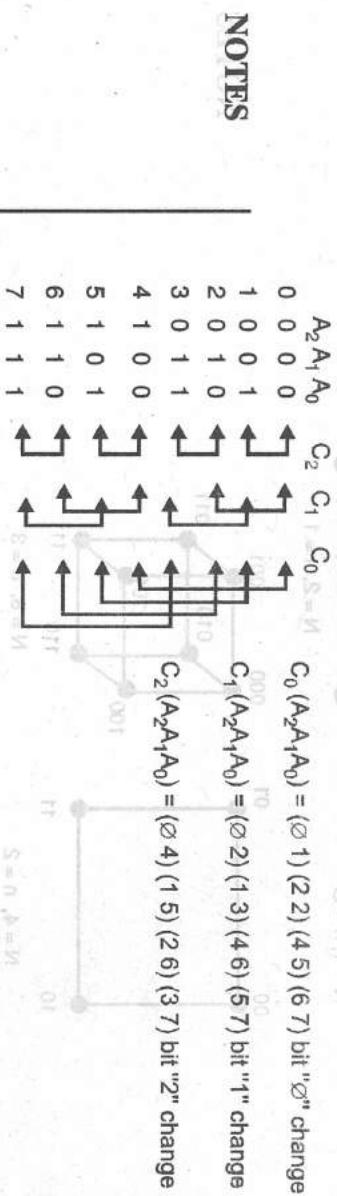
Routing by most significant bit C_2

S-MD

S-MD

Fig. 2.14 Recirculating Network

Multistage cube network takes $n \leq \log_2 N$ steps to rotate data from any processor to another (refer Figure 2.15).

Example: $N=8 \Rightarrow n=3$ Fig. 2.15 Multistage Cube Network for $N = 8$

The same set of cube routing functions, i.e., C_0, C_1, C_2 can also be implemented by a three stage network (refer Figure 2.16). Two functions switch box which can provide either straight or exchange routing is used for constructing multistage cube networks. The stages are numbered as 0 at input end and increased to $n-1$ at the output stage, i.e., the stage i implements the C_i routing function or we can say at i th stage connect the input line to the output line that differs from it only at the i th bit position.

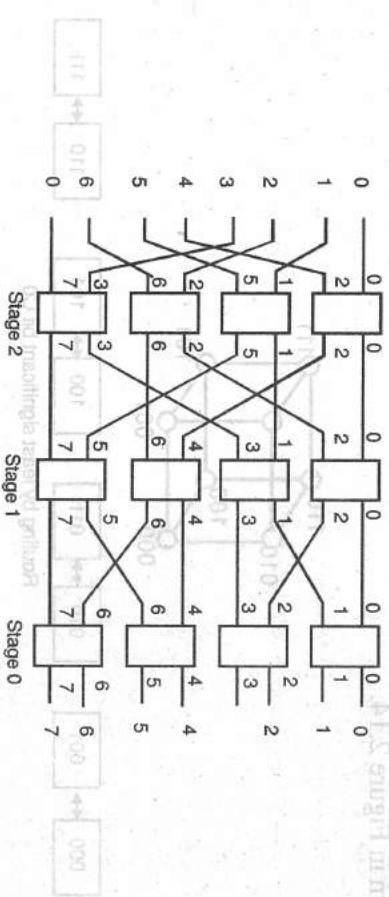


Fig. 2.16 Three-Stage Network

This connection was used in the early series of Intel Hypercubes and in the CM-2.

Suppose there are 8 process ring elements. In this case 3 bits are required for three addresses and the specific processor 000 is referred as the root. The children of the root are obtained by toggling the first address bit and so it becomes 000 and 100 (so 000 doubles as root and is left child). The children of the children are obtained by toggling the next address bit and so it becomes 000, 010, 100 and 110. Note that each node also plays the role of the left child. Finally, the leaves are obtained by toggling the third bit. Having one child identified with the parent causes no problems as long as algorithms use just one row of the tree at a time. This is illustrated in Figure 2.17.

Fig. 2.17 A Tree Embedded in a 3-D Hypercube

A Tree embedded in a 3-D Hypercube

NOTES

The evaluation of Static Interconnection Networks (SIN) is classified on the basis of the following characteristics:

Diameter: It specifies the maximum distance between any two processors in the network or in other words we can say diameter, is the maximum number of (routing) processors through which a message must pass on its way from source to reach destination. Thus the diameter measures the maximum delay for transmitting a message from one processor to another as it determines communication time hence the smaller the diameter the better will be the network topology.

Connectivity: It specifies that how many paths are possible between any two processors, i.e., the multiplicity of paths between two processors. Higher connectivity is desirable as it minimizes contention.

Arc Connectivity of the Network: It specifies the minimum number of arcs that must be removed for the network to break it into two disconnected networks. The arc connectivity of various network is as follows:

- 1 for linear arrays and binary trees.
- 2 for rings and 2-d meshes.
- 4 for 2-d torus.
- d for d-dimensional hypercubes.

Larger the arc connectivity lesser the conjunctions and better will be network topology.

Channel Width: The channel width is the number of bits that can communicate simultaneously by a interconnection bus connecting two processors

The bisection width and the bisection bandwidth for interconnection networks are considered as the two important parameters of a network. The term bisection width specifies the smallest number of links which have to be removed to split the network in two equal parts, while the term bisection bandwidth is used to bind the amount of data that can be moved between these parts. Typically both values can be derived from each other. If the network is segmented into two equal parts, this is the bandwidth between the two parts. Typically, this refers to the worst case segmentation but being of equal parts it refers to an actual bisection of the network.

Because bisection width refers to minimum number of communication links that can be removed to break it into two equal sized disconnected networks, hence the larger the bisection width the better is the network topology. Similarly, bisection

NOTES

bandwidth is the minimum volume of communication allowed between two bisected parts of the disconnected network with equal numbers of processors. Characteristically, bisection bandwidth is the rate at which communication can take place between one half of a computer and the other. A low bisection bandwidth or a large disparity between the maximum and minimum bisection bandwidths achieved by cutting the computers elements in different ways notifies that communications bottlenecks may occur in some calculations. This is important for the networks with weighted arcs where the weights correspond to the *link width*, i.e., how much data it can transfer.

Table 2.1 summarizes that the cost of networking can be estimated on a variety of criteria where we consider the number of communication links or wires used to design the network as the basis of cost estimation. Smaller, the number of links or wires the better the cost.

Table 2.1 Cost of Networking

Network	Diameter	Bisection Width	Connectivity	Cost (No. of Links)
Completely-Connected	$\lceil \log_2 p \rceil$	$p^2/4$	$p(p-1)$	$p(p-1)/2$
Star	1	1	1	$p-1$
Complete Binary Tree	$2 \log_2 (p+1)/2$	1	1	$15(p-1)/32$
Linear Array	$p-1$	1	1	$p-1$
2-D Mesh, No Wraparound	$2(\sqrt{p}-1)$	\sqrt{p}	2	$2(p-\sqrt{p})$
2-D Wraparound Mesh	$2\lceil \sqrt{p}/2 \rceil$	$2\sqrt{p}$	4	$2p$
Hypercube	$\log p$	$p/2$	$\log p$	$(2 \log p)/2$
Wraparound k -ary d -cube	$d\lceil k/2 \rceil$	$2k^{d-1}$	$2d$	$d p$

Dynamic Networks

The dynamic networks are those networks where the route through which data moves from one Processing Element or PE to another is established at the time the communication has to be performed. Usually all processing elements are equidistant and an interconnection path is established when two processing elements want to communicate by use of switches. Such systems are more difficult to expand as compared to static network. Examples include bus based, crossbar and multistage networks. Here the routing is done by comparing the bit level representation of the source and destination addresses. If there is a match it goes to next stage via pass through, else in case of a mismatch it goes via cross over using the switch.

2.2.3 Bus Interconnection

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 Computer System

A system operational. The CPU 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 to connect devices with each other as well as with 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.

- **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 2.18 shows the transfer among three registers R1, R2 and R3 through six data paths.

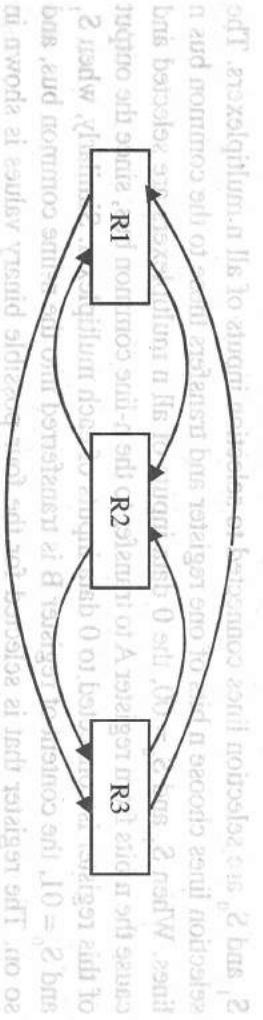


Fig. 2.18 Transfer among Three Registers

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 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 can be 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 2.19. 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.

And so on. So we have 10 lines in the bus. And the last two lines are for control lines, and the first two lines are for data transfer. And so on. The last two lines are for control lines, and the first two lines are for data transfer. And so on.

NOTES

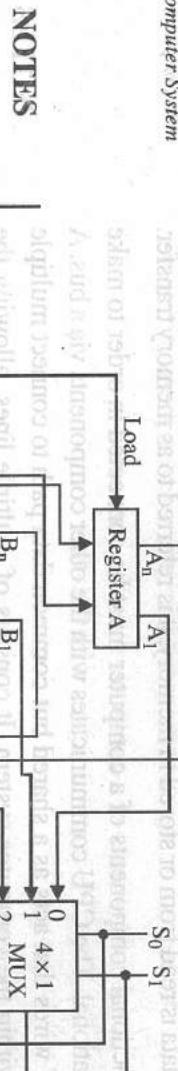


Fig. 2.19 Bus System for Four Registers

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 2.2.

Table 2.2 Function Table for Bus in Figure 2.19

S_1	S_0	Register Selected
0	0	A_n
0	1	A_1
1	0	B_n
1	1	B_1

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, 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 16×1 . Four selection lines are required. Also, the size of the destination decoder will be 4×16 .

Consider the following statement:

Allma C ← B mste vdonred vnonam vdonstrail vnonan zas startit 0C & singlet

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.

NOTES

sig DeligS mstamnayv vdonred vdonstrail vnonam vdonstrail vnonan zas startit 0C & singlet

Without a memory no information can be stored nor retrieved in a computer. Burks, Goldstine and John von Neumann stated that a computer memory has to be organized in a hierarchy. In such hierarchy, larger and slower memories are used to supplement smaller and faster ones. This observation has since then proven essential in constructing a computer memory. If the set of CPU registers are put aside as the first level for storing and retrieving information inside the CPU, then a typical memory hierarchy starts with a small, expensive and relatively fast unit called the cache. The cache is followed in the hierarchy by a larger, less expensive and relatively slow main memory unit. Cache and main memory are built using solid state semiconductor material. They are followed in the hierarchy by a far larger, less expensive and much slower magnetic memories that consist typically of the (hard) disk and the tape.

2.3 MEMORY SYSTEM DESIGN

ISSUE

The memory hierarchy consists of the total memory system of any computer. The memory components range from higher capacity slow auxiliary memory to a relatively fast main memory to cache memory that can be accessible to the high speed processing logic.

soft T At the top of this hierarchy, there is a CPU register which is accessed at full CPU speed. This is local memory to the CPU as the CPU requires it. Next comes cache memory, which is currently on the order of 32 KB to few megabytes or MB. After that is the main memory with sizes currently ranging from 16 MB for an entry level system to few gigabytes at the other end. Next are magnetic disks, and finally we have Magnetic tape and optical tapes. The memory hierarchy mainly depends on the following three key parameters:

- **Access Time:** CPU registers are the CPU's local memory which can be accessed in a few nanoseconds. Cache memory takes a small multiple of CPU registers.
- **Main memory access time:** Main memory access time is typically to few tens of nanoseconds.

The disk access time are at least 10 msec and tapes and optical disk access may be measured in seconds if the media is to be fetched and inserted into a drive.

Check Your Progress

1. Write the task of memory unit.
2. What is a common path?
3. What is a shared memory?
4. How is the speed of the bus affected?

NOTES

- Storage Capacity:** The storage capacity increases as we go down the memory hierarchy. CPU registers are good for 128 bytes. Cache memories are a few megabytes whereas the main memory are 10 to thousands megabytes. Magnetic disk of capacities are few gigabytes to tens of gigabytes. The capacity of tapes and optical disks are limited as they are usually kept offline.

Figure 2.20 illustrates memory hierarchy. Memory hierarchy starts with a small, expensive and relatively fast unit called the cache, followed by a larger, less expensive and relatively slow main memory unit. Cache and main memory are built using solid state semiconductor material typically Complementary Metal Oxide Semiconductor or CMOS transistors. It is customary to call the fast memory level as the primary memory. The solid-state memory is followed by larger, less expensive and far slower magnetic memories that consist typically of the (hard) disk and the tape. It is customary to call the disk as the secondary memory while the tape is conventionally called the tertiary memory. The objective behind designing a memory hierarchy is to have a memory system that performs as if it consists entirely of the fastest unit and whose cost is dominated by the cost of the slowest unit.

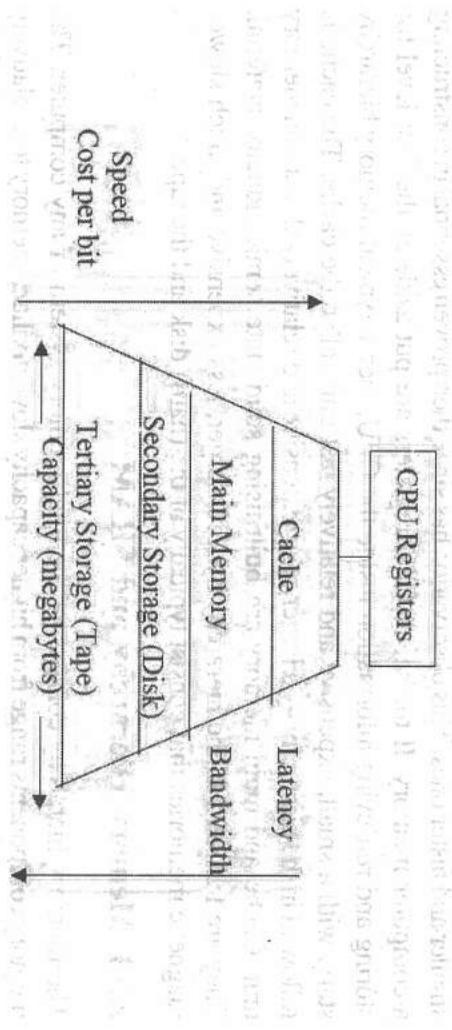


Fig. 2.20 Structure of Memory Hierarchy

Table 2.3 provides typical values of the memory hierarchy parameters. The term random access refers to the fact that any access to any memory location takes the same fixed amount of time regardless of the actual memory location and the sequence of accesses that take place. For example, if a write operation to memory location 100 takes 15ns and if this read is followed by a write operation to memory location 3000, then the write operation will take 15ns. This is to be compared to sequential access in which if access to location 100 takes 15ns and if a consecutive access to location 101 takes 20ns, then it is expected that an access to location 300 may take 1000 ns. This is because the memory has to cycle through locations 100 to 300 with each location requiring 5ns. Table 2.3 summarizes the access type, capacity, latency and bandwidth of different types of memory.

Memory Type	Access Type	Capacity (megabytes)	Latency (ns)	Bandwidth (MB/sec)
CPU Registers	Random	128	1	1000
Cache	Random	1–1000	1–10	100–1000
Main Memory	Random	10–10000	10–100	100–1000
Secondary Storage (Disk)	Sequential	>10000	>1000	<100

Table 2.3 Access Type, Capacity, Latency and Bandwidth for Registers and Memory

Registers and Main Memory	Access Type	Capacity	Latency	Bandwidth	NOTES
CPU Register	Random	64-1024 Bytes	1-10ns	System Clock Rate	
Cache Memory	Random	8-512 KB	15-20ns	10-20 MB/s	
Main Memory	Random	16-512 MB	30-50ns	1-2 MB/s	
Disk Memory	Direct	1-20 GB	10-30ns	1-2 MB/s	
Tape Memory	Sequential	1-20 GB	30-1000ns	1-2 MB/s	

The efficiency of a memory hierarchy depends on the principle of moving information into the fast memory infrequently and accessing it many times before replacing it with new information. This principle is possible due to a well-known phenomenon called locality of reference, i.e., within a given period of time, programs tend to reference relatively confined area of memory repeatedly.

Another way of viewing the memory hierarchy in any computer system is illustrated in Figure 2.20. The main memory is at the central place as it can communicate directly with the CPU and through the I/O processor with the auxiliary devices. Cache memory is placed in between the CPU and the main memory.

Cache usually stores the program segments currently being executed in the CPU and temporary data frequently asked by the CPU in the present calculations. The I/O processor manages the data transfer between the auxiliary memory and the main memory. Usually the auxiliary memory has a large storage capacity but has low access rate as compared to the main memory and hence is relatively inexpensive. Cache is very small but has very high access speed and is relatively expensive. Thus, we can say that, $\text{Access speed} \propto \text{Cost}$.

Thus, the overall goal of using a memory hierarchy is to obtain the highest possible average speed while minimizing the total cost of the entire memory system.

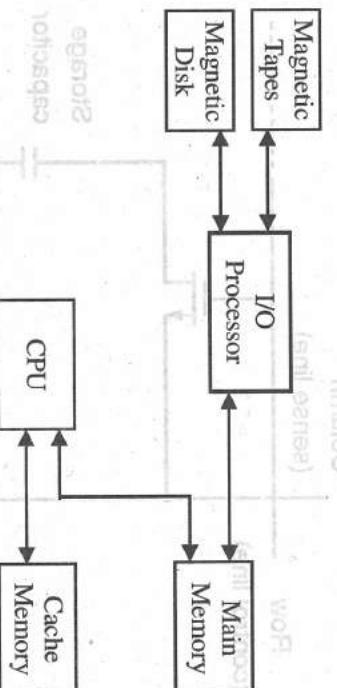


Fig. 2.21 Memory Hierarchy System

Static Random Access Memory or SRAM is a type of semiconductor memory. The word 'static' represents that the memory retains its content as long as power is applied to the circuit. It uses bistable latching circuitry to store each bit. SRAM exhibits

NOTES

data but is still volatile in the conventional way that data is eventually lost when the memory is not powered. SRAM is a type of RAM that holds data and placed within its cells until the values are either overwritten or the power is removed. This is opposed to Dynamic RAM or DRAM which allows the data exiting by the cells discharging every millisecond unless it is refreshed. In static RAM, a form of flip-flop holds each bit of memory. A flip-flop for a memory cell usually takes four or six transistors, but never has to be refreshed. This makes static RAM significantly faster with access times in the 10 to 30 nanosecond or ns range than DRAM. However, a static memory cell takes up more space on a chip than a dynamic memory cell. Therefore, you get less memory per chip which makes static RAM more expensive.

2.3.2 Advanced DRAM Organization

The Dynamic Random Access Memory (DRAM) is the lowest cost, highest density random access memory available. Nowadays, computers use DRAM for main memory storage with the memory sizes ranging from 16 to 256 MB.

The DRAM stores its binary information in the form of electric charges on capacitors. Data are stored as charge on every capacitor, which must be *recharged* or *refreshed* thousands of times every second in order to retain the stored charge. These memory devices make use of an integrated Metal Oxide Semiconductor or MOS capacitor as basic memory cell instead of a flip-flop. The advantage of this cell is that it allows very large memory arrays to be constructed on a chip at a lower cost per bit than in static memories. The disadvantage is that the MOS capacitor cannot hold the stored charge over an extended period of time and it has to be refreshed every few milliseconds. This requires more circuitry and complicates the design problem. Static RAMs are simpler than dynamic RAMs.

A typical dynamic RAM cell consisting of a single Metal Oxide Semiconductor Field Effect Transistor or MOSFET and a capacitor is shown in Figure 2.22. A dynamic RAM consists of an array of such memory cells. In this type of cell, the transistor acts as a switch. The memory cell also requires MOSFETs for READ and WRITE functions to operate the cell. Data input is connected for storage by a WRITE control signal.

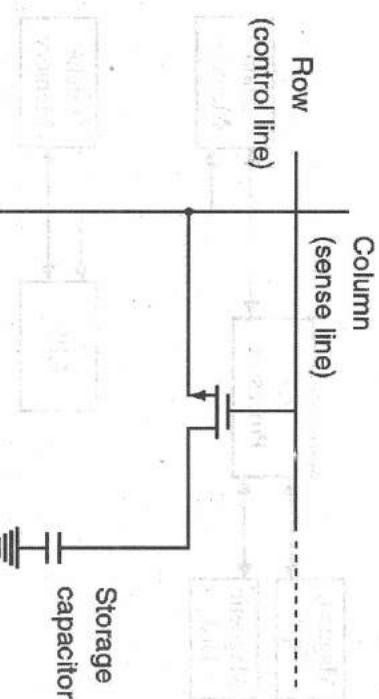


Fig. 2.22 A Dynamic MOS RAM Cell

The dynamic RAM offers reduced power consumption and large storage capacity in a single memory chip. With the availability of such high packing density memory Integrated Circuits or ICs, the capacity of memory will continue to grow.

Dynamic Memory Organization: The organization of a dynamic memory allows many cells to be accessed by a minimum amount of circuitry. Each memory cell is organized onto a data input/output line when a Row and a Column selected MOSFETs are determined as shown in Figure 2.23.

For example, to select the DRAM cell marked 1–1 (HIGH) has to be applied at Row 1 and Column 1 select inputs. A HIGH at Row 1 input selects all 128 transistors connected to the Row 1 select line while the HIGH at the Column 1 input selects one of the 128 possible Columns, thereby the cell marked 1–1 being read onto the I/O data line. A sense amplifier for each column is necessary to convert from the low voltage and low energy to a sufficient level voltage on the I/O data line. The circuitry interposed between a bit storage capacitor and external data line is called a *sense amplifier*. A matrix of 64×128 accounts for 8192 individual memory cells and only one cell is connected onto the data line at anytime.

NOTES

The Column and Row decoders together selects a single capacitor and provide access to that 1-bit storage site for reading and writing operations. All the capacitors in an entire row can be refreshed simultaneously. There is a provision on the chip for activating all the sense amplifiers simultaneously and closing simultaneously all the appropriate switches in the bit lines. Such simultaneous activation will refresh all the capacitors in the Row selected by the Row decoder. (1) in quo istib sigrte & bns (Q) called *address multiplexing* to reduce the number of address lines and thus the number of input/output pins on the IC package.

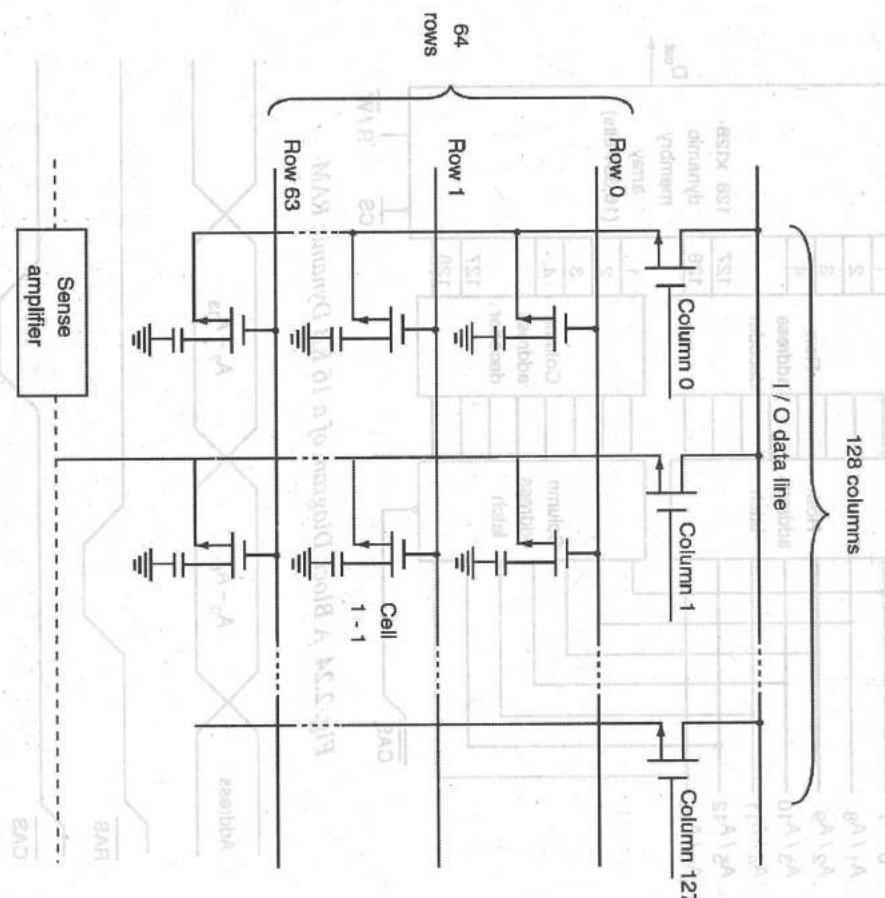


Fig. 2.23 Row and Column Selection of a DRAM Cell

NOTES

A block diagram of a 16 KB (16,384 bits) dynamic RAM that has been simplified to illustrate address multiplexing is shown in Figure 2.24. Since $2^{14} = 16,384$, the 14-bit address is applied (seven bits at a time) to the address inputs. First of all, the 7-bit Row address has to be applied and the $\overline{\text{RAS}}$ (Row Address Strobe) latches the 7-bits into the Row address latch. Next, the 7-bit Column address is applied to the address inputs and the $\overline{\text{CAS}}$ (Column Address Strobe) latches the remaining 7-bits into the Column address latch. Then, the 7-bit Row address and the 7-bit Column address are decoded to select the appropriate memory cell in the 128×128 dynamic memory array for a READ or WRITE operation. The timing diagram of the address multiplexing operation is shown in Figure 2.25.

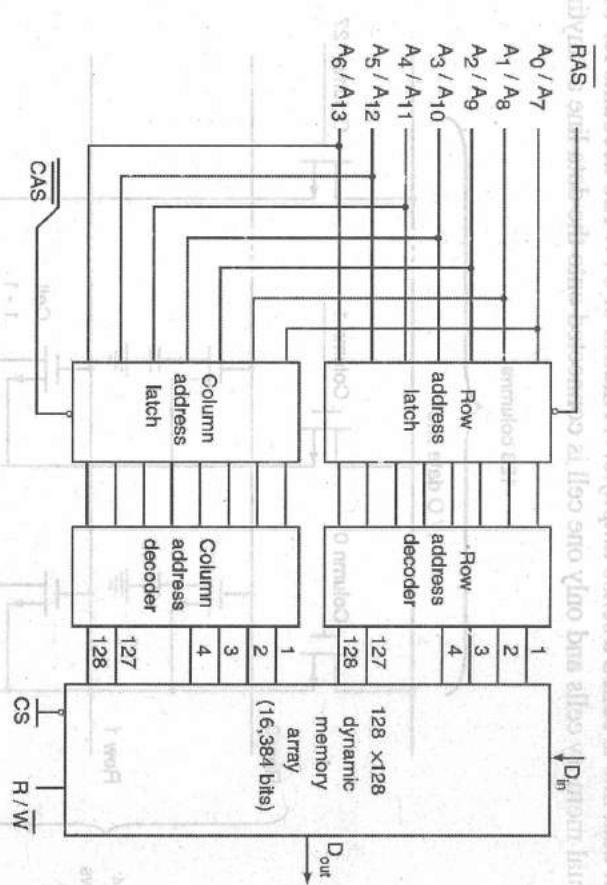


Fig. 2.24 A Block Diagram of a 16 KB Dynamic RAM

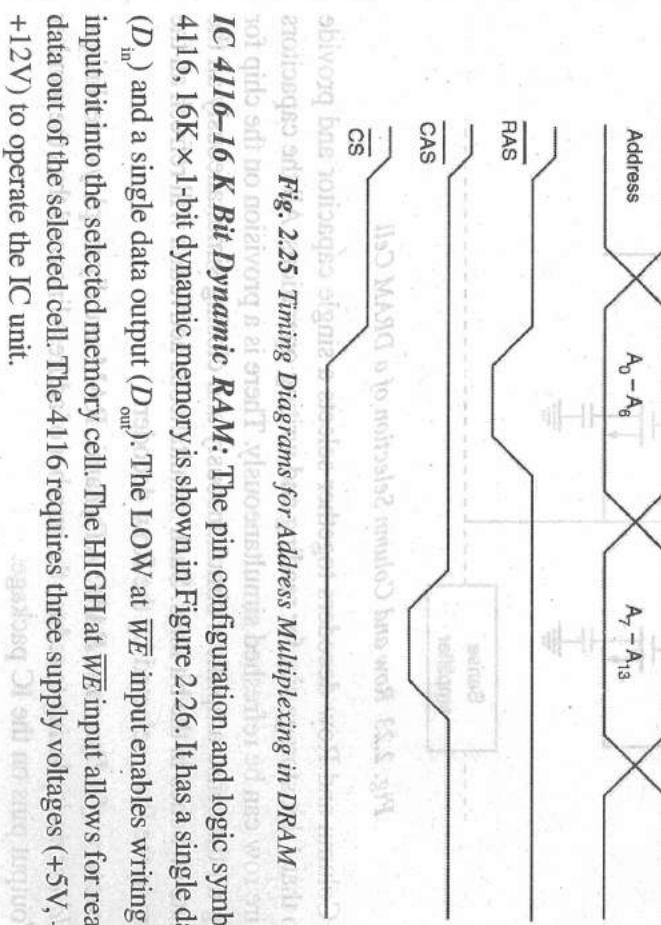


Fig. 2.25 Timing Diagrams for Address Multiplexing in DRAM

IC 4116-16 K Bit Dynamic RAM: The pin configuration and logic symbol of IC 4116, 16K \times 1-bit dynamic memory is shown in Figure 2.26. It has a single data input (D_{in}) and a single data output (D_{out}). The LOW at $\overline{\text{WE}}$ input enables writing the data input bit into the selected memory cell. The HIGH at $\overline{\text{WE}}$ input allows for reading the data out of the selected cell. The 4116 requires three supply voltages (+5V, -5V and +12V) to operate the IC unit.

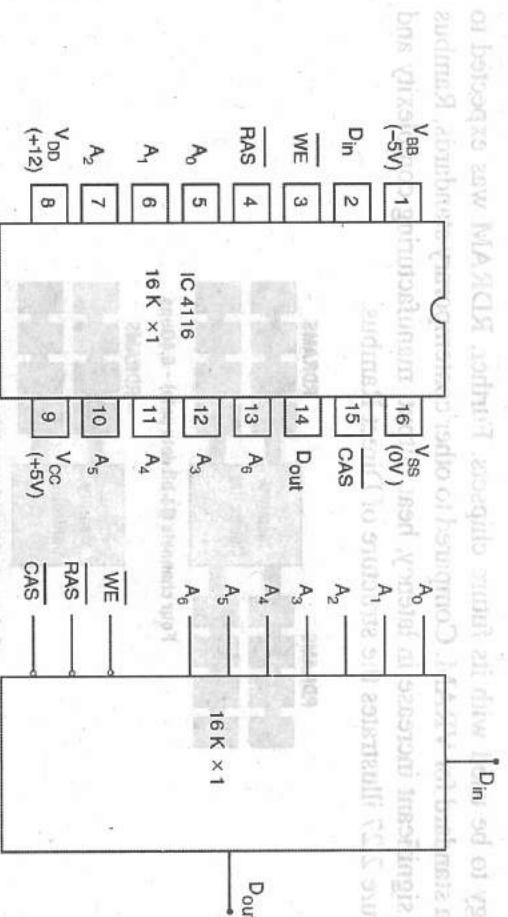


Fig. 2.26 IC 4116-16 K Bit Dynamic RAM

Refreshing a memory cell mechanism is performed at each of the 128 row address every 2ms. The organization of the memory cells allows selecting a single row using the row-address bits. A read operation for that row also refreshes all memory cells in that row. With 128 rows, the memory unit must be cycled to read from each of the 128 rows and it is repeated at least every 2 ms.

The block diagram of IC 4116 with 128 rows and 128 columns, showing the address and data latching operation as well as the memory organization is illustrated in Figure 2.26. Address bits at inputs A_0 through A_6 are latched as row-select bits by the \overline{RAS} signal and as column-select bits by the \overline{CAS} signal. The \overline{WE} signal latches the input data bit while \overline{CAS} latches the output data bit. Once latched, the row address bits are decoded to select one of the 64 rows in either top or bottom half of the memory stack (one of 128 rows), while the column address bits select one of 128 columns. The single memory cell is then latched as the output data bit (D_{out}) on a read operation or the input data bit (D_{in}) is written into the selected cell on a write operation.

To improve efficiency and speed, a number of methods for reading and writing the memory have been developed. In advanced DRAM organization, memory arrays are arranged in rows and columns of memory cells called WORDLINE and BITLINE, respectively. Each memory cell has a unique location or address defined by the intersection of a row and a column. An advanced DRAM memory cell is a capacitor that is charged to produce a 1 or a 0. In today's technologies, trenches filled with dielectric material are used to create the capacitive storage element of the memory cell. Enhanced DRAM contains small SRAM as well. SRAM holds last line read (locality of reference). It is used to improve performance by inserting high cost and high speed SRAM cache between the DRAM main memory and the processor.

Direct Rambus

Direct Rambus or DRAM or DRDRAM is a type of synchronous dynamic RAM. DRDRAM was developed by Rambus Inc. RDRAM was initially expected to become the standard in PC memory, especially after Intel agreed to license the Rambus

NOTES

technology to be used with its future chipsets. Further, RDRAM was expected to become a standard for VRAM. Compared to other contemporary standards, Rambus shows a significant increase in latency, heat output, manufacturing complexity and cost. Figure 2.27 illustrates the structure of Direct Rambus.

NOTES

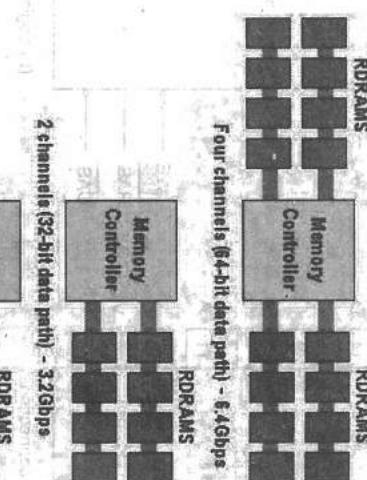


Fig. 2.27 Structure of Direct Rambus

At current speeds a single channel (16-bit data path) is capable of data transfer at 1.6 Gbps and multiple channels 64-bit data path can be used in parallel to achieve a throughput of up to 6.4 Gbps. The new architecture will be capable of operating at a system bus speed of up to 133MHz.

Cache DRAM

Cache DRAM (CDRAM) is a development that has a localized, on chip cache with a wide internal bus composed of two sets of static data transfer buffers between cache and DRAM. This architecture achieves concurrent operation of DRAM and SRAM synchronized with an external clock. Separate control and address input terminals of the two portions enable independent control of the DRAM and SRAM, thus the system achieves continuous and concurrent operation of DRAM and SRAM. CDRAM can handle CPU, Direct Memory Access (DMA) and video refresh at the same time by utilizing half-time multiplexed interleaving through a high speed video interface. The system transfers data from DRAM to SRAM during the CRT blanking period. Graphic memory as well as main memory and cache memory are unified in the CDRAM; it can replace cache and main memory, and it is has already been proven that a CDRAM based system has a 10 to 50 per cent performance advantage over a 256 KB cache based system. Cache DRAM is the larger SRAM component and used as cache or serial buffer.

2.3.3 Interleaved Memory

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 initiated 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 Personal Computer or 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 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 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.

NOTES

Memory Address				Memory Address				Memory Address				Memory Address			
M0		M1		M2		M3		M0		M1		M2		M3	
0	00	00	4	01	00	8	10	00	12	11	00	0	00	4	01
1	00	01	5	01	01	9	10	01	13	11	01	1	01	5	01
2	00	10	6	01	10	10	10	10	14	11	10	2	01	6	01
3	00	11	7	01	11	11	10	11	15	11	11	3	01	7	01
(a) High Order Arrangement and Low Order Arrangement of Interleaving															
0	00	00	1	00	01	2	00	10	3	00	11	0	00	1	00
4	01	00	5	01	01	6	01	10	7	01	11	4	01	5	01
8	10	00	9	10	01	10	10	10	11	10	11	8	10	9	10
12	11	00	13	11	01	14	11	10	15	11	11	12	11	13	11
(b) Low Order Arrangement of Interleaving															

Fig. 2.28 (a) High Order Arrangement and (b) Low Order Arrangement of Interleaving

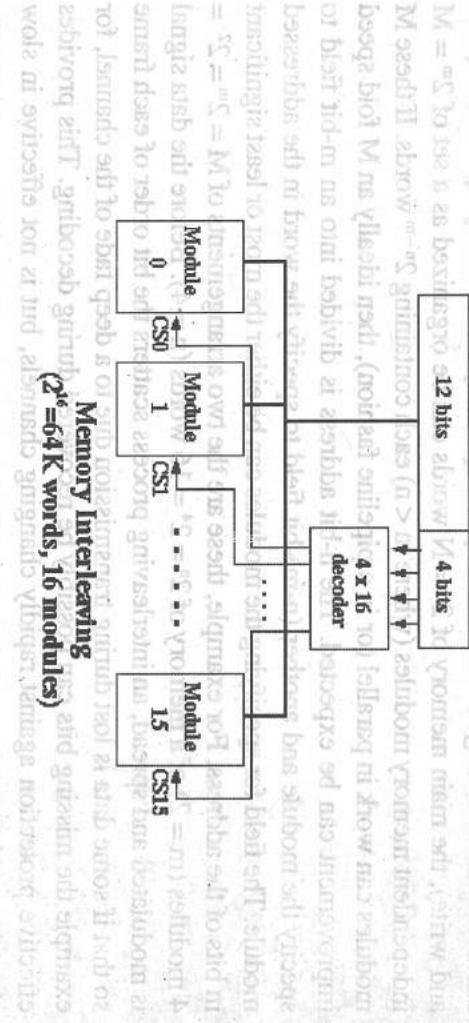
In general, the CPU is more likely to need to access the memory for a set of consecutive words either a segment of consecutive instructions in a program or the components of a data structure, such as an array, the interleaved (low order) arrangement is preferable as consecutive words are in different modules and can be fetched simultaneously

(refer Figure 2.28 (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 2.28 (a)). The following example on interleave infrastructure will make the concept clear.

NOTES

Example 2.1: 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.



2.3.4 Associative Memory

An associative memory, also called Content Addressable Memory (CAM), is a very high speed memory that provides a parallel search capability. It is capable of searching the contents of all its locations at any instant of time. An associative memory checks all data stored in it simultaneously, with a particular match pattern, i.e., here content is matched rather than address as done in random access memory. Hence, each word in such a memory should include a circuit that can do a pattern comparison. These memories involve a complex and advance circuitry making it more expensive than the conventional memories. It is used for the special purposes requiring high speed. However, there are areas within high performance architectures, such as cache and virtual memory management, where content addressable memories play a significant role and their cost can easily be justified.

The general structure of an associative memory is shown in Figure 2.29. It consists of a set of words, each having $n-1$ bits. When a word is written in it, no address is given. The memory is capable of finding an empty space to store the word in it. When memory has to read any data the content of word or the part of word is specified. This specified word or part of word is used for pattern matching. Usually such a memory stores large word sizes, often of size 100 bits or more. However, the number of words is limited because more the words, the more are circuitry requirements and high is the price.

diagram shows M to Y versus the **DATA**. A single input bit M is used to mask the contents of the array. If M = 1, then the entire word is returned. If M = 0, then the word is ignored.

NOTES

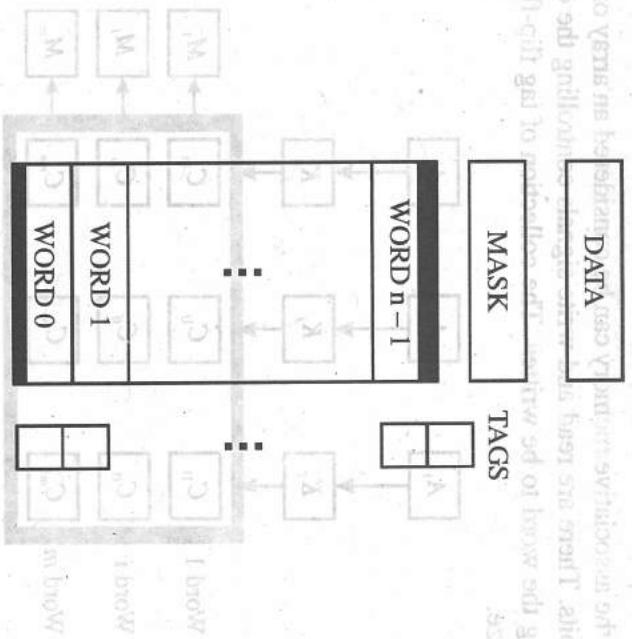


Fig. 2.29 An Associative Memory

Two registers are used with CAM — a MASK register, also called argument register and a data register, also called key register. MASK is used to mask the contents of registers in the register buffer. The size of register is same as one word stored in associative memory. In addition, each word has a circuit to perform the comparison operation. Corresponding to each word, one or more tag bits are associated. Each set of tag bits forms a bit-slice register which has same size as that of number of words in the CAM.

Mask register is used to select the word to be compared with the key register. Mask register is used to select the word to be compared with the key register.

Argument Register (A)

Key Register (K) is used to compare the key register with the word register.

Match Register is used to compare the key register with the word register.

Word register is used to store the word register.

Associative Memory Array and Logic receives the Input, Read, Write, and MASK signals. It also generates the Match Register output.

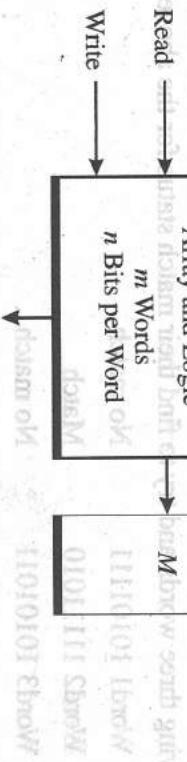


Fig. 2.30 Block Diagram of an Associative Memory

Diagram shows M to Y versus the **DATA**. A single input bit M is used to mask the contents of the array. If M = 1, then the entire word is returned. If M = 0, then the word is ignored.

In Figure 2.30, the associative memory can be considered an array of M words with each having n bits. There are read and write signals controlling the operation and n input bit holding the word to be written. The collection of tag flip-flop form match register of M size.

NOTES

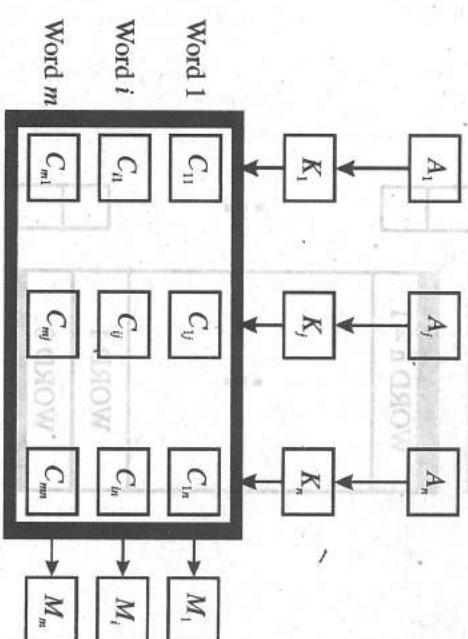


Fig. 2.31 Associative Memory of m Words Each of n Size

Here each word is matched in parallel with the content of the argument register. The bit corresponding to each word of memory in M holds the match status. Once matching process is done, those bits in match register will be set corresponding to which word in associative memory has been matched. We can use some portion of argument register by using a mask in key register. The entire argument is compared with each memory word if the key register contains all 1's. Else only those bits in the argument that have 1's the corresponding bit in key register are 1. Thus, the key provides mask or identifying piece of information which specifies how memory reference is to be made.

Let us consider an example where the argument register A and the key register B have bit configuration as follows:

A 10101010
B 00001111

Now these two registers set the search pattern to 1010 in last four bits. For all words that contain the pattern 1010 in last four bits will set the match bit. Let us consider the following three words and try to find their match status for the above pattern.

Word1 10101111	No match
Word2 11111010	Match
Word3 10101011	No match

Here only word 2 sets match status. Thus, in this case when the CAM performs a search, the logic associated with pattern, i.e., the presence of 1010 pattern in the last four bit (selected pattern) will be compared with each word. The tag bit for the corresponding word will be set to one if the match is found. Thus, tag bit for word 2 will be set. At the end of this process, all matching words may be identified by their tag bits. In case of system that supports more complex operations often more than one tag bit is used.

Some cache designs have bypasses. A bypass is a connection between the input and output of a cache. However, it is not always necessary. If the bypass is not used, then the cache is a full cache. If the bypass is used, then the cache is a partial cache.

NOTES

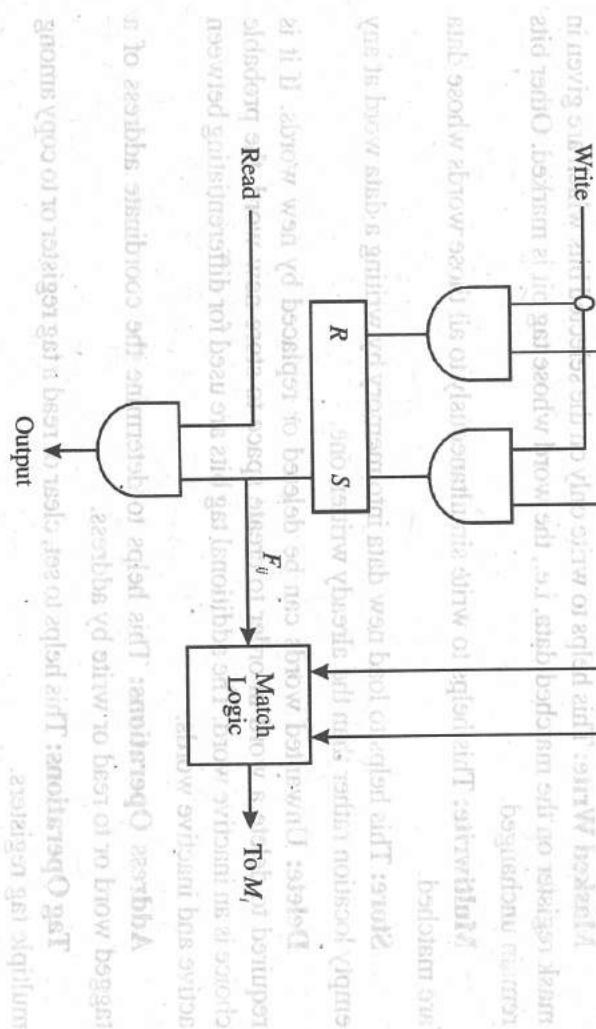


Fig. 2.32 One Cell of Comparator Circuit

The high speed of the associative memory is achieved by performing the matching operation simultaneously for all words stored in the associative memory. Hence, a comparator is required with every word in the memory so that all the comparisons are done in parallel with help of the circuit shown in Figure 2.32.

An associative memory has an n -bit input but not necessarily all possible 2^n combinations are present. The n -bit address input is a tag stored in argument register that is compared with a tag field in each location simultaneously. The circuit in Figure 2.32 is one cell of associative memory. If the input tag matches a stored tag, the data associated with that location is the output. Otherwise the associative memory produces a miss output. An associative memory does not have explicit address as other memories, such as RAM, ROM or hard disk has, rather data itself acts as address. For knowing the information we have to check the content, i.e., whether that data is stored somewhere or not. Cache memory is usually an associative memory. For data searching, associative mapping is used.

After the search operation is complete, one or more than one or NIL tag can show the match status. Apart from the main search operation, the other common operations performed on associative memory are READ and WRITE. If only one word is matched, a required READ operation may be performed to transfer data from the selected location. If no match is found, an error signal will be returned. In case of more than one match, the READ operation will select any one of matched words, read it and clear its tag bit. The similar operation will be performed for the successive READs, such that all of the matched words are accessed. An associative memory is dynamic memory and hence should also have a write capability for storing the information to be searched.

Some other operations associated with associative memories are as follows:

NOTES

Count Matches: This helps the user know how much matches are there, i.e., how many tag bits are set. As said earlier, it can be zero, one or many. However, it is difficult to know the exact count if there are more than one matches.

Masked Write: This helps to write only on the selected bits which are given in mask register on the matched data, i.e., the word whose tag bit is marked. Other bits remain unchanged.

Multiwrite: This helps to write simultaneously to all those words whose data are matched.

Store: This helps to load new data into memory by writing a data word at any empty location rather than the already written one.

Delete: Unwanted words can be deleted or replaced by new words. If it is required to delete a word in order to create space to store new word, the probable choice is an inactive word. The additional tag bits are used for differentiating between active and inactive words.

Address Operations: This helps to determine the coordinate address of a tagged word or to read or write by address.

Tag Operations: This helps to set, clear or read a tag register or to copy among multiple tag registers.

Other than cache memory another common use of associative memory is to hold page map table in case of virtual memory (refer Figure 2.33).

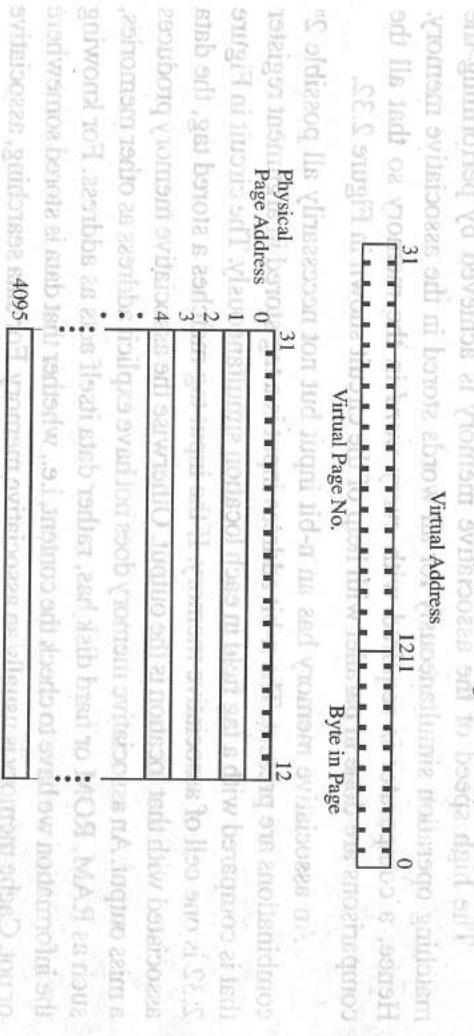


Fig. 2.33 Associative Memory

Acceleration of Virtual Address Translation or VAT

The virtual memory system requires at least two memory accesses, one for searching the page in the page map table and the other for actual data fetching. In order to reduce access time we can store the page map table in associative memory instead of main memory. Hence, the access time is reduced by two if page map table is kept in main memory. We can use an associative memory called a Translation Lookaside Buffer (TLB) for this purpose (refer Figure 2.34). Since associative memory is very expensive, it is often not feasible to store complete mapping information in it. Hence,

we keep a small amount of recently used pages in the associative memory and the complete copy of page map table in the main memory. So, if the recently used page is used again, like is quite probable based on the principle of locality, the associative memory fetches the translation and inserts it directly into the physical address avoiding the references made to main memory.

As it is not possible to store the complete table in TLB, hence there are chances of misses in TLB. In case of a miss, the page map table stored in main memory is to be accessed to generate the address. This address is also stored in TLB as a new entry. A TLB is typically fully associative and is of small size that can store ten to a few thousand entries. These entries can help in translation of address to a large number of locations (e.g., 4K). Normal exploitation of memory locality ensures that entries in the TLB change much less frequently than cache entries. Typical hit rates in a TLB are approximately 45 per cent. Also, when a miss occurs, an extra time of about 10 to 40 clock cycles is wasted for retrieving data from the main memory. So, if there is 1 in 100 accesses miss ratio in the TLB and the penalty is 40, then the processor slows down by nearly one third of the actual time.

Associative memory is also very commonly used in database system for fast retrieval of information by feeding some data, such as getting the information about the customer by feeding the customer ID. Now search in case of associative memory will be content based, i.e., on the customer ID.

NOTES

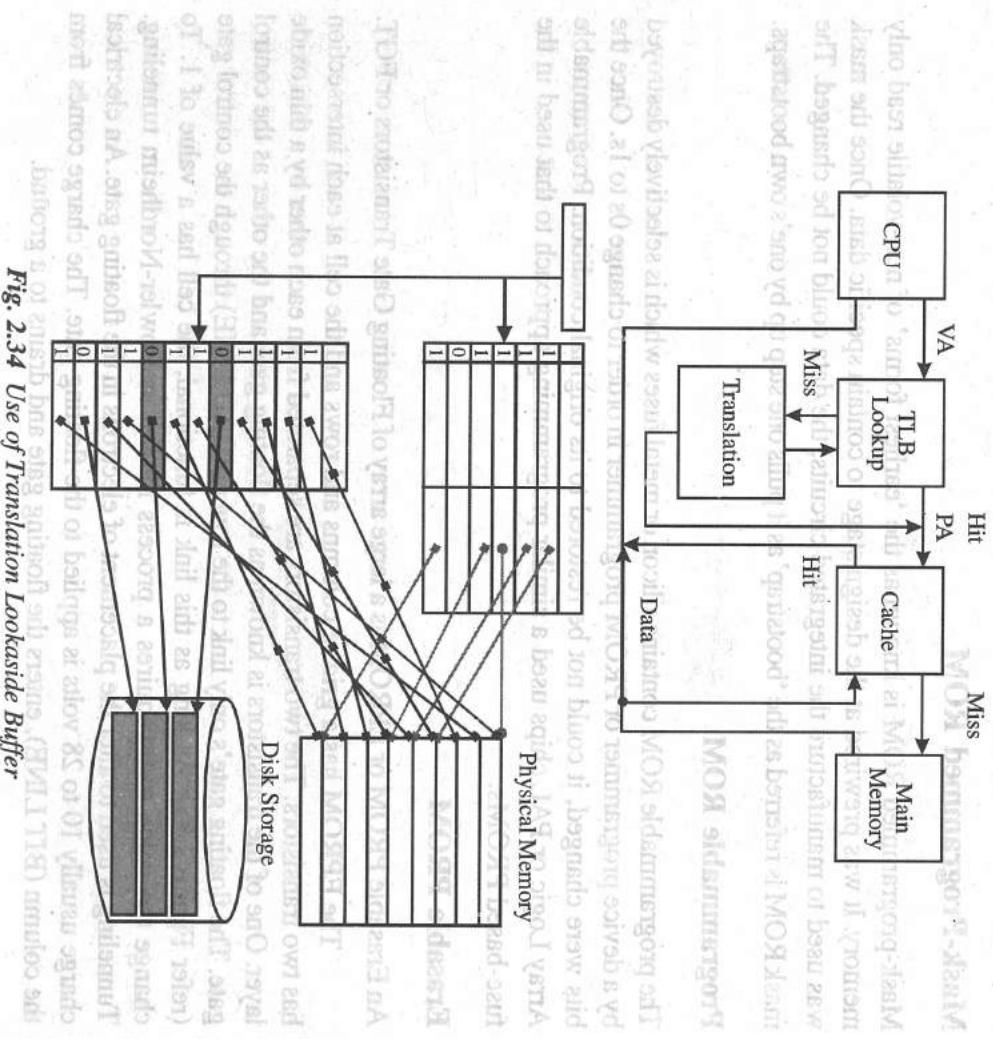


Fig. 2.34 Use of Translation Lookaside Buffer.

NOTES

2.3.5 Non-Volatile Memory

Non-volatile memory is computer memory that can retain the stored information even when power is not on. Examples of non-volatile memory include Read Only Memory or ROM, flash memory, Ferroelectric RAM or FRAM and early computer storage methods, such as paper tape and punched cards. Non-volatile memory is basically used for the task of secondary storage. The most widely used form of primary storage today is a volatile form of Random Access Memory or RAM which specifies that when the computer is shut down, anything contained in RAM is lost. International Business Machines or IBM is currently developing MRAM or Magnetoresistive RAM. Non-volatile data storage can be categorized in electrically addressed systems (ROM) and mechanically addressed systems, such as hard disks, optical disc, magnetic tape, etc. Electrically addressed systems are expensive but fast. FRAM is a non-volatile memory combining both ROM and RAM advantages along with non-volatility features. Its higher speed is in write mode but its lower power consumption as well as its higher endurance makes it superior to any other memory type. The ROM can be implemented as a combinational circuit because the contents of the ROM can be treated as the functions of the address. The information is stored in the structure and connections of the circuit without any real storage capability actually needed. Electrically addressed non-volatile memories are based on charge storage. These can be categorized according to their write mechanism as follows:

Mask-Programmed ROM

Mask-programmed ROM is known as the 'earliest forms' of non-volatile read only memory. It was prewired at the design stage to contain specific data. Once the mask was used to manufacture the integrated circuits, the data could not be changed. The mask ROM is referred as the 'bootstrap' as it pulls one step up by one's own bootstraps.

Programmable ROM

The programmable ROM contains silicon or metal fuses which is selectively destroyed by a device programmer or PROM programmer in order to change 0s to 1s. Once the bits were changed, it could not be restored to its original condition. Programmable Array Logic or PAL chips used a similar programming approach to that used in the fuse-based PROMs.

Erasable PROM

An Erasable PROM or EPROM is a large array of Floating Gate Transistors or FGT.

The EPROM has a grid of columns and rows and the cell at each intersection has two transistors. The two transistors are separated from each other by a thin oxide layer. One of the transistors is known as the floating gate and the other as the control gate. The floating gate's only link to the row (WORD LINE) through the control gate (refer Figure 2.35). As long as this link is functional, the cell has a value of 1. To change the value to 0 requires a process known as Fowler-Nordheim tunneling. Tunneling is used to alter the placement of electrons in the floating gate. An electrical charge usually 10 to 28 volts is applied to the floating gate. The charge comes from the column (BIT LINE), enters the floating gate and drains to a ground.

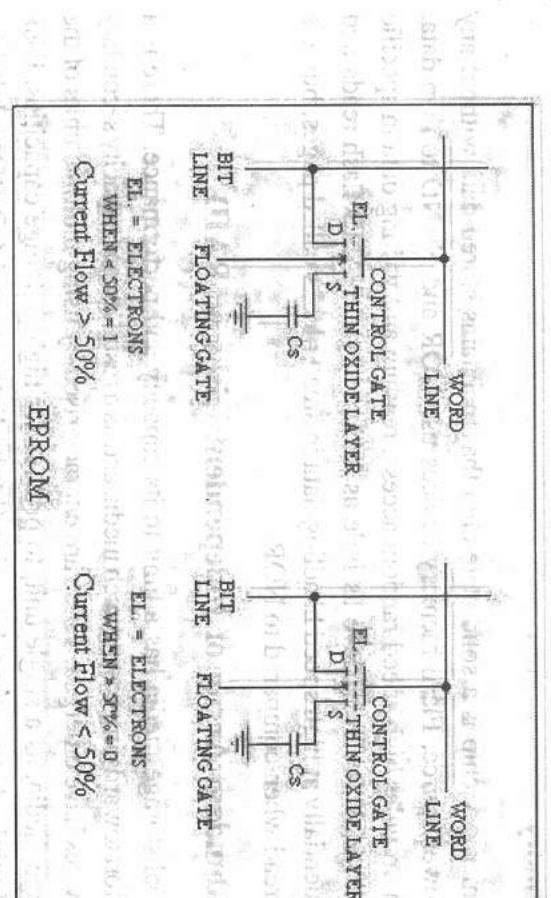


Fig. 2.35 Structure of EEPROM

This charge causes the floating gate transistor to act like an electron gun. The charged electrons are pushed through and trapped on the other side of the thin oxide layer giving it a negative charge. These negatively charged electrons act as a barrier between the control gate and the floating gate. A device called a cell sensor monitors the level of the charge passing through the floating gate. If the flow through the gate is greater than 50 per cent of the charge, it has a value of 1. When the charge passing through drops below the 50 per cent (Current Flow < 50 %) threshold, the value changes to 0. A blank EEPROM has all of the gates fully open giving each cell a value of 1. To rewrite an EEPROM, you must erase it first. To erase it, you must supply a level of energy strong enough to break through the negative electrons blocking the floating gate. The following are two classes of non-volatile memory chips based on EEPROM technology:

- **Ultra Violet Erase EEPROM:** The original erasable non-volatile memories were EEPROM's. They were readily identified by the distinctive quartz window in the center of the chip package. These operated by trapping an electrical charge on the gate of a field-effect transistor in order to change a 1 to a 0 in memory.
- **One-Time Programmable EEPROM:** A One-Time Programmable or OTP is electrically an EEPROM but with the quartz window physically missing. PROM it can be written once it cannot be erased.

Electrically Erasable PROM

Electrically Erasable PROM or EEPROM can be selectively erased without the need to remove the chip from the circuit. While an erase and rewrite of a location appears nearly instantaneous to the user, the write process is slightly slower than the read process; the chip can be read at full system speeds. EEPROMs are useful for storing settings or configuration for devices ranging from dial-up modems to satellite receivers.

The flash memory chip is a solid-state chip that maintains stored data without any external power source. Flash memory devices use NOR and NAND to map data. NOR flash provides high speed random access, reading and writing data in specific memory locations; it can retrieve as little as a single byte. NAND flash reads and writes sequentially at high speed handling data in small blocks called pages, but it is slower on read when compared to NOR.

2.3.6 Redundant Array of Independent Disks or RAID

Any physical storage media has a limit to its capacity and performance. There is a constant effort towards improving such media and as a result, larger capacity secondary storage devices have emerged. These are characterised by using multiple units of the same storage media, as a single unit, to provide for higher storage capacities. Disk Arrays (Multiple disks), Tape Libraries (Multiple tapes) and CD-ROM Jukebox (Multiple CDs) are the three most commonly used mass storage devices.

Mass data storage devices are characterized by relatively slow access time. This is because additional time in terms of first locating the desired disk, tape, or CD-ROM (as the case maybe) needs to be accounted for. However, they are more cost effective in case of applications that require huge storage capacity and for which rapid access to data is not the prime consideration.

They can also be used for off-line or archival storage of information/data since they can support huge volumes of information/data to be backed up.

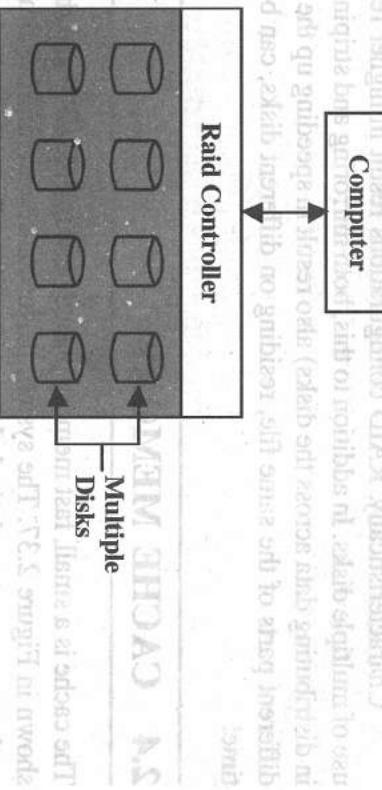
RAID is the abbreviation of Redundant Array of Independent Disks and originally it was referred as Redundant Array of Inexpensive Disks. It is a storage technology that combines multiple disk drive components into a logical unit. Data is distributed across the storage drives in a number of ways termed as 'RAID levels' depending on what level of redundancy and performance is required using parallel communication. RAID is nowadays used as an umbrella term for computer data storage methods that can divide and replicate data between multiple physical drives. Typically, the different methods or architectures are named by the word RAID followed by a number, such as RAID 0, RAID 1, RAID 0+1 and RAID5. Each method provides a specific equilibrium between the key goals, namely reliability, availability, performance and capacity. The levels of RAID higher than RAID level 0 provide protection against unrecoverable sector read errors in addition to whole disk failure.

Disk array RAID (Redundant Array of Independent Disks) is an acronym for a disk array and consists of a number of hard disks and disk drives with a controller in a single box.

Figure 2.36 illustrates the structure of RAID that consists of 8 disks.

Computer

Computer consists of several components. Fig. 2.36 A RAID Consisting of 8 Disks



The basic idea of RAID was to combine multiple small, inexpensive disk drives into an array of disk drives which yields performance exceeding that of a Single Large Expensive Drive (SLED). Additionally, this array of drives appears to the computer as a single logical storage unit or drive.

The concept was pioneered through academic research funded by Digital Equipment Corporation (DEC) and has now become a standard in the computing industry for applications requiring fast, reliable storage of large volumes of data.

There are several different types of RAID configurations that are described in terms of 'levels'. The various levels of RAID storage are as follows:

- **RAID 0:** Data is split across drives, resulting in higher data throughput. Since no redundant information is stored, performance is very good but the failure of any disk in the array results in data loss. This level is commonly referred to as striping.
- **RAID 1:** It provides redundancy by writing all data to two or more drives. The performance of a level 1 array tends to be faster on reads and slower on writes compared to a single drive, but if either drive fails, no data is lost. This level is commonly referred to as mirroring. Mirroring is the most expensive RAID option (since it doubles storage requirements), but it offers the ultimate in reliability.
- **RAID 0+1:** It is a combination of striping and mirroring. This configuration provides optimal speed and reliability, but possesses the same cost problem as RAID1.
- **RAID 5:** It employs a combination of striping and parity checking. The use of parity checking provides redundancy without the overhead of having to double disk capacity. Simply put, parity checking involves determining whether each given block has an odd or even value. These values are summed across the stripe sets to obtain a parity value. With this parity value, the contents of a failed disk can easily be determined and rebuilt on a spare drive.

These are other RAID configurations in addition to the ones described, but these are the ones most commonly used in the industry.

NOTES

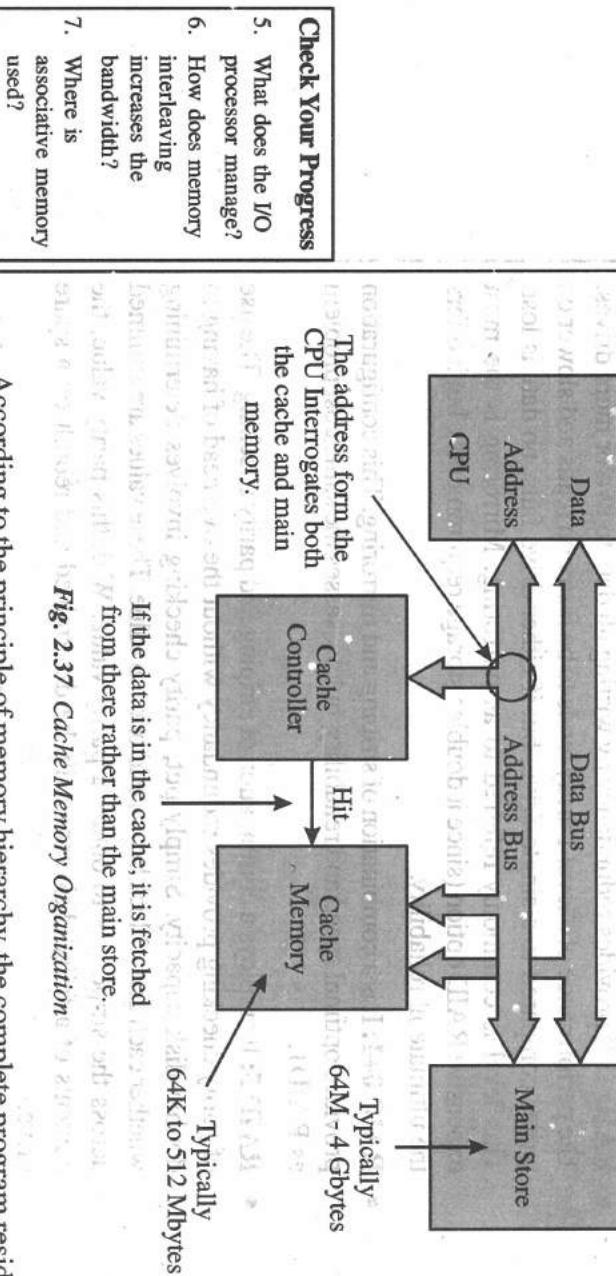
Characteristically, RAID configurations result in higher reliability due to the use of multiple disks. In addition to this, both mirroring and striping (techniques used in distributing data across the disks) also result in speeding up the read process since different parts of the same file, residing on different disks, can be read at the same time.

NOTES

2.4 CACHE MEMORY

The cache is a small, fast memory placed between the CPU and the main memory as shown in Figure 2.37. 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. In a system that uses a tiny 512MB 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 8KB to 32KB became common. By the end of the 1990s, multilevel cache configuration became common. The multilevel chip has one cache of capacity up to 128KB internal on the chip and other is external to chip and form second level caches having capacity up to 1MB.

In Figure 2.37, 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.



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, 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.

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:

$$\begin{aligned}\text{Access time of main memory} &= t_m \\ \text{Access time of cache memory} &= t_c \\ \text{Hit ratio of cache memory} &= h \\ \text{Miss ratio of cache memory} &= m \\ \text{Speedup ratio} &= S\end{aligned}$$

The average time to access a data is calculated as follows:

$$\text{Average Time to Access a Data} = \text{Hit Ratio} * \text{Cache Access Time} + ((1 - \text{Hit Ratio}) * (\text{Memory Access Time}))$$

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 'stall' 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.

Example 2.2: Let us consider a system which has a cache 'hit rate' of 95%, cache memory of 20 ns cycle time and main memory of 150 ns cycle time. The average cycle time of the system will be:

$$t_{avg} = (0.95) * 20 \text{ ns} + 0.05 * 150 \text{ ns} = 26.5 \text{ ns}$$

Solution: The effective memory cycle time as a function of cache hit rate for the above system is summarized in Table 2.4.

NOTES

Cache hit %	Effective cycle time (ns)
80	46
35	39.5
90	33
95	26.5
100	20

From the Table 2.4, 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 $N(ht_c + nt_m)$ 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 this case, this ratio is of the order .02. Therefore,

$$S = \frac{\frac{t_m}{t_m}}{\frac{ht_c + (1 - h)\frac{t_m}{t_m}}{t_m}} = \frac{1}{hk + (1 - h)}$$

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; that is, it is 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 $\frac{1}{2}$ Mbytes. 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:

$$t_{\text{cache}} = hL_1 t_{c1} + tL2 t_{c2} + (1 - hL_1 - hL_2) t_{\text{memory}}$$

Here, 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.

NOTES

Structure of Cache Memory

The Computer System

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 (refer Figure 2.38). 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 in prefetching of data.

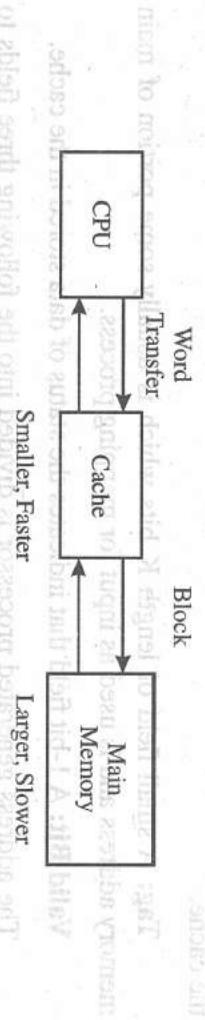


Fig. 2.38 Transfer of Information between CPU, Cache and Memory

Figure 2.39 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 $C \times K$ 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 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.

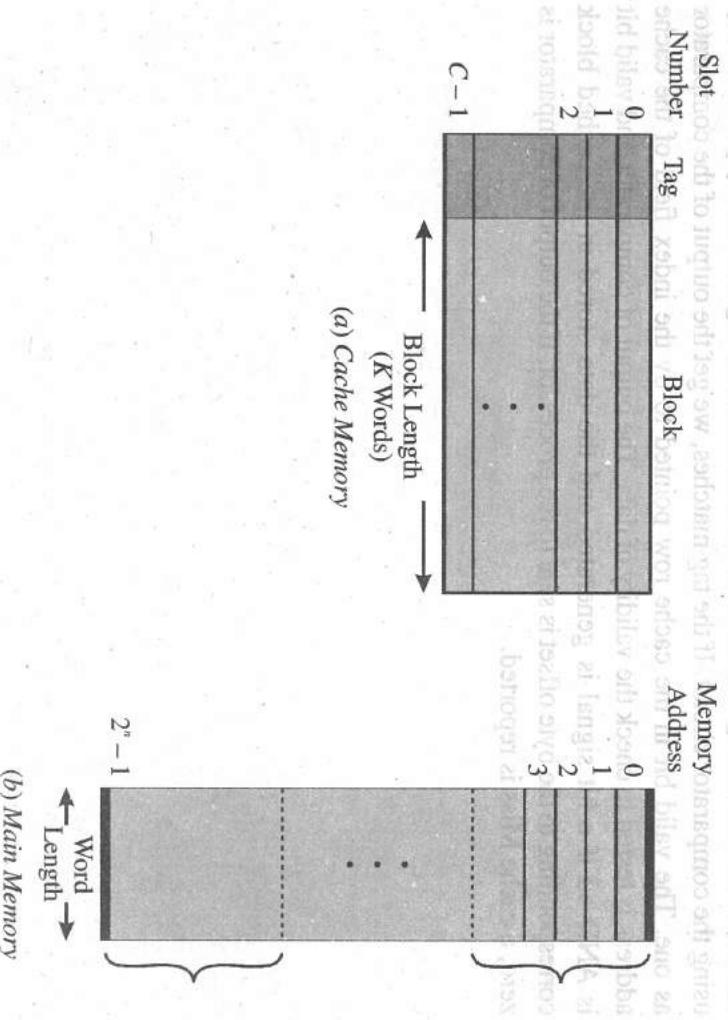


Fig. 2.39 Structure of Cache/Main Memory

NOTES

NOTES

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, 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 2.39, 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.

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 2.40. 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 AND-ed 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.

31 30 ... 13 12 11 ... 2 1 0 Byte Offset

of main memory

Tag bits Index 0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10

20 10

to select block. Cache Address

NOTES

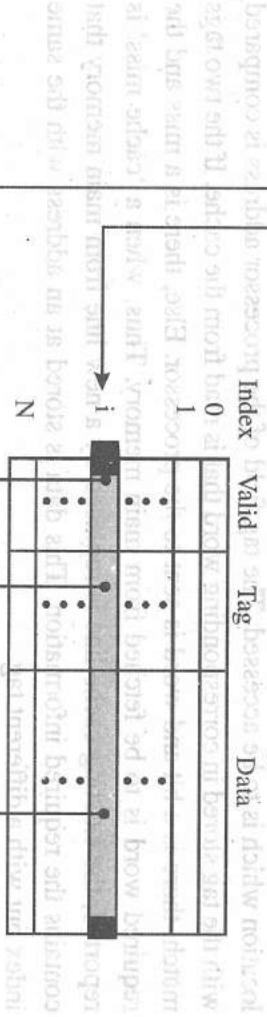


Fig. 2.40 Schematic Diagram of a Cache

Techniques of Mapping Addresses

The following three types of mapping techniques are popularly used:

- Direct-mapped caches
- Associative mapping
- Set associative mapping

Let us discuss each of them in detail.

Direct-Mapped Caches

The simplest way of organizing a cache memory is to employ direct mapping. It is based on a simple algorithm to map data block from the main memory into data block in the cache. There is a one-to-one correspondence between each block of data in the cache and memory blocks. Thus to find a memory block there is one and only one place in the cache where it is stored.

The address generated is divided into two fields, namely *index* and *tag*. The number of bits in the index field is equal to the number of address bits required to access the cache memory. Let us consider an example that we have 2^n words in main memory and 2^k words in cache memory. The n -bit address of main memory is divided into two fields: index and tag. The low-order k bits, referred to as the index field, correspond to address of a word stored in the cache. The remaining $n-k$ high-order bits are referred to as the *tag field*. As already seen, each line in cache memory consists of data word and its associated tag. As more than one word is stored in each cache, we need offset information to retrieve the required word. Thus, index field is further

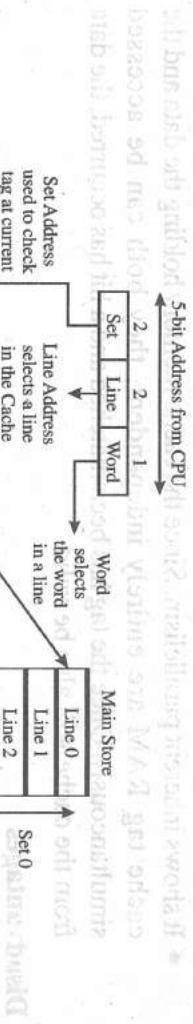
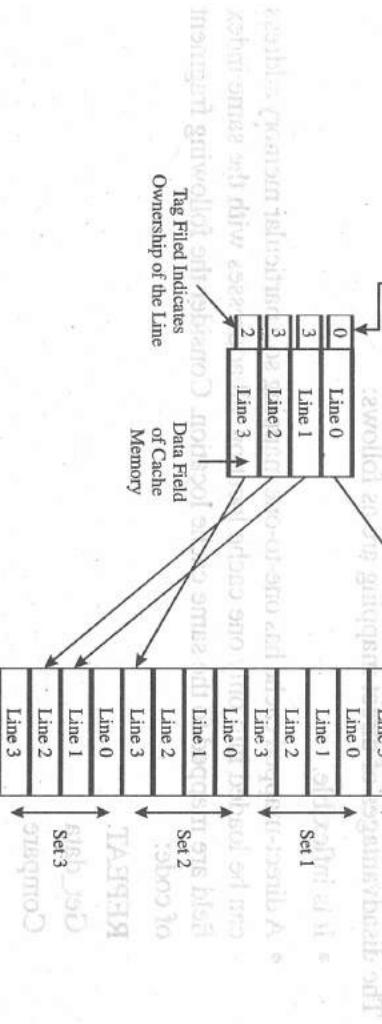
divided into the *slot* field and *offset* field. The slot field points to a particular block of the cache, while the offset field identifies a particular memory word in that block. When a block is written on the cache from the memory, tag field corresponding to main memory is stored in the *tag field* of the cache slot.

When CPU generates an address, the index field tells us about the cache location which is to be accessed. The tag field of the processor address is compared with the tag stored in corresponding word that is read from the cache. If the two tags match, there is a hit and word is sent to the processor. Else, there is a miss and the required word is to be fetched from main memory. Thus, when a ‘cache miss’ is reported, the existing cache line is replaced by a new line from main memory that contains the required information. This data is stored at an address with the same index but with a different tag.

Lets us understand how direct mapping is implemented with the following simple example (refer Figure 2.41). Let the memory is composed of 32 words, i.e., its address has 5-bit in it. This 5-bit address is composed of 2 bits to store tag field and 2 bits to represent slot line in cache. The last bit is the offset filed. The cache memory has 4 lines (2^2) such that each line stores two words. When the processor generates an address, the appropriate line in the cache is accessed. For example, if the processor generates the 5-bit address 1110_2 , the 4th line of cache memory (Line 3 in the Figure 2.41) will be accessed from which, if tag matches, the first word will be retrieved. Now consider the main memory space which is divided into sets of equal size where the size is same as that of cache. Figure 2.41 reveals that there are four possible lines that can occupy 4th line of the cache, i.e., 4th line in set 0, in set 1, in set 2 and set 3. In this example, the processor accesses line 4 in set 4. Now the question arises: ‘How does the system resolve this issue?’

Figure 2.41 shows how a direct-mapped cache resolves the contention between lines. As we know each line in cache has a tag which identifies the set to which this particular line belongs. When the processor accesses line 3 of cache, the tag value stored in this line will be sent to a comparator as one input and the other input will come from set field for address generated by processor. In this example, the tag field stored in the 4th line is 2, while the address bit generated processor as 3 stored in the tag field. Thus, the two are not same and a miss will be reported. However, if the two are same, a hit occur and the line in the cache is the desired line. Figure 2.42 provides a skeleton structure of a direct-mapped cache memory system for implementation.

NOTES

**NOTES****Advantages**

The advantages of direct mapping are as follows:

- In direct mapping, a block of main memory is always loaded into the same cache line in the cache.
- The cache memory and the cache tag RAM are widely available devices.
- No complex line replacement algorithm is required. If line 3 in set 4 is to be accessed and tag field is 2, a miss takes place. Line 3 from set 4 of the main memory is loaded into line 3 in the cache memory and the tag is set to 4. This scheme involves no decision to be taken regarding which line has to be rejected when a new line is to be loaded.

Fig. 2.43 Example of Direct-Mapped Cache

- It shows inherent parallelism. Since the cache memory holding the data and the cache tag RAM are entirely independent, they both can be accessed simultaneously. Once the tag has been matched and a hit has occurred, the data from the cache will also be valid.

NOTES

Disadvantages

The disadvantages of direct mapping are as follows:

- It is inflexible.
- A direct-mapped cache has one-to-one mapping so a particular memory address can be loaded into only one cache location, all addresses with the same index field are mapped to the same cache location. Consider the following fragment of code:

```
REPEAT
    Get_data
    Compare
UNTIL match OR end_of_data
```

Suppose the Get_data routine and compare routine use two blocks with both the blocks having the same index but different tags. A cache Miss will be reported every time whenever the code is repeatedly accessed. Thus, the performance of a direct-mapped cache in the above circumstances will be very poor. However, statistical results show that the chances of such worst-case behaviour of direct-mapped caches will be very low probability. Thus, in over all real programs, there is no significant impact of such cases on the average behaviour of the system.

Example 2.3: How can direct mapping be implemented for 32-bit memory addresses, 2 K cache slots, each of which can store 8 words?

Solution: The total cache size is $2 \text{ K} \times 8 = 16 \text{ K}$.

The last 3 bits are the offset bits which tell us which of the 8 words must be accessed within a cache slot. Therefore, the word field is 3 bits. A cache has 2K slots, hence requiring 10 address bits to identify a given slot. Thus, there are 11 bits in the slot field ($11+3=14$) from the index and the remaining bits ($32 - 14 = 18$) from the tag field (refer Figure 2.43).

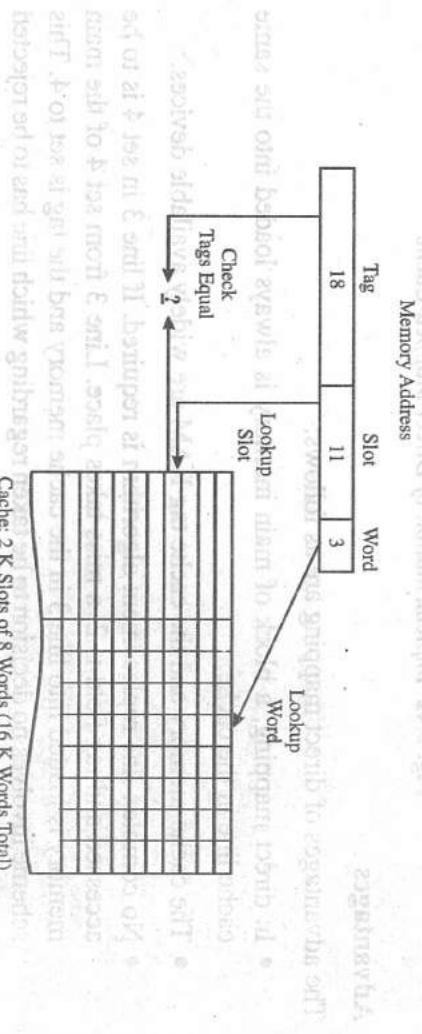


Fig. 2.43 Example of Direct-Mapped Cache

Associative Mapping

Another way of organizing a cache memory is associative cache memory, which is able to overcome the limitations of direct mapped cache, i.e., in it, there is no restriction on what data can be stored in the cache memory. An associative memory is the fastest and most flexible way of cache organization. It stores data as well as the address of the main memory location in the cache. An associative memory has an n-bit input. An address from the processor is divided into three fields: the *tag*, the *line* and the *word*. The mapping is done with storing tag information in n-bit argument register and comparing it with address tag in each location simultaneously. If the input tag matches a stored tag, the data associated with that location is output. Otherwise, the associative memory produces a Miss output.

Unfortunately, large associative memories are not yet cost effective. Once the associative cache is full, a new line can be brought in only by overwriting an existing line that requires a suitable line replacement policy. Associative cache memories are efficient because they place no restriction on the data they hold and permit to store any word from main memory to any location in the cache memory. Table 2.5 summarizes the data and their corresponding CPU addresses in associative cache.

Table 2.5 CPU Address in Associative Cache

CPU Address (argument register)	unmapped address	Data
Address 01101001		10010100
10010001		10101010

Figure 2.44 illustrates the structure of associative mapping in which any line in memory can map to any line in cache.

Fully Associative

Any line in memory can map to any line in Cache

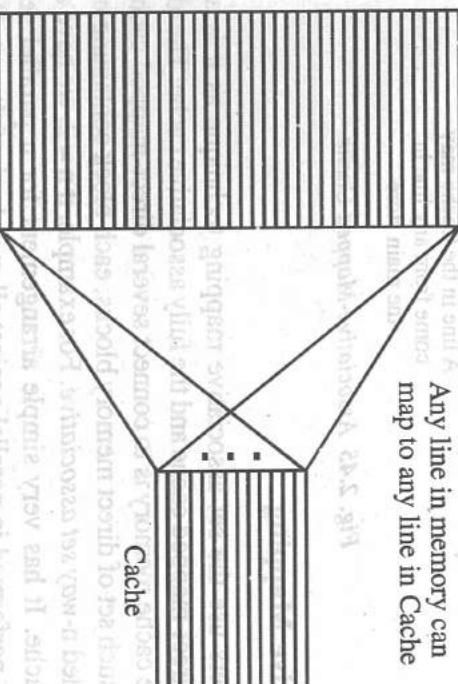


Fig. 2.44 Associative Mapping (Fully Associative Cache) In this mapping, any line in memory can map to any line in cache. It is very simple, cache (RAM) has to be very large to store all the data in memory. So simple, but expensive.

NOTES

NOTES

All comparisons are done simultaneously. So, the search is performed very quickly. Associative mapping is very expensive because each memory location must have both a comparator and a storage element. Like the direct mapped cache, the smallest unit of data transferred into and out of the cache in associative mapping is the line. Unlike the direct-mapped cache, there is no relationship between the location of lines in the cache and lines in the main memory.

When the processor generates an address, the word bits select a word location in both the main memory and the cache. The tag resolves which of the lines is actually present. In an associative cache, any of the 44 K lines in the main store can be located in any of the lines in the cache. Consequently, the associative cache requires a 14-bit tag to identify one of the 2^{14} lines from the main memory. Because the cache's lines are not ordered as well as the tags are not ordered, it may be anywhere in the cache or it may not be in the cache. In associative mapped cache, the Tag which is a field in the address bus is compared with all Tags in the cache memory simultaneously (refer Figure 2.45).

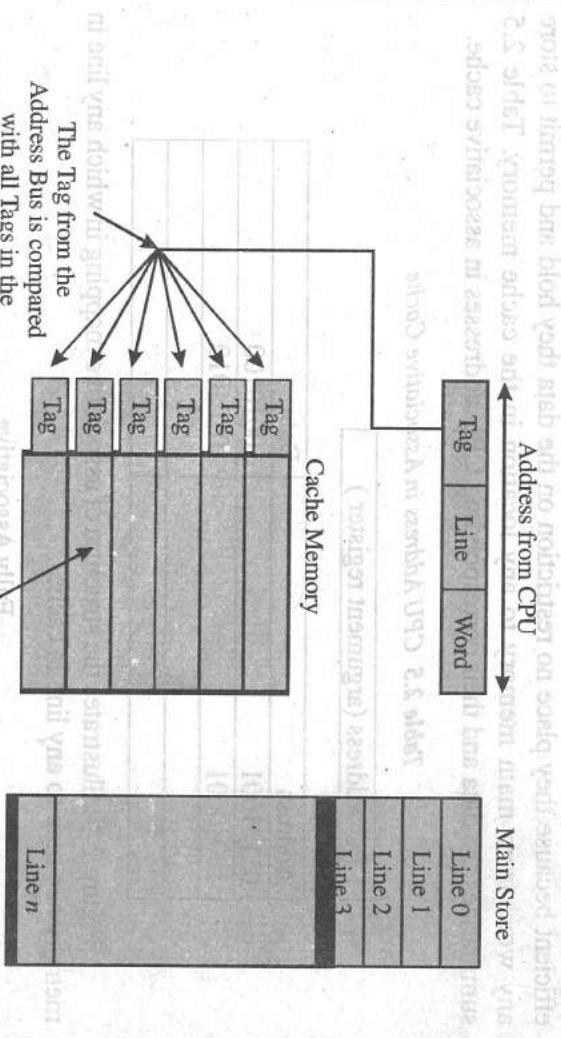


Fig. 2.45 Associative-Mapped Cache

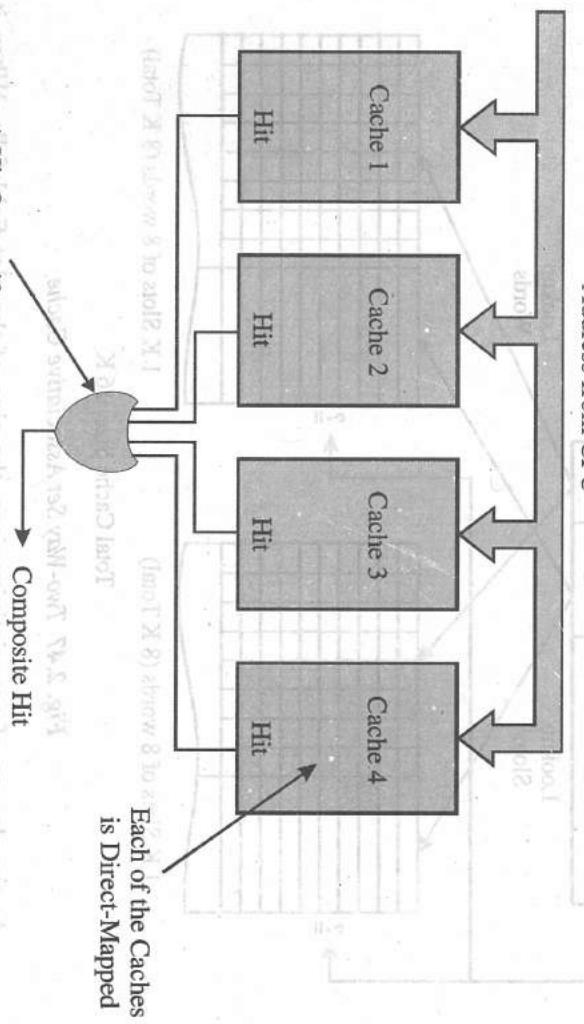
Set Associative Mapping

Most computers use the set associative mapping technique as it is a compromise between the direct-mapped cache and the fully associative cache. The idea behind the set associative cache memory is to connect several direct-mapped caches in parallel. If there are n such set of direct memory blocks, each block containing n entries in the cache, it is called n -way set associative. For example, if $n = 2$, we have a two-way set associative cache. It has very simple arrangement for n parallel sets; an n -way comparison is performed in parallel against all members of the set. Usually $n = 2^k$, where $k = 1, 2, 4$ are chosen for a set associative cache ($k = 0$ corresponds to direct mapping). As n is small (typically 2 to 14), the logic required to perform the comparison is not complex. This is a widely used technique. For example, 80486 uses 4-way set

associative and P4 uses 2-way set associative for the instruction cache and 4-way for the data cache.

Figure 2.46 describes the working of the 4-way set associative cache. When the processor accesses the cache memory, index bit tells the line number and the appropriate line is searched simultaneously in all four direct-mapped caches. The tags of all four lines are matched. If any one of it is matched, hit is reported. Thus, a set of tags associated with the line needs to be matched. A simple associative match can be used to determine which of the lines in cache is to supply the data. In the figure, the hit output from each direct-mapped cache is input of an OR gate which generates a composite hit. Hence, there will be a hit if any one the set contains the data.

Fig. 2.46 Set Associative-Mapped Cache



A Hit Occurs if any one cache has a matching tag. A miss occurs if none of the four Caches responds to an Access.

Effectively, the set associative mapping involves the division of the cache into a number of smaller caches, each of which may contain the word. All these caches are searched simultaneously. Thus, the set associative mapping does not require any special memory. Only a few small ordinary memories that are accessed in parallel are required. In a 2-way set associative cache, each line will store two tag values and two data block. In a 2-way set associative memory if 3 bits are required for storing tag and 16 bit for data, each cache line will have $2 \times (3+16) = 38$ bits. If there are 512 lines, in the cache 9 bits are required as index address and the total size of cache will be $512 \times 38 = 19456$ bits. It can accommodate 1024 main memory words since there are 512 lines with 2 words in each of them ($512 \times 2 = 1024$).

In a 2-way set associative cache, if cache miss is reported and one data word for a particular index is empty, then it is filled. If both data words are filled, then one must be overwritten by the new data. Similarly, in an n-way set associative cache, if all n data and tag fields in a set are filled, then one value in the set must be overwritten,

NOTES

or replaced, in the cache by the new tag and data values. There are many replacement algorithms, such as LRU, FIFO, etc., that are used for finding the word to be replaced. In order to implement it, extra bits are required in each word of the cache.

NOTES

Let us study an example of 2-way set associative cache having total capacity of 14 K as shown in Figure 2.47. Each line contains 8 words so the cache containing a total of 2 K lines, or 1 K sets, each set being 2-way associative. The sets correspond to the rows in the Figure 2.47. Thus, in this cache organization, a given memory location can be mapped to any of the two cache locations.

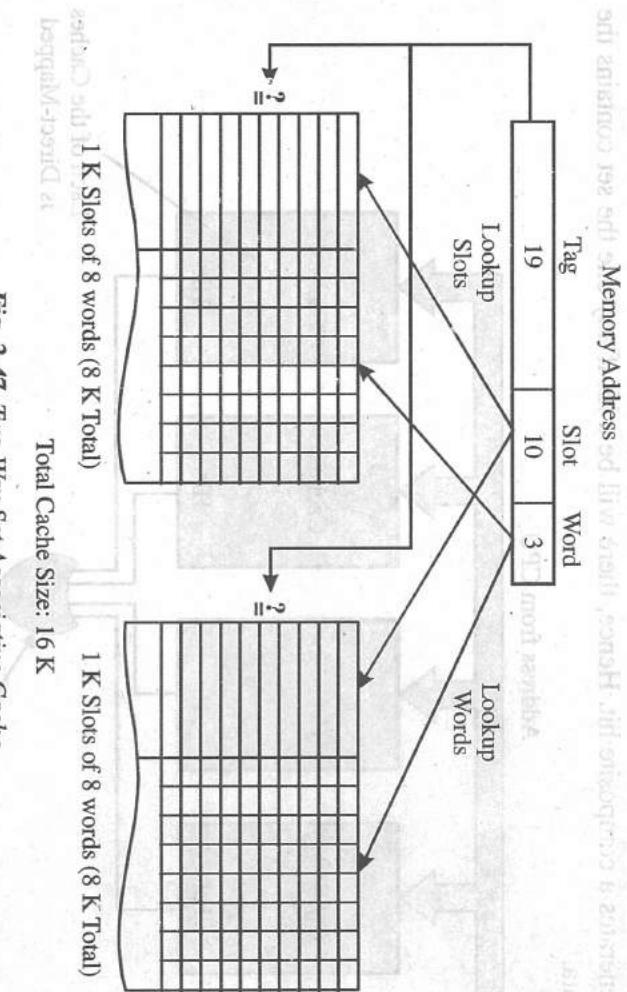


Fig. 2.47 Two-Way Set Associative Cache

As the degree of associativity rises, the size of the slot field falls. When it reaches zero, we will have a fully-associative cache. At the other extreme, you will notice that a 1-way set-associative cache is just a direct-mapped cache. In general, direct-mapped and fully-associative caches are special cases of the more general set-associative cache. Table 2.6 summarizes a comparison between various mapping functions.

Table 2.6 Mapping Function Comparison Table

Cache Type	Hit Ratio	Search Speed
Direct Mapped	Good	Best
Fully Associative	Best	Moderate
N-Way Set Associative, N>1	Very Good, Better as N Increases	Good, Worse as N Increases

2.4.1 Cache Memory Principles

In general cache memory works by attempting to predict which memory the processor is going to need next, and loading that memory before the processor needs it, and saving the results after the processor is done with it. Whenever the byte at a given memory address is needed to be read, the processor attempts to get the data from the cache memory. If the cache does not have that data, the processor is halted while it is

loaded from main memory into the cache. At that time memory around the required data is also loaded into the cache. Figure 2.48 illustrates the flowchart of cache read operation.

Diagram illustrating the cache read operation flowchart.

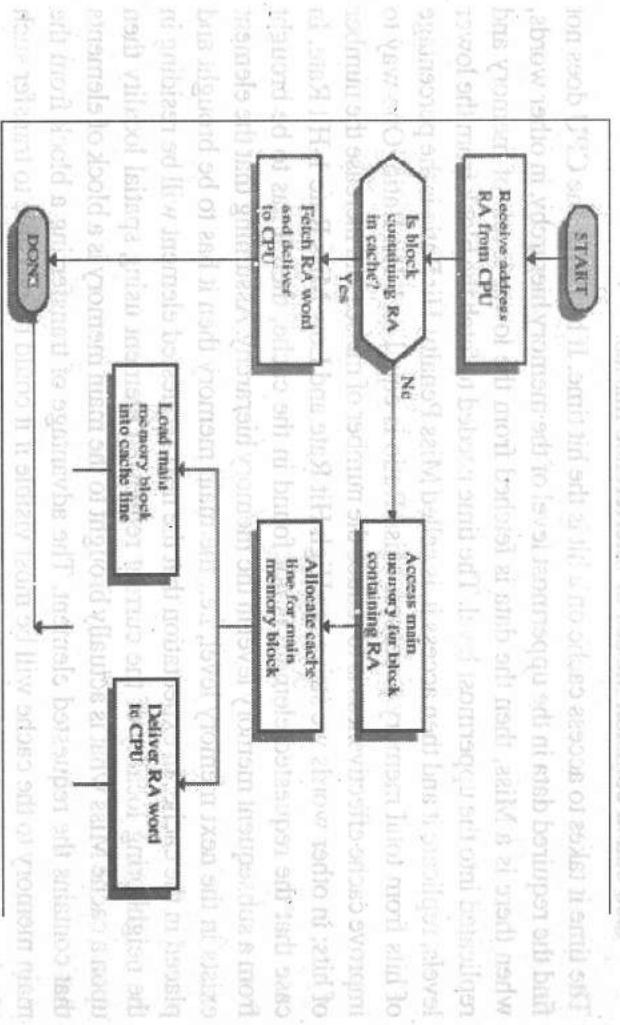


Fig. 2.48 Cache Read Operation

The basic principle that cache technology is based upon is locality of reference in which programs tend to access only a small part of the address space at a given point in time. This notion has three underlying assumptions: temporal locality, spatial locality and sequentiality.

- Temporal locality states that referenced memory is likely to be referenced again soon. In other words, if the program has referred to an address it is very likely that it will refer to it again.
 - Spatial locality states that memory close to the referenced memory is likely to be referenced. This means that if a program has referred to an address, it is very likely that an address in close proximity will be referred to soon.
 - Sequentiality refers to the future memory access is very likely to be in sequential order with the current access.
- The effectiveness of the cache is determined by the number of times the cache successfully provides the required data. This is the place to introduce some terminology. A hit is the term used when the data, required by the processor is found in the cache. If data is not found, we have a cache miss. There are three types of misses:
- **Compulsory Misses:** In this approach, the first access to a block is not in the first cache, so the block must be brought into the cache. These are also called cold start misses or first reference misses.
 - **Capacity Misses:** In this approach, if the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur due to blocks being discarded and later retrieved. These are misses due to cache size.
 - **Associativity Misses:** In this approach, when a block is removed before it can be used again in the cache and new data is stored in the same address space.

NOTES

- **Conflict Misses:** In this approach, if the block placement strategy is set associative or direct mapped then conflict misses will occur because a block can be discarded and later retrieved if too many blocks map to its set. These are also called collision misses or interference misses.

NOTES

The time it takes to access cache on a hit is the hit time. If however, the CPU does not find the required data in the uppermost level of the memory hierarchy, in other words, when there is a Miss, then the data is fetched from the lower levels of memory and replicated into the uppermost level. The time needed to fetch the block from the lower levels, replicate it and then access it, is called Miss Penalty. Hit Ratio is the percentage of hits from total memory accesses. Miss Ratio is equal to 1-Hit Ratio. One way to improve cache effectiveness is to reduce the number of misses and increase the number of hits; in other words we strive for High Hit Rate and Low Miss Rate 1-Hit Rate. In case that the requested element is not found in the cache, then it has to be brought from a subsequent memory level in the memory hierarchy. Assuming that the element exists in the next memory level, i.e., the main memory then it has to be brought and placed in the cache. In expectation that the next requested element will be residing in the neighboring locality of the current requested element using spatial locality then upon a cache Miss what is actually brought to the main memory is a block of elements that contains the requested element. The advantage of transferring a block from the main memory to the cache will be most visible if it could be possible to transfer such block using one main memory access time.

2.4.2 Elements of Cache Design

In order to improve the cache performance, Hit Ratio is increased. Many different techniques have been developed and the following are the most commonly used elements for cache design.

Cache Size

One way to decrease Miss rate is to increase the cache size. The larger the cache, the more information it will hold, and the more likely a processor request will produce a cache Hit, because fewer memory locations are forced to share the same cache line. However, applications benefit from increasing the cache size up to a point, at which the performance will stop improving as the cache size increases. In general the cache has to be small enough so that the overall average ‘Cost Per Bit’ is close to that of main memory alone and large enough so that the overall average access time is close to that of the cache alone. There are several other reasons to minimize the size of the cache. The larger the cache, the larger the number of gates involved in addressing the cache. The result is that large caches tend to be slightly slower than small ones even when using the same integrated circuit technology and put into the same place on the chip. Cache size is also limited by the available space on the chip. Typical cache sizes today range from 1K words to several mega-words. Multitasking systems tend to favor 256K words as nearly optimal size for main memory cache.

Mapping Function

The mapping function is used between main memory blocks and cache lines. Since each cache line is shared between several blocks of main memory the number of memory blocks $>>$ the number of cache lines. When one block is read in another one

should be moved out. Mapping functions minimize the probability that a moved-out block will be referenced again in the near future. There are three types of mapping functions: direct, fully associative and n-way set associative.

Cache Line Replacement Algorithms

NOTES

When a new line is loaded into the cache, one of the existing lines must be replaced. In a direct mapped cache, the requested block can go in exactly one position, and the block occupying that position must be replaced. In an associative cache we have a choice of where to place the requested block and hence a choice of which block to replace. In a fully associative cache, all blocks are candidates for replacement. In a set associative cache, we must choose among the blocks in the selected set. Therefore a line replacement algorithm is needed which sets up well defined criteria upon which the replacement is made. A large number of algorithms are possible and many have been implemented. Four of the most common cache line replacement algorithms are as follows:

- **Least Recently Used:** The cache line that was last referenced in the most distant past is replaced.
- **First In, First Out:** The cache line from the set that was loaded in the most distant past is replaced.
- **Least Frequently Used:** The cache line that has been referenced the fewest number of times is replaced.
- **Random:** This cache line replacement algorithm is a randomly selected line from cache is replaced.

The most commonly used algorithm is LRU. LRU replacement is implemented by keeping track of when each element in a set was used relative to the other elements in the set. For a two-way set associative cache, tracking when the two lines were used can be easily implemented in hardware by adding a single bit (use bit) to each cache line. Whenever a cache line is referenced its use bit is set to 1 and the use bit of the other cache line in the same set is set to 0. The line selected for replacement at any specific time is the line whose use bit is currently 0. The principle of the locality of reference means that a recently used cache line is more likely to be referenced again. LRU tends to give the best performance. The FIFO replacement policy is again easily implemented in hardware by the cache lines as queues. The LFU replacement algorithm is implemented by associating with each cache line a counter which increments on every reference to the line.

Cache Write Policies

Before a cache line can be replaced, it is necessary to determine if the line has been modified. The contents of the main memory block and the cache line that corresponds to that block are essentially copies of each other and should therefore hold the same data. If cache line has not been modified since its arrival in the cache, then the main memory block is updated to correspond as required prior to its replacement. The incoming cache line can simply overwrite the existing cache memory. On the other hand, if the cache line has been modified, at least one write operation has been performed on the cache line and thus the corresponding main memory block must be updated. Basically, there are two different policies that can be employed to ensure

that the cache and main memory contents are the same: write-through and write-back.

NOTES

- **Write-Through:** Assuming a cache Hit (a write Hit), the information is written immediately to both the line in the cache and to the block in the lower level memory with its normal wait-state delays. The advantages of this technique are that the contents of the main memory and the cache are always consistent, it is easy to implement and read Miss never results in writes to main memory.

On the other hand, the write through policy has a significant drawback. It needs a main memory access, which is slower and results in a more memory bandwidth usage. In spite of this, most Intel microprocessors use a write-through cache design.

- **Write-Back:** This approach sometimes called a *posted write* or *copy back* cache. On a cache Hit, the information is written only to the line in the cache. This allows the processor to immediately resume processing. The modified cache line is written to main memory only when it is replaced. To reduce the frequency of writing back blocks on replacement, a *dirty bit* is commonly used. This status bit indicates whether the block is dirty (modified while in the cache) or clean (not modified). If it is clean the block is not written on a miss. The advantages of the write-back policy are that writes occur at the speed of the cache memory, multiple writes within a block require only one write to main memory, which results in less memory bandwidth usage.

Block and Line Size

Another element in the design of a cache system is that of the line size. This is the number of bytes per cache line, sometimes also referred to as block size. When a block of data is retrieved from main memory and placed in the cache, not only the requested word is loaded but also some number of adjacent words those in the same block is retrieved. As the block size increases from very small to larger sizes, the hit ratio will at first increase because of the principle of locality. However, as the block becomes even bigger the hit ratio will begin to decrease because the probability of using the newly fetched information will be less than the probability of reusing the information that has been replaced. The relationship between block size and the hit ratio is complex and depends heavily on the locality characteristics of a particular program. No definitive optimum value has been found. A size from two to eight words seems to work reasonably close to optimum.

2.4.3 Improving Cache Performance

A cache is a component that transparently stores data so that future requests for that data can be served faster. The data that is stored within a cache might be values that have been computed earlier or duplicates of original values that are stored elsewhere. If requested data is contained in the cache (**Cache Hit**), this request can be served by simply reading the cache, which is comparatively faster. Otherwise (**Cache Miss**), the data has to be recomputed or fetched from its original storage location, which is comparatively slower. Hence, the greater the number of requests that can be served from the cache, the faster the overall system performance becomes.

The performance of cache memory is frequently measured in terms of a quantity called **Hit Ratio**. When the CPU refers to any word to the memory and finds the word in the cache memory, it is said to produce a Hit. If the word referred to by the CPU is not found in the cache memory, it is in the main memory and counts as a miss. Thus, the ratio of the total number of hits divided by the total CPU references to memory (Number of Hits + Number of Miss) is referred to as Hit Ratio.

NOTES

Let t_c be the cache access time where,
 t_m is the Hit ratio.

t_m is the main memory access time.

Then, the average access time is given by relation as follows:

$$= ht_c + (1 - h)(t_c + t_m)$$

The Hit ratio h always lies in the closed interval of 0 and 1.

The equation $= ht_c + (1-h)(t_c + t_m)$ is derived using the fact that when there is a cache Hit, the main memory is not accessed and in the event of a cache miss, both the main memory and cache will be accessed. Suppose, the ratio of main memory access time to cache access time is g , then an expression for the efficiency of a system that employs a cache can be derived as follows:

$$\text{Efficiency} \quad A = \frac{t_c}{t}$$

$$\begin{aligned} \text{Efficiency} &= \frac{t_c}{ht_c + (1-h)(t_c + t_m)} \\ &= \frac{t_c}{ht_c + (1-h)t_c + (1-h)t_m} \\ &= \frac{t_c}{t_c(h+1-g) + t_m(1-g)} \\ &= \frac{t_c}{t_c(h+1-g)\left(1+\frac{t_m}{t_c}\right)} \end{aligned}$$

$$\begin{aligned} \text{Efficiency} &= \frac{t_c}{t_c(h+1-g)\left(1+\frac{t_m}{t_c}\right)} \\ &\therefore \gamma = \frac{t_m}{t_c} \\ &= \frac{t_c}{t_c(h+1-g)\left(1+\frac{\gamma}{h+1-g}\right)} \\ &= \frac{1}{h+1-g+h+\gamma(1-h)} \\ &= \frac{1}{2h+1-\gamma} \end{aligned}$$

Thus bus ratio more equal to γ to reduce access time to cache.

2.5 INPUT/OUTPUT

A hardware unit that plays an important role between the CPU and peripheral devices to supervise the input and output transfer is known as input/output interface.

The input-output system of a computer processes the input information by converting them into computer readable binary form and then displaying the final result as output in user readable form. The processed result is provided to the users. Figure 2.49 will make the concept clear.

Check Your Progress

9. What is hit ratio?
10. What is the line size/block for the cache?
11. How is cache memory accessed?

NOTES

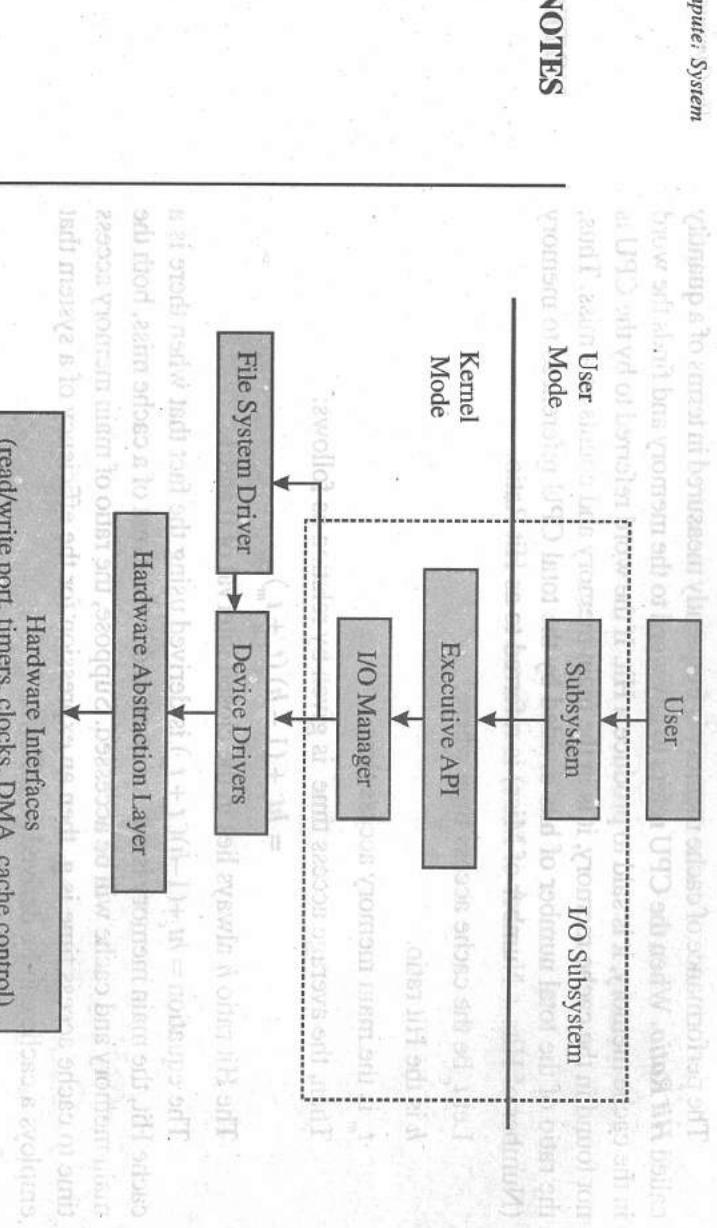


Fig. 2.49 Input/Output Interface Process

I/O interface provides a method for transferring information between internal storage and external I/O devices. Peripherals connected to a computer need special communication links for interfacing with the central processing unit. The purpose of the communication link is to resolve the differences that exist between the central computer and each peripheral.

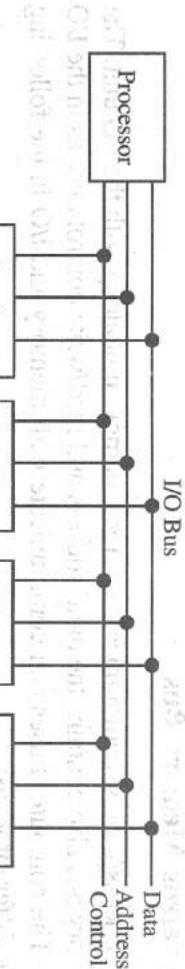
The major differences are as follows:

- Peripherals are electromechanical and electromagnetic devices and their manners of operations are different from the operation of the CPU and memory, which are electronic devices. Therefore, the conversions of signals are required.
- The data transfer rate of peripherals is usually slower than the transfer rate of the CPU and hence, a synchronization mechanism is needed.
- Data codes and formats in peripherals are different from the word format in the CPU and memory.

- The operating modes of peripherals are different from each other and each must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

To resolve these differences, computer systems include special hardware known as 'interface units' between the CPU and peripherals to supervise and synchronize all input and output transfers.

A typical communication link between the processor and peripherals is shown in Figure 2.50. The I/O bus consists of data lines, address lines and control lines.

**NOTES***Fig. 2.50 Communication Link between the Processor and Peripherals*

Each peripheral device has its associated interface unit. Each interface unit decodes the address and control received from the I/O bus, interprets them for the peripheral and provides signals for the peripheral controller. It also synchronizes the data flow and supervises the transfer between peripherals and processor. Each peripheral has its own controller that operates the particular electromechanical device. For example, the printer controller controls the paper motion, the printing timing and the selection of printing characters.

The I/O bus from the processor is attached to all peripherals interfaces. To communicate with a particular device, the processor places a device address on the address lines. Each interface attached to the I/O bus contains an address decoder that monitors the address lines. When the interface detects its own address, it activates the path between the bus lines and the devices that it controls. All peripherals whose address does not correspond to the address in the bus are disabled by their interfaces.

At the same time that the address is made available in the address lines, the processor provides a function code in the control lines. The interface selected responds to the function code and proceeds to execute it. These function codes are referred to as I/O commands. There are four types of commands that the interface may receive. They are as follows:

- **Control Command:** A control command is issued to activate the peripheral and to inform it what to do. The particular control command issued depends on the peripheral.
- For example, a magnetic tape unit may be instructed to back space the tape by one record, to rewind the tape or to start the tape moving in the forward direction.
- **Status Command:** A status command is used to test various status conditions in the interface and the peripherals.
- **Output Data Command:** A data output command causes the interface to respond by transferring data from the bus into one of its registers.
- **Input Data Command:** The data input command is the opposite of the data output command. In this case, the interface receives an item of data from the peripheral and places it in its buffer register. The processor checks if the data are available by means of a status command and then issues a data input command. The interface places the data on the data lines, where they are accepted by the processor.

The processor must communicate with the memory unit along with the I/O unit. The memory bus also contains the data, address and read/write control lines as in the I/O bus. The computer buses can communicate with memory and I/O in the following three different ways:

- Use two separate buses, one of memory and one for I/O.
- Use one common bus for both memory and I/O bus having separate control lines for each.
- Use one common bus and common control lines for both memory and I/O.

Isolated versus Memory-Mapped I/O

The transfer of information between the CPU and the memory or I/O is done by means of a common bus. By enabling the specific read or write lines, the CPU specifies whether the address on the address bus is for a memory word or for an interface register. During a memory transfer, the memory read or memory write control lines are enabled and during an I/O transfer, the I/O read or I/O write control lines are enabled. In this way, the I/O interface addresses are isolated from the memory addresses; this process is termed as isolated I/O method for assigning addresses in a common bus.

In isolated I/O configuration, the input and output instructions for both interface register and memory are different. When the CPU fetches and decodes the instructions for I/O interfaces, it enables the I/O read or I/O write control line and informs the other components attached to the common bus that the address in the address bus is for the interface register and not for the memory word. Similarly, when the CPU fetches and decodes the instruction from memory, it places the memory address on the address bus and enables the memory read or memory write control line, which informs the external components that the address is for the memory word and not for the I/O interface. Hence, the memory and I/O addresses are isolated in this method.

An alternative method is known as memory-mapped I/O. In this method, the computer uses the same address space for memory and I/O both. In memory-mapped I/O, there is no specific input or output instructions. The same instruction can be used for manipulating I/O data residing in interfaces and can also be used to manipulate memory words. Computers with memory-mapped I/O can use memory type instructions to access I/O data which allows the computer to use the same instructions for memory transfers or I/O transfers.

2.5.1 Peripheral Devices

The devices that are connected to a computer and controlled by the processor are called peripheral devices. The peripheral devices of a computer system are also known as I/O units. They provide an efficient way of communication between the computer system and the outside world. A computer must have a system to receive information (input) and must be able to communicate results (output) to the external world. Most common peripherals devices are keyboard, mouse, monitor, printer, etc.

Keyboard

A keyboard is the most familiar input device for entering information into a computer. The layout of the keyboard is similar to that of the traditional typewriter with some extra command and functions keys. When a key on the keyboard is pressed, a binary coded character is send to the computer and the corresponding character is displayed on the screen. If a wrong key is pressed, it can be erased and the correct character can be re-typed. The fastest possible speed of entering the information using keyboard depends on the person's typing speed. A general keyboard used in a computer system is shown in Figure 2.51.

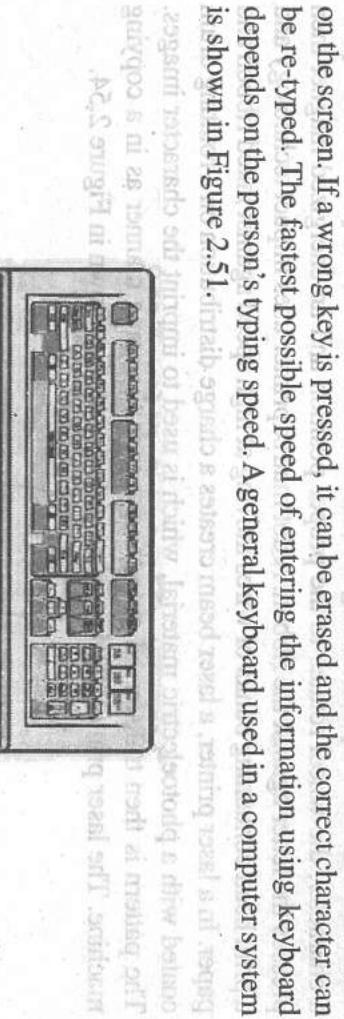


Fig. 2.51 A Keyboard

Mouse

A mouse is a pointing device. It can be moved on a smooth surface so that it points to the desired location on the screen. It can be optical or mechanical. A user can move the mouse, stop it at a desired location and with the help buttons can select choices. The mouse used in a computer system is shown in Figure 2.52.



Fig. 2.52 A Mouse

Display Devices

One of the most important peripherals of a computer system is display monitor. There are different types of display monitors but the most popular is Cathode Ray Tube (CRT). Others are Liquid Crystal Display (LCD) and Overhead Projection (OHP). CRT contains a beam of electrons (cathode rays) emitted by an electron gun which passes through focusing and deflection systems that direct the beam toward specified position on the phosphor-coated screen. The phosphor then emits a small spot of light at each position contacted by the electron beam. A monitor is shown in Figure 2.53.

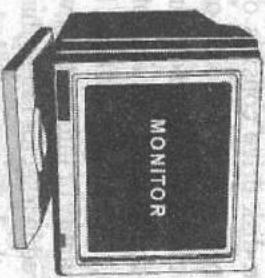


Fig. 2.53 Display Monitor

NOTES

2.2.4 Printers

Printers are used for getting the output on paper. Printers can be classified according to the print quality and the speed of printing. The three basic types of character printers are daisywheel, dot matrix and laser printers. A daisywheel printer contains a wheel and the characters are placed along the circumference. When the printer has to print a character, the wheel rotates to the proper position and an energized magnet then presses the letter against the ribbon. A dot matrix printer uses impact technology and a print head containing banks of wires moving at high speed against inked ribbon and paper. In a laser printer, a laser beam creates a charge distribution on a rotating drum coated with a photoelectric material, which is used to imprint the character images. The pattern is then transferred onto the paper in the same manner as in a copying machine. The laser printer used in a computer system is shown in Figure 2.54.

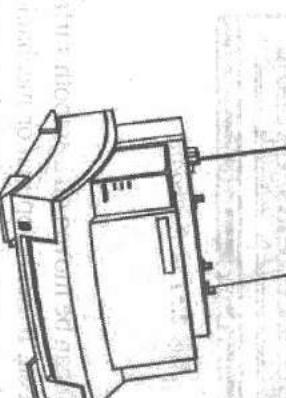


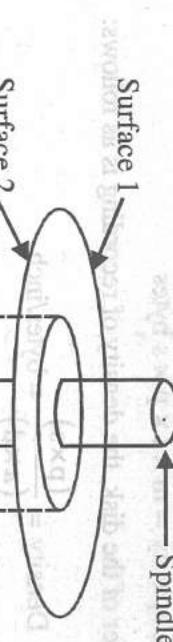
Fig. 2.54 A Printer

2.5.2 External Devices

The storage devices that provide backup storage are called auxiliary memory. RAM is a volatile memory and thus a permanent storage media is required in a computer system. Auxiliary memory devices are used in a computer system for the permanent storage of information and hence are the devices that provide backup storage. They are used for storing system programs, large data files and other backup information. The auxiliary memory has a large storage capacity and is relatively inexpensive, but has low access speed as compared to the main memory. The most common auxiliary memory devices used in computer systems are magnetic disks and tapes. Now optical disks are also used as auxiliary memory.

Magnetic Disk

Magnetic disks are circular metal plates coated with magnetized material on both sides. Several disks are stacked to a spindle one below the other with a read/write head to make a disk pack. The disk drive consists of a motor and all disks rotate together at very high speed. Information is stored on the surface of a disk along concentric sets of rings called tracks. These tracks are divided into sections called sectors. A set of corresponding tracks in all surfaces of a disk pack is called cylinder. Thus, if a disk pack has n plates, there are $2n$ surfaces, hence the number of tracks per cylinder is $2n$. The minimum quantity of information, which can be stored, is a sector. If the number of bytes to be stored in a sector is less than the capacity of the sector, the rest of the sector is padded with the last type recorded. Figure 2.55 shows a magnetic disk memory.

**NOTES**

Surface 1
Surface 2
Surface 3
Surface 4

Spindle

Read/write head

Cylinder

Lateral Layer

Surface 1 shows a magnetic disk with four concentric tracks. A read/write head is positioned above the outermost track. The disk rotates around a central spindle. The diagram illustrates the concept of surfaces and cylinders in disk storage.

Fig. 2.55 Magnetic Disk A magnetic disk consists of two concentric circular plates called platters. Each platter has a number of concentric tracks. A read/write head is positioned above one of the tracks. The disk rotates around a central spindle. The diagram illustrates the concept of surfaces and cylinders in disk storage.

The subdivision of a disk surface into tracks and sectors is shown in Figure 2.56.



Magnetic Disk

Sector

Tracks

ReadWrite head

Read Write

A magnetic disk consists of two concentric circular plates called platters. Each platter has a number of concentric tracks. A read/write head is positioned above one of the tracks. The disk rotates around a central spindle. The diagram illustrates the concept of surfaces and cylinders in disk storage.

A track is a circular path on the surface of a disk. Suppose s bytes are stored per sector, there are p sectors per track, t tracks per surface and in surfaces. Then, the capacity of disk will be defined as follows:

Suppose s bytes are stored per sector, there are p sectors per track, t tracks per surface and in surfaces. Then, the capacity of disk will be defined as follows:

If d is the diameter of the disk, the density of recording is as follows:

$$\text{Density} = \frac{(p \times s)}{(\pi \times d)} = \text{bytes/inch}$$

NOTES

A set of disk drives are connected to a disk controller. The disk controller accepts commands and positions the read/write heads for reading or writing. When the read/write command is received by the disk controller, the controller first positions the arm so that the read/write head reaches the appropriate cylinder. The time taken to reach the appropriate cylinder is known as *Seek Time* (T_s). The maximum seek time is the time taken by the head to reach the innermost cylinder from the outermost cylinder or vice versa. The minimum seek time will be 0 if the head is already positioned on the appropriate cylinder. Once the head is positioned on the cylinder, there is further delay because the read/write head has to be positioned on the appropriate sector. This is rotational delay also known as *Latency Time* (T). The average rotational delay equals half the time taken by the disk to complete one rotation.

Floppy Disk

A floppy disk, also known as diskette, is a very convenient bulk storage device and can be taken out of the computer. It can be either 5.25" or 3.5" size, the 3.5" size being more common. It is contained in a rigid plastic case. The read/write heads of the disk drive can write or read information from both sides of the disk. The storage of data is in the magnetic form, similar to that in hard disk. The 3.5" floppy disk has storage up to 1.44 MB. It has a hole in the centre for mounting it on the drive. Data on the floppy disk is organized during the formatting process. The disk is organized into sectors and tracks. The 3.5" high-density disk has 80 concentric circles called tracks and each track is divided into 18 sectors. Tracks and circles exist on both sides of the disk. Each sector can hold 512 bytes of data plus other information like address, etc. It is a cheap read/write bulk storage device.

Magnetic Tapes

Magnetic disk is used by almost all computer system as a permanent storage device; however, magnetic tape is still a popular form of low-cost magnetic storage media and it is primarily used for backup storage purposes. The standard backup magnetic tape device used today is Digital Audio Tape (DAT). These tapes provide approximately 1.2 GB of storage on a standard cartridge-size cassette tape. These magnetic tapes memories are similar to that of audio tape recorders.

A magnetic tape drive consists of two spools on which the tape is wound. Between the two spools, there is a set of nine magnetic heads to write and read information on the tape. The nine heads operate independently and record information on nine parallel tracks, parallel to the edge of the tape. Eight tracks are used to record a byte of data and the ninth track is used to record a parity bit for each byte. The standard width of the tape is half an inch. The number of bits per inch (bpi) is known as *recording density*.

Normally, when data is recorded into the tape, a block of data is recorded and then a gap is left and then another block is recorded and so on. This gap is known as

Inter-Block Gap (IBG). The blocks are normally 10 times long as that of IBG. The Beginning Of Tape (BOT) is indicated by a metal foil known as start of tape marker and the End Of Tape (EOT) is also indicated by a metal foil known as end of tape marker.

The data on the tape is arranged as blocks and cannot be addressed. They can only be retrieved sequentially in the same order in which they are written. Thus, if a desired record is at the end of the tape, earlier records have to be read before it is reached and hence the access time is very high as compared to magnetic disks.

NOTES

Optical disk storage technology provides the advantage of high volume and economical storage with somewhat slower access times than traditional magnetic disk storage.

Optical Disks

Compact Disk-Read Only Memory (CD-ROM) optical drives are used for the storage of information that is distributed for read-only use. A single CD-ROM can hold up to 800 MB of information. Software and large reports distributed to a large number of users are good candidates for this media. CD-ROM is also more reliable for distribution than floppy disks or tapes. Nowadays, almost all software and documentations are distributed only on CD-ROM.

In CD-ROMs the information is stored evenly across the disk in segments of the same size. Therefore, in CD-ROMs, data stored on a track increases as we go towards the outer surface of disk and hence CD-ROMs are rotated at variable speeds for the reading process.

Information in a CD-ROM is written by creating pits on the disk surface by shining a laser beam. As the disk rotates, the laser beam traces out a continuous spiral. When 1 is to be written on the disk, a circular pit of around 0.8 micrometer diameter is created by the sharply focused beam and no pit is created if a zero is to be written. The pre-recorded information on the CD-ROM is read with the help of a CD-ROM reader, which uses a laser beam for reading. For this, the CD-ROM disk is inserted into a slot of CD drive. Then the disk is rotated by a motor. A laser head moves in and out to the specified position. As the disk rotates, the head senses pits and land, which is converted to 1's and 0's by the electronic interface and sent to the computer. Figure 2.57 depicts the tracks on a disk surface of a CD-ROM.

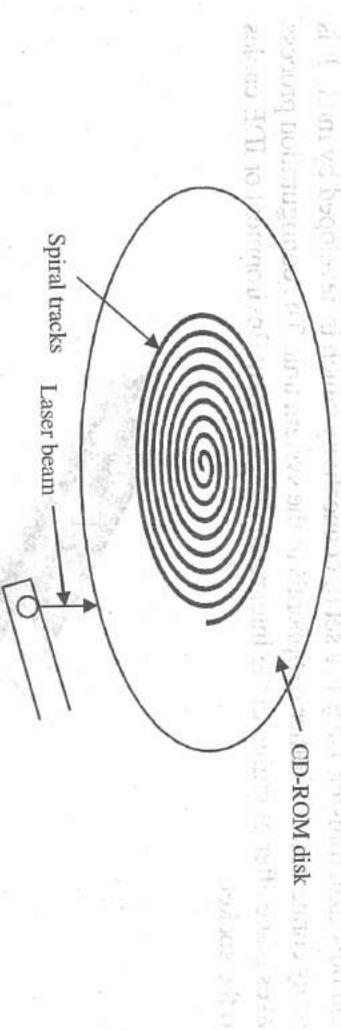


Fig. 2.57 Tracks on a Disk Surface

NOTES

The speed of the disk is indicated by $n \times$, where n is an integer indicating the factor by which the original nominal speed of 150 KB/S is multiplied. Thus, a $52 \times$ CD-ROM disk speed will be $52 \times 150 = 7800$ KB/S. CD-ROM has a buffer size of 256 KB to keep data temporarily. It is connected to the computer system by a Small Computer System Interface (SCSI) adapter.

The main advantages of CD-ROMs are as follows:

- Large data/information storage capacity.
- Mass replication is inexpensive and fast.
- These are removable disks.

Disadvantages of CD-ROMs are as follows:

- It is read-only and hence cannot be updated.
- Access time is longer than that of a magnetic disk.

Erasable Optical Disk

Recent development in optical disks is the erasable optical disks. They are used as an alternative to standard magnetic disks when speed of the access is not important and the volume of the data stored is large. They can be used for image, multimedia, a high-volume, low-activity backup storage. Data in these disks can be changed as repeatedly as in a magnetic disk. The erasable optical disks are portable and highly reliable and have longer life. They use format that makes semi-random access feasible.

The external devices directly associated with memory management. Universal Serial Bus (USB), external hard disk, pen drives, etc., are considered as external devices and in network era these devices are used frequently by the users.

Universal Serial Bus Device

Universal Serial Bus (USB) is considered as high speed serial bus and prime tool for external devices. Its data transfer rate is higher than that of a serial port. It supports interfaces, such as monitors, keyboard, mouse, speaker, microphones, scanner, printer and modems. It allows interfacing several devices to a single port in a daisy-chain. USB provides power lines along with data lines. USB cable contains four wires collectively. Two of them are used to supply electrical power to peripherals eliminating bulky power supply. The other two wires are used to send data and commands. USB uses three types of data transfer and they are isochronous or real time, interrupt driver and bulk data transfer. USB is a set of connectivity which is developed by Intel. It is easily connected with other peripherals to the system unit. The configuration process takes place after plugging in the Integrated Development Environment or IDE cables to the socket.



Fig. 2.58 Twin Universal Serial Bus Device

Figure 2.58 shows the prime external device known as twin universal serial bus device. Basically, it is a matched pair of USB devices that can be used to share information between two computers, securely.

Pen/Thumb Drive Device

The thumb drive or pen drive device is also considered as prime and frequently used external device. With the help of this drive you can share various documents, multimedia presentations, pictures, and Moving Picture Experts Group Audio Layer III or MP3 files by using this hot Plug-and-Play (PnP), high speed USB flash drive. Simply plug in your USB flash disk and transfer or backup the files. With capacities up to 16 GB or more you can share your important files, music and images.

NOTES

The I/O allows you to interface with your own electronics. It is the perfect tool to implement your own module. Figure 2.59 displays the structure of I/O modules. CPU consists of the interconnected functional units. These functional units are registers, ALU and control circuitry. Registers are temporary storage units within the CPU. Some registers, such as the program counter and instruction register have dedicated uses. Other registers, such as the accumulator, are for more general purpose use. The accumulator usually stores one of the operands to be manipulated by the ALU. A typical instruction might direct the ALU to add the contents of some other register to the contents of the accumulator and store the result in the accumulator itself. In general, the accumulator is both a source (operand) and a destination (result) register. The instructions that make up a program are stored in the system's memory. The central processor references the contents of memory, in order to determine what action is appropriate. This means that the processor must know which location contains the next instruction. Each of the locations in memory is numbered, to distinguish it from all other locations in memory. The number which identifies a memory location is called its address. The control circuitry is the primary functional unit within a CPU. Using clock inputs, the control circuitry maintains the proper sequence of events required for any processing task. After an instruction is fetched and decoded, the control circuitry issues the appropriate signals (to units both internal and external to the CPU) for initiating the proper processing action. Often the control circuitry will be capable of responding to external signals, such as an interrupt or wait request. An interrupt request will cause the control circuitry to temporarily interrupt main program execution, jump to a special routine to service the interrupting device then automatically return to the main program. A Wait request is often issued by a memory or I/O element that operates slower than the CPU. The control circuitry will idle the CPU until the memory or I/O port is ready with the data.

2.5.3 I/O Modules

The I/O allows you to interface with your own electronics. It is the perfect tool to implement your own module. Figure 2.59 displays the structure of I/O modules. CPU consists of the interconnected functional units. These functional units are registers, ALU and control circuitry. Registers are temporary storage units within the CPU. Some registers, such as the program counter and instruction register have dedicated uses. Other registers, such as the accumulator, are for more general purpose use. The accumulator usually stores one of the operands to be manipulated by the ALU. A typical instruction might direct the ALU to add the contents of some other register to the contents of the accumulator and store the result in the accumulator itself. In general, the accumulator is both a source (operand) and a destination (result) register. The instructions that make up a program are stored in the system's memory. The central processor references the contents of memory, in order to determine what action is appropriate. This means that the processor must know which location contains the next instruction. Each of the locations in memory is numbered, to distinguish it from all other locations in memory. The number which identifies a memory location is called its address. The control circuitry is the primary functional unit within a CPU. Using clock inputs, the control circuitry maintains the proper sequence of events required for any processing task. After an instruction is fetched and decoded, the control circuitry issues the appropriate signals (to units both internal and external to the CPU) for initiating the proper processing action. Often the control circuitry will be capable of responding to external signals, such as an interrupt or wait request. An interrupt request will cause the control circuitry to temporarily interrupt main program execution, jump to a special routine to service the interrupting device then automatically return to the main program. A Wait request is often issued by a memory or I/O element that operates slower than the CPU. The control circuitry will idle the CPU until the memory or I/O port is ready with the data.

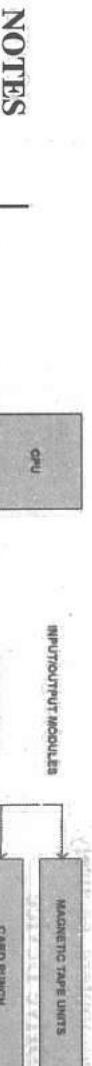


Fig. 2.59 Structure of I/O Modules

Data channels are independent I/O modules and contain their own processor and instruction set allowing these modules to take care of the I/O instructions, resulting in relieving the CPU of considerable processing burden. The multiplexer acts as a central termination point for the data channels, the CPU and memory. Scheduling access to each of these devices, it allows them to act independently of each other. An I/O module can exchange data directly with the processor. Just as the processor can initiate a read or write with memory, designating the address of a specific location, the processor can also read or write data to an I/O module. The processor identified a specific device that is controlled by a particular I/O module. In some cases, it is desirable to allow I/O exchanges to occur directly with memory. In such a case, the processor grants to an I/O module the authority to read from or write to memory so that the I/O memory transfer can occur without tying up the processor. During such a transfer, the I/O module issues read or write commands to memory, relieving the processor of responsibility for the exchange. This operation is known as Direct Memory Access or DMA.

2.5.4 Programmed I/O

Programmed I/O is the simplest method of I/O data transfer and it executes a program that contains dedicated I/O instructions. When the processor encounters an I/O instruction, it issues a command for the appropriate I/O module that executes the given instruction. It does not interrupt the processor but lets the I/O module to perform the requested action and, at the same time, sets the appropriate bits in the I/O status register which alerts the processor for further action. As interrupts are not used, the processor has to periodically check the status of the I/O module and find whether the required operation can be performed depending on the status of I/O module. The I/O program gets direct control to execute the I/O instruction. In order to execute it, the I/O program performs other operations also, such as:

- Sensing the status of the device.
- Sending read or write command.
- Transferring the data.

the required operation, the processor has to issue an instruction that includes the address, which specifies I/O module and external device, and an I/O command which performs the required operation. The common I/O commands are as follows:

Control: To activate a peripheral device and tell it what to do.

- **Test:** To test the various status conditions associated with an I/O module and devices attached to it.

- **Read:** To cause the I/O module to obtain an item of data from the peripheral and place it into an internal register.

- **Write:** To cause the I/O module to accept a unit of data from the data bus and transmit it to the peripheral.

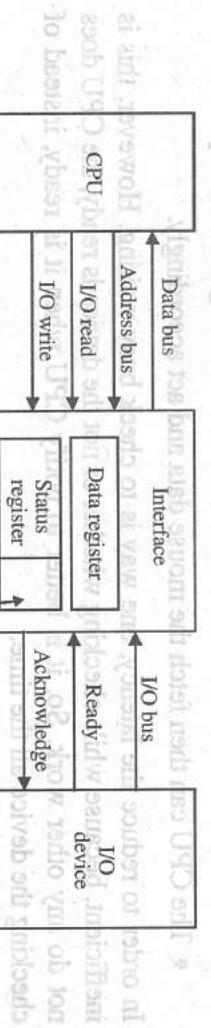


Fig. 2.60 Data Transfer for Programmed I/O

There are many I/O devices connected through I/O modules to the system, each having some unique address. The processor views I/O operations as memory operations and issues commands containing device address. Each I/O module checks the address lines to see if the command is meant for itself. The following are two ways in which addressing can be done:

- **Memory-Mapped I/O:** Here a single address space is used for storing both memory and I/O devices. The processor treats the status and data registers of I/O module as memory locations.
- **Isolated I/O:** Separate address spaces are used for both memory and I/O devices.

The major disadvantages of this technique are as follows:

- The I/O instructions are usually quite slow.
- In case of input devices, a lot of CPU time is wasted in checking the device as most of the time the data is not available (processor speed is much higher than speed with which the user gives the input, which is even more slow if data feeding is manual). This wastage of CPU time in checking a device, which prevents any data from being read, is called *spin waiting*.
- This technique is especially very problematic if device is to be initiated independently, e.g., mouse. We all know that the mouse movements are often quite random and one does not know when the next one will take place. With this technique, how can one find out when the mouse has moved?
- If some input devices, such as the keyboard cannot wait for the program, one may miss a word/byte of data.

NOTES

- In case of output devices, the processor cannot send bytes when the device is busy, i.e., when output device has not completed its previous task. Again, CPU busy cycles are wasted in waiting for the device to become ready.

NOTES

Checking each device at regular intervals is called *polling*. Question arises how frequently one should do polling. It depends on the nature of the devices. Some devices require constant attention, while others may require service only once in a while. Further, more devices mean more time to be wasted on each of them. To reduce this spinning time, one can use buffer as writing a program to the buffer is much faster. Figure 2.61 illustrates the flowchart of I/O data transfer. Using buffering in the programmed I/O, the CPU determines whether the mouse has moved by periodically polling the mouse command buffer:

- If the mouse has moved, the command buffer will flag this technique.
- The CPU can then fetch the mouse data and act accordingly.

In order to reduce the latency, one way is to check busy waiting. However, this is inefficient, because while checking whether or not the device is ready, the CPU does not do any other work. So, it is better to notify CPU when it is ready, instead of checking the device all the time.

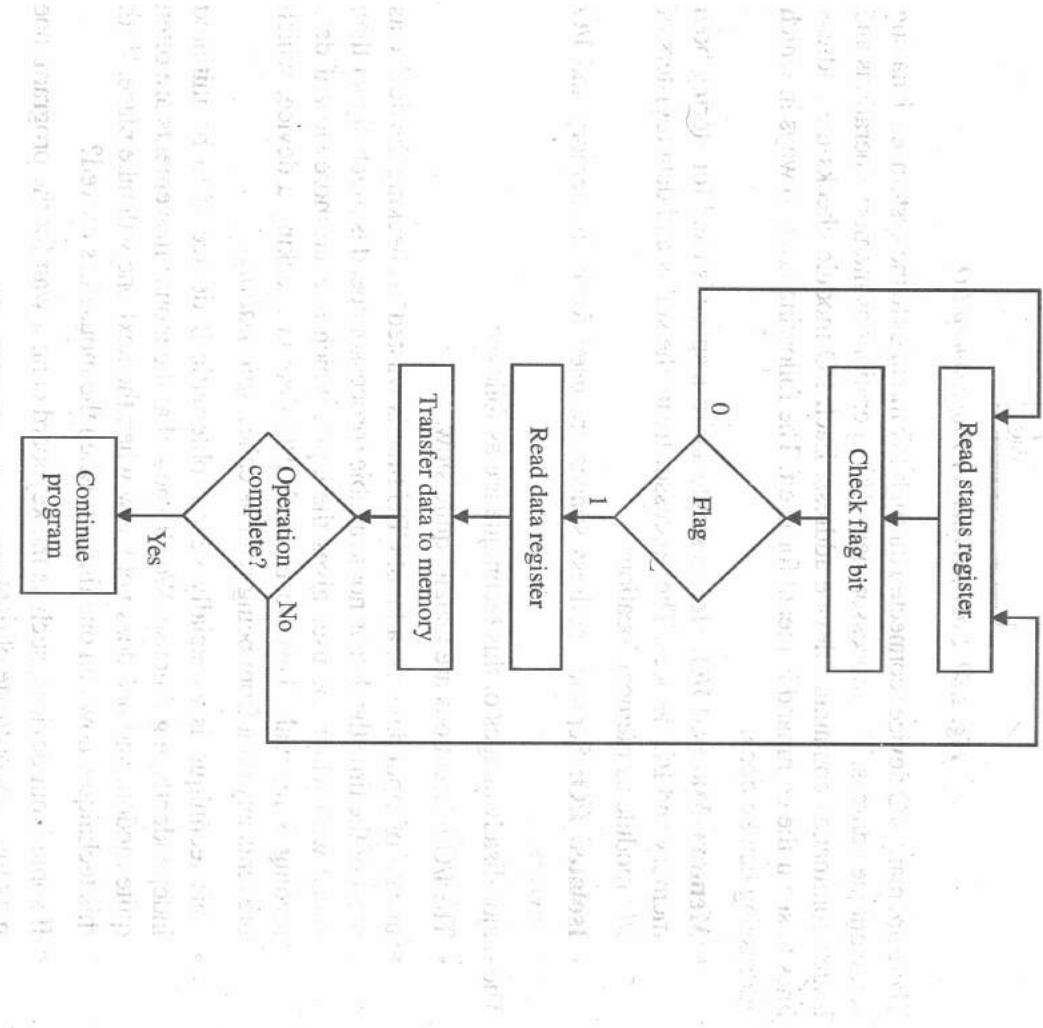


Fig. 2.61 Flowchart of I/O Data Transfer

An interrupt signal is a signal sent by an I/O interface to the CPU when it is ready to send information to the memory or receive information from the memory.

NOTES

The term, interrupt, is used for any exceptional event that causes the CPU to temporarily transfer the control from its current program to another program, an interrupt handler, which will service the interrupt. This program is known as *Interrupt Service Routine* (ISR). Interrupts are the primary means by which input/output devices obtain the services of the CPU. Various sources, internal and external to the CPU, can generate interrupts. I/O interrupts are external requests to the CPU to initiate or terminate an I/O operation, such as data transfer with a hard disk. Interrupts are also produced by hardware or software error-detection circuits that invoke error-handling routines within the operating system. An attempt by an instruction to divide by zero is an example of software-generated interrupts. A power supply failure can generate an interrupt that requests the interrupt handler to save critical data about the system's state.

An interrupt is initiated by a signal generated by an external devices or a signal generated internally by the CPU. When the CPU receives an interrupt signal from peripherals, it stops executing the current program, saves the contents or status of various registers in the stack and then executes a subroutine in order to perform the specific task requested by the interrupt. The interrupts generated by special instructions are called software interrupts; they are used to implement system services.

The interrupts are useful for efficient data transfer between the CPU and input-output peripherals. The interrupt I/O is a process of data transfer where an external device or a peripheral informs the CPU that it is ready for data transfer.

The data transfer between the CPU and peripheral devices can be implemented by either the polling technique or the interrupt method. In the polling technique, the processor has to periodically poll or check the status of the device and it can perform data transfer only when the device is ready. In the polling technique, the processor has to wait and hence, processor time is wasted. The processor has to suspend its work and check the status of the device in predefined intervals.

2.5.6 Direct Memory Access

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 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

NOTES

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.

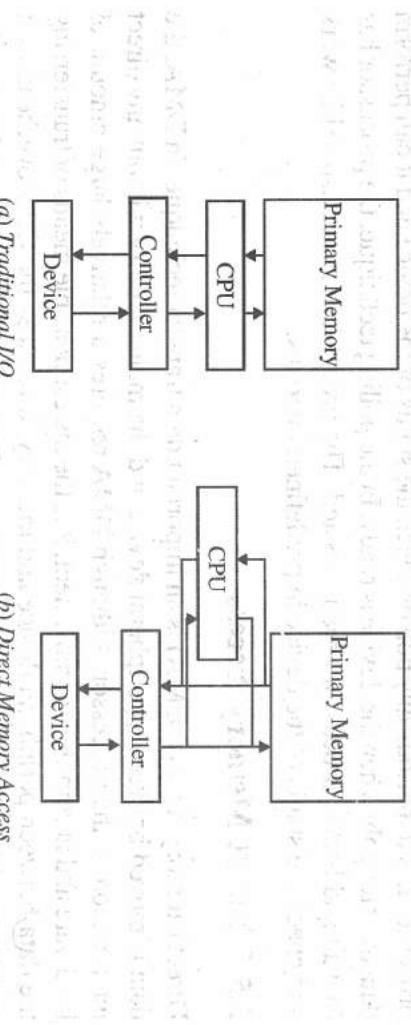
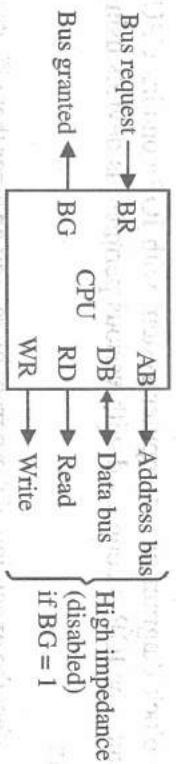


Fig. 2.62 Comparing the Functioning of Traditional I/O and DMA

Thus major part of CPU overhead is the time the CPU spends in reading operation as traditional case now overcame as shown in Figure 2.62. 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.

**NOTES**

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 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 (refer Figure 2.63). 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. There are three possible ways of organizing DMA module using detached bus or integrated bus or separate I/O bus.

- (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.

2.6 I/O CHANNELS AND PROCESSORS

The I/O Processor (IOP) enables direct communication between the CPU and the peripherals connected to the computer system. The communication takes place without any interference of the interface connected to a peripheral device. The IOP may support the DMA capability to establish communication between the CPU and peripheral device. The IOP works in the similar way as a CPU except it handles the I/O processing mainly. In addition, the IOP are designed to perform arithmetic operations, logical operations and branching.

Check Your Progress

12. What is input/output interface?
13. Why are printers used?
14. What are tracks?
15. What is recording density?
16. What is an interrupt signal?
17. Why is DMA technique useful?

The block diagram of a computer system with IOP contains CPU, IOP and memory unit. The IOP is connected with various peripherals devices using I/O bus.

Interrupts

NOTES

Interrupts affect the execution of a program by raising an error that affects the normal execution of the program. An interrupt handling routine is used to rectify this error, which is stored at a fixed memory location. There are two types of interrupts, vectored and non-vectored. In non-vectored interrupt, the branch address is assigned to fixed memory location whereas in vectored interrupt, the branch location is not fixed and is supplied by the source. The branch address stores the memory location to which the CPU control transfers when interrupt occurs in the program execution. This memory location contains the first address of the I/O service routine to rectify the interrupt in computer systems.

The device drivers of the input-output devices initiate the programs to get executed. The CPU initiates the execution of a program and also checks whether the interrupt occurs in the computer system or not. When the CPU receives any interrupt, the interrupt handler is used to service the interrupt. After the interrupt is serviced, the CPU resumes the execution of the program in which interrupts had occurred.

Channels

Channels are used to handle the I/O operation of computer system. The channel term is also referred for I/O processor that is used to manage the processing of I/O operations. The number of channels connected to the computer system is determined on the basis of application running on the system. Each channel handles the processing of one or more I/O devices connected to it. The various types of channels are as follows:

- Multiplexed channels, which are connected to peripherals of slow and medium data access speed and can process various I/O devices simultaneously.
- Selector channels, which are connected to peripherals of high access data speed and can process one peripheral at a time.

• Block multiplexer channels, which are connected to various peripherals of high access data speed and can process various peripheral simultaneously. Channels can communicate directly with CPU using dedicated control lines. The communication between channels and the CPU can take place indirectly using reserved storage space in memory. When the channel stores in the memory word format are used to enable operation of the channel. There are three types of word formats of the channel. These word formats are as follows:

1. I/O Instruction Word Format.
2. Channel Status Word Format.
3. Channel Command Word Format.

I/O Instruction Word Format

The I/O instruction word format is divided into three parts: operation code, channel address and device address. Operation code is used to specify the type of operation to be performed on peripherals. The operation code part of I/O instruction word format specifies one of the operation from eight possible operations. These operations includes

start I/O, start I/O fast release, test I/O, halt I/O, clear I/O, halt device, test channel and store channel.

NOTES Channel address part of the word format is used to specify the address of channel that is used to perform operation on the peripheral. The address of the peripheral is stored in the device address part of the I/O instruction word format. Figure 2.64 shows the I/O instruction word format of the channel.

Operation code	Channel address	Device Address
b11111111 e000	00000000	00000000

Fig. 2.64 I/O Instruction Format

Channel Status Word Format

Channel status word format is divided into four parts: key, address, status and count. The key part of this word format is used to provide the protection mechanism to the information of a user from unauthorised access. The address part of this word format is used to specify the address of the last command word used by the channel. The status part of the word format is used to specify the error conditions in the peripheral and channel, which terminates the data transfer. If the data is transferred successfully then the value of the status part becomes zero. The count part is used to specify the residual count when the data transfer gets terminated. Figure 2.65 shows the I/O instruction word format of the channel.

Key	Address	Status	Count
00000000	00000000	00000000	00000000

Fig. 2.65 Channel Status Word Format

Channel Command Word Format

Channel Command Word or CCW format is divided into four parts: command code, data address, flags and count. The data address part of the word format specifies the first address of the memory where data is stored. The count part is used to specify the total number of bytes involved in the transfer. The flags part of the format is used to specify the status of the channel. The command code part of the word format specifies the operation that needs to be performed by the channel. This part specifies one of the operations among six I/O operations. These I/O operations are:

- **Write Command:** It is used to transfer data from memory to I/O device.
- **Read Command:** It is used to transfer data from I/O device to memory.
- **Read in Backward Direction:** It reads data from I/O device in backward direction. For example, in a magnetic tape, if the head points to the third record currently and the CPU needs to access the 2nd record, then CCW or Channel Command Word format issues this command.
- **Control:** It is used to initialize an operation that does not constitute a data transfer operation.

- **Sense:** It is used to transfer the respective channel status word to memory address 64.

Transfer: It is used to branch the control of the CPU to some specified memory location in the memory.

Figure 2.66 shows the I/O instruction word format of the channel.

Command code	Data address	Flags	Count
address	address	req elco	

Fig. 2.66 Channel Command Word Format

Till now you 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 you can deploy one or more external processors and assign them the task of communicating directly with I/O devices without any 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 2.67, such a processor has one memory unit and 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).

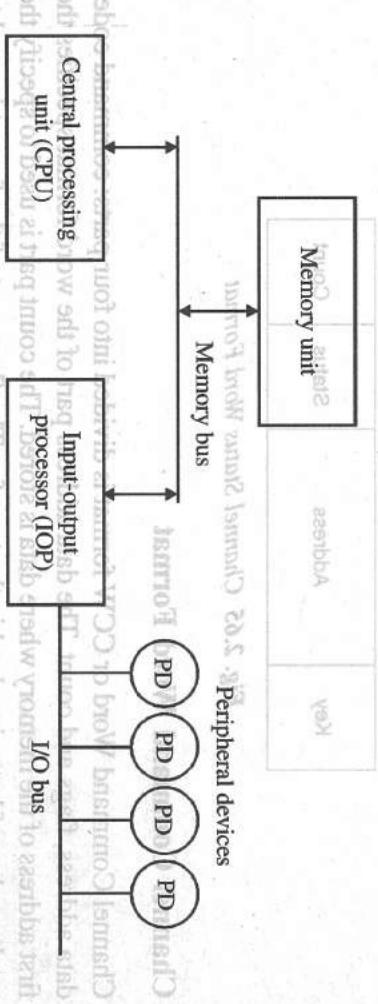
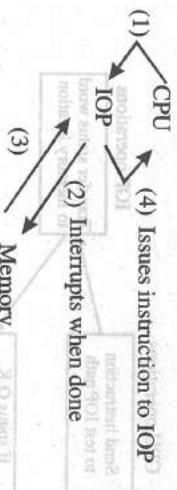


Fig. 2.67 Block Diagram of an IOP

Figure 2.67 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 the 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 2.68).



NOTES

Instructions that are used for reading from memory by an IOP are called commands, there are instructions words that are used as CPU instructions. The CPU informs IOP where the command is in memory and when it is to be executed (refer Figure 2.69).

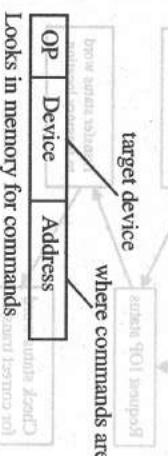


Fig. 2.69 CPU Command for Memory

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 has taken place and any other special request (refer Figure 2.70).

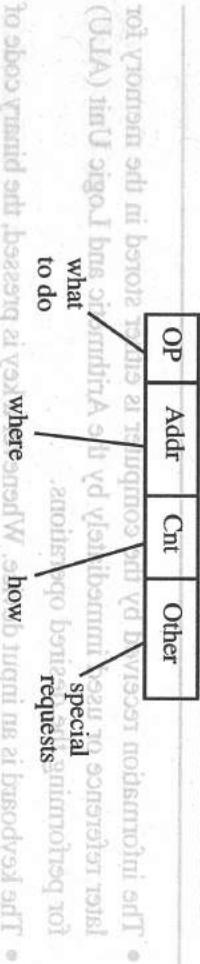


Fig. 2.70 IOP Instruction

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 the status bit is OK, the CPU sends 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. 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.

Figure 2.71 illustrates the communication between CPU and IOP.



NOTES

- Check Your Progress**
18. Why are channels used?
 19. Name the four parts of channel command word format.
 20. What is a data communication processor?

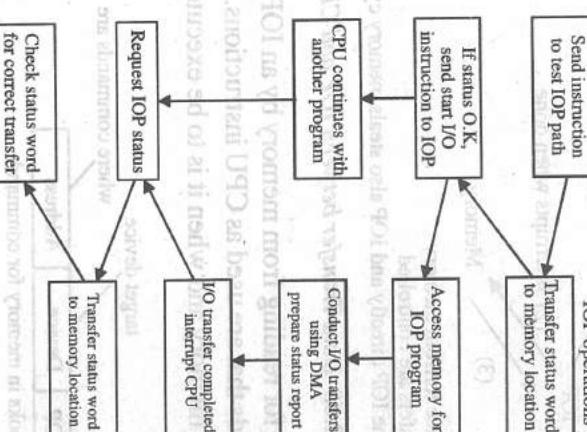


Fig. 2.71 CPU-IOP Communication Process

2.7 SUMMARY

- The information received by the computer is either stored in the memory for later reference or used immediately by the Arithmetic and Logic Unit (ALU) for performing the desired operations.
- The keyboard is an input device. Whenever a key is pressed, the binary code of the corresponding letter or digit is transferred to the memory unit or processor.
- The principal characteristic of a multiprocessor is its ability to share a set of main memory and some I/O devices. This sharing is possible through some physical connections between them which are called the interconnection structure.
- The Dynamic Random Access Memory or DRAM is the lowest cost, highest density random access memory available. Nowadays, computers use DRAM for main memory storage with the memory sizes ranging from 16 to 256 MB.
- 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.
- An associative memory, also called Content Addressable Memory or CAM, is nothing a very high speed memory that provides a parallel search capability. It is capable of searching the contents of all its locations at any instant of time.
- RAID 0+1 is a combination of striping and mirroring. This configuration provides optimal speed and reliability, but possesses the same cost problem as RAID1.

- 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.

NOTES

- 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 performance of cache memory is frequently measured in terms of a quantity called hit ratio. When the CPU refers to any word to the memory and finds the word in the cache memory, it is said to produce a hit. If the word referred to by the CPU is not found in the cache memory, it is in the main memory and counts as a miss.

- The input-output system of a computer processes the input information by converting them into computer readable binary form and then displaying the final result as output in user readable form.

- A control command is issued to activate the peripheral and to inform it what to do. The particular control command issued depends on the peripheral.

- A mouse is a pointing device. It can be moved on a smooth surface so that it points to the desired location on the screen. It can be optical or mechanical. A user can move the mouse, stop it at a desired location and with the help buttons can select choices.

- RAM is a volatile memory and thus a permanent storage media is required in a computer system. Auxiliary memory devices are used in a computer system for the permanent storage of information and hence, are the devices that provide backup storage. They are used for storing system programs, large data files, and other backup information.

- Programmed I/O is the simplest method of I/O data transfer and it executes a program that contains dedicated I/O instructions. When the processor encounters an I/O instruction, it issues a command for the appropriate I/O module that executes the given instruction.

- Channels are used to handle the I/O operation of computer system. The term channel is also referred for I/O processor that is used to manage the processing of I/O operations. The number of channels connected to the computer system is determined on the basis of application running on the system.

2.8 KEY TERMS

- **Channel width:** The number of bits that can communicate simultaneously by a interconnection bus connecting two processors
- **Bus:** A subsystem that transfers data between components inside a computer or between computers.
- **Interleaving:** An advanced technique used by high end chipsets to improve memory performance
- **Cache:** Small and fast memory placed between the CPU and main memory

- **Var'd bit:** 1-bit field indicates the status of data stored in the cache.
- **Peripherals:** Electromechanical and electromagnetic devices connected to a computer system
- **Magnetic disk:** A memory device that is covered with a magnetic coating in a disk platter.
- **I/O channel:** A high performance I/O architecture implemented in mainframe computers

2.9 ANSWERS TO ‘CHECK YOUR PROGRESS’

1. The task of the memory unit is to safely store programs as well as to manage input, output and intermediate data.

2. The simplest interconnection system for multiprocessors is a common global communication path connecting all of the functional units. This common path is called as time shared or common bus.

3. The common memory is also called shared memory and contains global information.

4. The speed of the bus is affected by its length as well as by the number of devices sharing it.

5. The I/O processor manages the data transfer between the auxiliary memory and the main memory.

6. Memory interleaving increases the bandwidth by allowing simultaneous access to more than one stack of memory.

7. Associative memory is used in database system for fast retrieval of information by feeding some data like getting the information about by customer by feeding the customer ID.

8. RAID (Redundant Array of Independent Disks) is an acronym for a disk array and consists of a number of hard disks and disk drives with a controller in a single box.

9. The hit ratio is number of hits divided by the total number of references a processor makes to cache memory (hits plus misses).

10. The amount of information which is replaced at one time in the cache is called the line size/block for the cache.

11. The cache memory is accessed by mapping the physical address with the tag stored in the cache.

12. A hardware unit that plays an important role between the CPU and peripheral devices to supervise the input and output transfer is known as input/output interface.

13. Printers are used for getting the output on paper.

14. Information is stored on the surface of a disk along concentric sets of rings known as tracks.

15. The number of bits per inch (bpi) is known as recording density.

KEY TERMS

8.5

16. An interrupt signal is a signal sent by an I/O interface to the CPU when it is ready to send information to the memory or receive information from the memory.
17. The DMA technique is useful for transferring the large amount of data (e.g., disk images, disk transfer, etc.) to memory.
18. Channels are used to handle the I/O operation of computer system.
19. Channel command word format is divided into four parts: command code, data address, flags and count.

NOTES

15. Illustrate the I/O system.

20. The processor that communicates with remote terminals like telephone or any other serial communication media in serial fashion is called a Data Communication Processor (DCP).

2.10 QUESTIONS AND EXERCISES**Short-Answer Questions**

1. What is a word length?
2. Why are conflict resolution methods used?
3. What do you mean by bus transfer?
4. Why does dynamic RAM employ a technique?
5. Write the function of RAID 5.
6. Where is cache placed?
7. Name the three types of mapping addresses.
8. Write one of the advantages of direct mapping.
9. What is n-way set associative mapping?
10. Why are interrupts useful?
11. Name the parts of channel status word format.
12. Write the function of control I/O operation.

Long-Answer Questions

1. Explain the various computer functions with the help of examples.
2. Discuss the function of single bus multiprocessor and multibus multiprocessor organizations with the help of illustrations and examples.
3. Discuss the bus system and its function for four registers.
4. Explain the memory hierarchy with the help of examples and illustrations.
5. Discuss the various types of RAID with the help of examples.
6. Explain cache memory organization with the help of illustrations and examples.
7. Discuss the structure of cache memory with the help of illustration.
8. Explain the elements of cache design with the help of examples.
9. Explain the process for improving cache memory.

10. Explain the functions of twin universal serial bus device, external hard disk drive and pen or thumb drive device.
11. Describe the process of data transfer for programmed I/O with the help of illustrations.
12. Illustrate the I/O instruction format and channels status word format with the help of examples.

NOTES

2.11 FURTHER READING

- Tanenbaum, Andrew S. 1999. *Structured Computer Organization*, 4th edition. New Jersey: Prentice-Hall Inc.
- Wilkinson. 1996. *Computer Architecture: Design and Performance*, 2nd edition. Hertfordshire: Prentice-Hall.
- Mano, M. Morris. 1993. *Computer System Architecture*, 3rd edition. New Jersey: Prentice-Hall Inc.
- Stalling, William. 2006. *Computer Organization and Architecture*, 7th edition. New Jersey: Prentice-Hall Inc.
- Hamacher, V.C., Z.G Vranesic and S.G. Zaky. 2002. *Computer Organization*, 5th edition. New York: McGraw-Hill International Edition.
- Conte, T. M. and C. E. Gimarc. 1995. *Fast Simulation of Computer Architecture*. Boston: Kluwer Academic Publishers.
- Gilmore, M. 1996. *Microprocessors: Principles and Applications*, 2nd edition. New York: McGraw-Hill.

UNIT 3 CENTRAL PROCESSING UNIT

Structure

3.0 Introduction

3.1 Unit Objectives

3.2 Instruction Set : Characteristics and Functions

3.3 Instruction Set : Addressing Modes and Formats - An Introduction

3.3.1 Addressing Modes

3.3.2 Instruction Formats

3.4 CPU Structure and Function

3.4.1 Processor Organization

3.4.2 Register Organization

3.4.3 Instruction Cycle

3.4.4 Instruction Pipelining

3.5 Reduced Instruction Set Computing or RISC

3.6 Instruction Level Parallelism and Superscalar Processors

3.6.1 Superscalar versus Superpipelined

3.6.2 Limitations

3.6.3 Instruction Issue Policy

3.6.4 Superscalar Execution

3.6.5 Superscalar Implementation

3.6.6 Instruction Level Parallelism and Machine Parallelism

3.7 Summary

3.8 Key Terms

3.9 Answers to 'Check Your Progress'

3.10 Questions and Exercises

3.11 Further Reading

3.1 UNIT OBJECTIVES

- Explain the basic components of a CPU.
- Explain the basic functions of a CPU.

3.0 INTRODUCTION

In the previous unit, you learnt about the computer system. In this unit, you will learn about central processing unit. The central processing unit is the portion of a computer system that carries out the instructions of a computer program to perform the basic arithmetical, logical and I/O operations of the system. Instruction set is the basic set of commands or instructions that a microprocessor understands. The operation of the processor is determined by the instructions it executes referred to as machine instructions or computer instructions. The instruction must specify the I/O module and device for the operation. If memory-mapped I/O is used then the address is represented either by main or virtual memory address. You will also learn about CPU structure and function. The CPU consists of three prime components; the register set that stores the traditional data while processing the programs and commands, ALU which performs the necessary microoperations for processing the programs and information amongst registers and directs the ALU on the instructions to follow. Input, storage, processing, output and control are considered as the prime functions of CPU.

NOTES

Input is the process of entering data and programs into the computer system. Output is the process of producing results from the data for getting useful information. Instruction cycle is the process by which a computer retrieves a program instruction from its memory, determines instruction required action and carries out those actions. RISC or Reduced Instruction Set Computer is a microprocessor that is designed to perform specific types of computer instructions and operates at a higher speed. Several instructions are executed in the same clock cycle by the processor. Such processors are capable of achieving an instruction execution throughput for more than one instruction per cycle and are known as superscalar processors. In superscalar processors, a pipeline hazard occurs, a condition in which a program ceases to work correctly due to implementation of the processor with a pipeline. Structural hazards occur when a specific hardware is used in more than one stage of the pipeline and leads to the deadlock of the required resources by two different instructions. Rename buffers establish the actual framework for renaming. Branch prediction increases the number of instructions available for the scheduler to issue and also instruction level parallelism. It allows to complete the essential tasks while waiting for the branch to resolve. The purpose of the branch predictor is to improve the flow in the instruction pipeline. Finally, you will learn about instruction level parallelism and machine parallelism. Machine parallelism is based on the concepts of superscalar and superpipelined machines. Superscalar machines can issue several instructions per cycle. Superpipelined machines can issue only one instruction per cycle but they have cycle times shorter than the latencies of their functional units. Machine parallelism interface allows us to specify details about the pipeline, functional units, cache and register set.

3.1 UNIT OBJECTIVES

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

- Understand instruction set, its functions and characteristics
- Explain addressing modes and formats
- Illustrate CPU structure and function
- Describe RISC and its microprocessors
- Explain instruction level parallelism and superscalar processors
- Discuss the instruction issue policy, register renaming and branch prediction
- Describe instruction level parallelism and machine parallelism

3.2 INSTRUCTION SET

An **instruction** is a command given to a computer to perform a specified operation on some given data. These instructions tell the Central Processing Unit or 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:

- Opcode
- Operand

Thus,

$$\text{Instruction} = \text{Opcode} + \text{Operand}$$

Opcode (operation code) is an instruction field that specifies the particular operation to be performed by the instruction. Each operation has its unique opcode and may take several microoperations to accomplish. MOV, ADD and SUB are examples of Intel 8086 opcodes.

Operand fields specify where to get the source and destination operands for the operation specified by the opcode. The source/destination of operands can be the memory or one of the general-purpose registers. The complete set of opcodes 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 increased cost of memory where the instructions are to be stored.

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 R0 to the content of register R2 and the result is to be stored in R2. The execution of this instruction is performed in the following steps:

- Step 1: Fetch and decode the instruction.
- Step 2: Fetch the operand.
- Step 3: Perform the operation (addition).
- Step 4: Store the result in R2.

NOTES

Instruction set is regular and compatible in which programs written for previous versions of machines need for basic operations. It is simple and easy to implement. The operation of the processor is determined by the instructions which are executed as machine instructions or computer instructions. The collection of different instructions that the processor can execute is referred to as the processor's instruction set. Each instruction must contain the information required by the processor for execution which can be referred to 'machine instruction characteristics'. These instructions are discussed below:

- **Operation Code:** This element specifies the operation to be performed, for example ADD, I/O, etc. The operation is specified by a binary code known as 'operation code'.
- **Source Operand Reference:** The operation may involve one or more source operands, i.e., operands that are inputs for the operation.
- **Result Operand Reference:** The operation can produce a result.
- **Next Instruction Reference:** This element instructs the processor where to fetch the next instruction after the execution of this instruction is complete.

Each instruction in the instruction set describes one particular CPU operation. Each instruction is represented in both assembly languages by the mnemonics and machine language (binary) by a word of 32 bits subdivided into several fields. Figure 3.1 illustrates the format of instruction representation in which opcode uses 4 bits, operand reference uses 6 bits and operand reference retains 6 bits because each instruction has a unique bit pattern in machine code. The significance of instruction set is that it is used by computer designer and programmer. From the computer design point of view, the instruction set provides the functional requirements of the CPU. For implementing the CPU design, one of the main tasks is to implement the instruction set for that CPU. From the user point of view, machine or assembly instructions are needed for low-level programming.

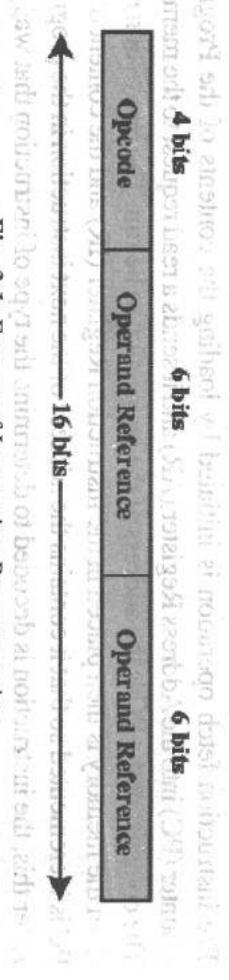


Fig. 3.1 Format of Instruction Representation

Instructions are represented as sequence of bits. An instruction is divided into a number of fields. Each of these fields corresponds to a constituent element of instruction. A layout of instruction is termed as instruction format.

3.2.2 Types of Operands

An operand is the part of a computer instruction. It specifies the process which instructs how data is to be operated or manipulated. Basically, a computer instruction describes an operation (add, subtract, etc.) and the operand or operands on which the operation

is to be performed. Operands include numbers (integer/floating point), characters (American Standard Code for Information Interchange or ASCII) and logical data (bits or flags). Following are the types of operand:

- **Addresses:** Addresses are considered as a form of data which is used in calculation of physical memory address of an operand. In most of the cases, the addresses provided in instruction are operand references and not the actual physical memory addresses.

• **Numbers:** All machines provide numeric data types. One of the prime features of numbers used in computers is that they are limited in magnitude and hence the underflow and overflow may occur during arithmetic operations on these numbers. Numbers include fixed point numbers or integers (signed or unsigned), floating-point numbers and decimal numbers. Many machine arithmetic instructions which perform operations are packed decimal digits.

• **Characters:** It is the most widely used character representation is ASCII. It has 7 bits for coding data pattern which implies 128 different characters. It is also used in conversion of 7 bits ASCII and a 4-bit packed decimal number. The last four digits of ASCII number are binary equivalent of digits 0-9. The other character code is Extended Binary Coded Decimal Interchange Code (EBCDIC). It is 8-bit code and is compatible with decimal number system in same way as ASCII code works.

• **Logical Data:** In general, a data word or any addressable unit, such as byte, half word, etc., are considered as a single unit of data. Logical data refers to bit oriented data. The advantages of bit oriented data are to store an array of Boolean or binary data items most efficiently and to be in a position to manipulate the bits of any data item. If we take each bit of a n-bit data as an item then it can be considered to be logical data and each of these n items can have a value 0 or 1.

Source and result operands can be one of the following areas:

- **Main or Virtual Memory:** As with next instruction references, the main or virtual memory address must be supplied.

• **Processor Register:** With rare exceptions, a processor contains one or more registers that may be referenced by machine instructions. If only one register exists, reference will be explicit. If more than one register exists then each register is assigned with a unique number and the instruction must contain the number of the desired register.

• **I/O Device:** The instruction must specify the Input/Output or I/O module and device for the operation. If memory-mapped I/O is used then the address is represented either by main or virtual memory address.

3.2.3 Types of Operations

A computer system consists of an interrelated set of components. The system is characterized in terms of structure in which components are interconnected and perform the operation of the individual components which are assembled with the system units. Data transfer operations, arithmetic operations, logical operations, transfer control operations, I/O operations and conversion operations are the types of operations which are discussed below. Table 3.1 summarizes the data transfer operations.

NOTES

Operation Name	Description
Move (transfer)	This operation is used to transfer word or block from source to destination.
Store	This operation is used to transfer word from processor to memory.
Load (fetch)	This operation is used to transfer word from memory to processor.
Exchange	This operation is used to swap contents of source and destination.
Clear (reset)	This operation is used to transfer word of 0s to destination.
Set	This operation is used to transfer word of 1s to destination.
Push	This operation is used to transfer word from source to top of stack.
Pop	This operation is used to transfer word from top of stack to destination.

Table 3.2 summarizes the arithmetic operations.

Operation Name	Description
AND	This operation is used to perform the specified logical operation (AND) bitwise.
OR	This operation is used to perform the specified logical operation (OR) bitwise.
NOT	This operation is used to perform the specified logical operation (NOT) bitwise.
Exclusive-OR	This operation is used to perform the specified logical operation (XOR) bitwise.
Tested	This operation is used to replace operand by its absolute value.
Compare	This operation is used to test specified conditions; set flags based on outcome.
Set Control Variables	This operation is used to set controls for protection purposes, interrupt handling, timer control, etc.
Shift	This operation is used to left (right) shift operand introducing constants at end.
Rotate	This operation is used to left (right) shift operand with wraparound end.
Jump (branch)	This operation is used to unconditional transfer; load program counter with specified address; jump to address which specifies target location.

Table 3.3 summarizes the transfer control operations.

Table 3.3 Transfer Control Operations

Operation Name	Description	NOTES
Jump to Subroutine	This operation is used to place current program control information in known location; jump to specified address.	
Return	This operation is used to replace contents of PC and other register from known location.	
Execute	This operation is used to fetch operand from specified location and execute as instruction.	
Skip	This operation is used to increment PC to skip next instruction.	
Halt	This operation is used to stop program execution.	
Wait (hold)	This operation is used to stop program execution and test specified condition.	
No Operation	No operation is performed but program execution is continued.	

Table 3.4 summarizes the Input/Output or I/O operations. *Table 3.4 Input/Output Operations*

Operation Name	Description
Input (read)	This operation is used to transfer data from specified I/O port or device to destination, for example main memory or processor register.
Output (write)	This operation is used to transfer data from specified source to I/O port or device.
Start I/O	This operation is used to transfer instructions to I/O processor to initiate I/O operation.
Test I/O	This operation is used to transfer status information from I/O system to specified destination.

Table 3.5 summarizes the conversion operations. *Table 3.5 Conversion Operations*

Operation Name	Description
Translate	This operation is used to translate values in a section of memory based on a table of correspondences.
Convert	This operation is used to convert the contents of a word from one form to another, for example packed decimal to binary.

3.3 INSTRUCTION SET: ADDRESSING MODES

TO MEMORY AND FORMATS—AN INTRODUCTION

Along with the reduction in the hardware prices and the increase in the number of computer instructions, the complexity of the computer system has also increased. New models can provide more customer-based applications. It is easier to add more

instruction to facilitate the translation from high level language into machine language program.

NOTES

The instruction set is an important aspect of any computer organization. Every instruction has primarily two components: opcodes and operands. You will learn how to get operands on which all the manipulations are to be performed. 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 in 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.

As it is 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 even for some systems, 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 compile-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., $\text{operand} = \text{EA}$. There are two ways by which the control unit determines the addressing mode used by an instruction:

- The opcode itself explicitly specifies the addressing mode used in the instruction.
- The use of a separate mode field in the instruction indicates the addressing mode used.

Every instruction is represented by a sequence of bits and contains the information required by the CPU for execution. Depending on the format of instruction, each instruction is divided into fields, with each field corresponding to some particular interpretation. A general instruction format is given in Figure 3.2.



Fig. 3.2 Instruction Format

- **Opcode-Field:** It specifies the operation to be performed. The operation is specified by a binary code, known as the operation code or opcode.
- **Address-Field:** If provides operands on which operation is to be performed or provides the addresses of CPU register or main memory addresses which store the operands. These operands can be classified as follows:
- **Local Source Operand Reference:** The operation may involve one or more source operands; that is, operands that are the inputs for the operation.

Result Operand Reference: The operation may produce a result, i.e.,
Execution of the operand stores output.

Next Instruction Reference: This tells the CPU from where to fetch the
code to the next instruction after the execution of the current instruction is complete.

All the temporary data can be stored either in the main memory or register or can
directly be sent to the input-output device. Thus, the address field can be:

- **Main or Virtual Memory:** It contains the memory address of the main memory.
machine instructions. If only one register exists, reference to it may be implicit.
If more than one register exists, then each register is assigned a unique number,
and the instruction must contain the number of the desired register.
- **CPU Register:** CPU contains one or more registers that may be referenced by
instructions. If only one register exists, reference to it may be implicit.
If more than one register exists, then each register is assigned a unique number,
and the instruction must contain the number of the desired register.

Input/Output Device: The instruction must specify the I/O module or device
for the operation. The memory-mapped I/O can be used for storing or retrieving
instructions from another memory address.

Since there are variable sources that are used for storing and retrieving data, the
question arises as to what should be the maximum number of addresses one might
need in an instruction.

3.3.1 Addressing Modes

The technique for specifying the address of operands is known as **addressing mode**.
The address of an operand is known as **effective address**.

Each instruction needs data on which it has to perform the specified operation.
The operand (data) may be in an accumulator, a general-purpose register or at some
specified memory location. Thus, there are various ways of specifying the address of
data, known as **addressing modes**.

The different addressing modes are as follows:

- Implied addressing.
- Immediate addressing.
- Direct (absolute) addressing.
- Indirect addressing.
- Register addressing.
- Register indirect addressing.
- Relative addressing.
- Index addressing.
- Base register addressing.

Instruction Cycle

Instruction cycle performs the following operations:

- Fetch the instruction from the memory.
- Decode the instruction.
- Execute the instruction.

NOTES

NOTES

A program counter keeps trail of the instructions in the program stored in the memory. The program counter stores the address of the command to be run next and is increased each time an instruction is fetched from memory. Decoding determines the operation to be done, the addressing mode of the instruction and the location of the operands. The computer then executes the instruction and then returns to fetch the next instruction in the sequence.

There are two modes that need no address field at all. These are, ‘implied’ and ‘immediate’. Zero-address instructions in a stack-organized computer are implied mode instructions since the operands are implied to be on the top of the stack.

Implied Addressing Mode

The implied addressing mode is also known as implicit or inherent addressing mode. The operands are evidently specified in the definition of the instruction itself. ‘Complement Accumulator’ is an implied mode instruction because the operand in the accumulator register is implied in the definition of the instruction.

Following examples show the implied addressing mode:

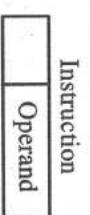
CMA: Take complement of the content of AC.

RLC: Rotate the contents of the accumulator.

All reference instructions that use an accumulator are implied mode instructions.

Immediate Addressing Mode

In this mode, the operand is indicated in the instruction itself, i.e., in the immediate addressing mode, the instruction has an operand field rather than an address field. The operand field contains the actual operand. This mode is useful for initializing registers to a constant value.



Following examples show the immediate addressing mode:

MVI 06 Move 06 to the accumulator.

ADD 05 Add 05 to the content of the AC.

Merits: The operand is available in the instruction as soon as the instruction fetch is over. Hence, the instruction executes quickly.

Demerits: The length of the operand field in instruction limits the value of the operand.

Register Addressing Mode

In the register-addressing method, the operands are in registers that exist within the CPU, i.e., the contents of the register are the operand itself. The instruction contains the register number that has the operand. This addressing mode is very useful in a long program for storing the intermediate results in registers rather than in memory. Figure 3.3 illustrates the register addressing mode.

Opcode	Register Number (R = Operand)
--------	-------------------------------

MOV R1, R2 R1 \leftarrow R2 Transfer the contents of register R2 to that of register R1

ADD R1 AC \leftarrow AC + R1 Add the contents of register R1 to the

accumulator

LD R1 AC \leftarrow R1 Load the content of register to the accumulator

Merits: Operands are fetched very fast without using memory.

Demerits: Since the number of registers is limited, the utilization of these registers must be done very carefully.

Register Indirect Addressing Mode

The command identifies a register in the CPU whose contents offer the address of the memory location where the operand is stored, i.e., the selected register comprises the address of the operand rather than the operand itself. In this mode, the register acts as the memory address register. Figure 3.4 illustrates register indirect addressing mode.

LD (R1) AC \rightarrow M[R1]

Opcode	Register Number (R = Operand Address)
LD (R1) AC \rightarrow M[R1]	R1



Fig. 3.4 Register Indirect Addressing Mode

Merits: Since the register number is specified with bits, the length of the instruction must be decided very carefully.

NOTES

Following examples show the register addressing mode:

MOV R1, R2 R1 \leftarrow R2 Transfer the contents of register R2 to that of register

R1

ADD R1

AC \leftarrow

AC + R1 Add the contents of register R1 to the

accumulator

LD R1

AC \leftarrow R1 Load the content of register to the accumulator

Merits: Operands are fetched very fast without using memory.

Demerits: Since the number of registers is limited, the utilization of these registers must be done very carefully.

Register Indirect Addressing Mode

The command identifies a register in the CPU whose contents offer the address of the memory location where the operand is stored, i.e., the selected register comprises the address of the operand rather than the operand itself. In this mode, the register acts as the memory address register. Figure 3.4 illustrates register indirect addressing mode.

LD (R1) AC \rightarrow M[R1]

Opcode	Register Number (R = Operand Address)
LD (R1) AC \rightarrow M[R1]	R1



Fig. 3.4 Register Indirect Addressing Mode

Merits: Since the register number is specified with bits, the length of the instruction must be decided very carefully.

Direct Addressing Mode

The direct addressing mode is also known as absolute addressing mode (refer Figure 3.5). In this mode, the address of data, i.e. the operand is specified in the instruction itself. In other words, the operand resides in the memory and its address is given directly by the address field of the instruction.



Following example shows the direct addressing mode:

LD ADR AC $\leftarrow M[ADR]$

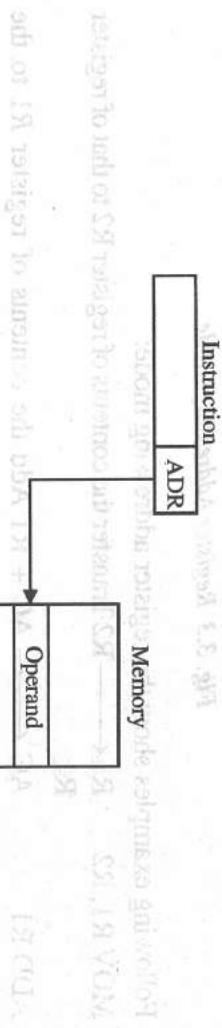


Fig. 3.5 Direct Addressing Mode

Merits: The instruction itself contains the operand address and hence, there is no need to find the effective address of the operand. Therefore, the instruction executes fast.

Demerits: The number of bits for specifying operand address is limited.

Indirect Addressing Mode

In this mode, the address field of the instruction gives the address where the operand is stored in the memory. Figure 3.6 illustrates the indirect addressing mode, where the address field of the instruction refers to the address of a word in memory which in turn contains the full length address of the operand.



Following example shows the indirect addressing mode:

LD @ ADR

AC $\leftarrow M[M[ADR]]$

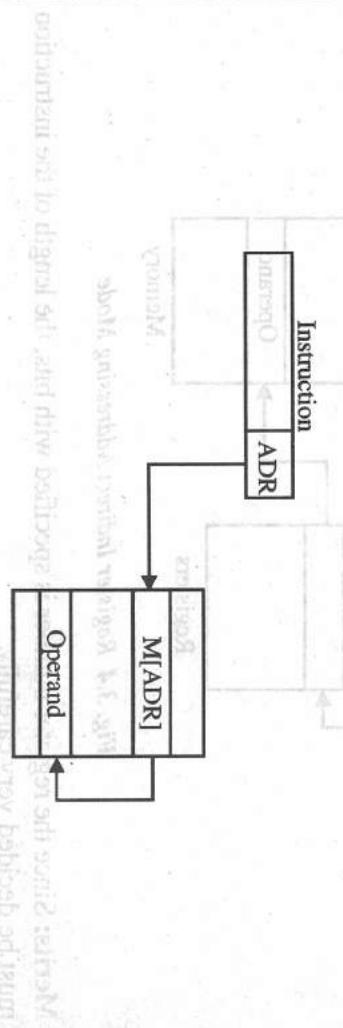


Fig. 3.6 Indirect Addressing Mode

Merits: The merit of this mode is that for the word length of N, an address space of $2N$ can be addressed.

Demerits: The demerit is that instruction execution requires two memory reference to fetch the operand. Multilevel or cascaded indirect addressing can also be used.

Displacement Addressing Mode

These addressing modes require that the address field of the instruction be added to the content of a specific register in the CPU to get the effective address.

Thus, $\text{Effective address} = \text{Address part of instruction} + \text{Content of the CPU register}$

The CPU register used in computing the effective address may be a program counter, an index register or a base register. Displacement addressing modes are of the following three types:

- **Relative Addressing Mode:** In the relative addressing mode for calculating the effective address, the contents of the program counter and the address part of the instruction are added.

For example, let the program counter contains 750 and the address part of the instruction contains the number 35. The instruction at the memory location 750 is read from memory during the fetch phase and the program counter is incremented by one to 751. The effective address computation for the relative address mode is $751 + 35 = 786$.

- **Index Addressing Mode:** In the index addressing mode to calculate the effective address, the contents of the index register and the address part of the instruction are added.

- **Base Register Addressing Mode:** In the base register addressing mode for calculating the effective address, the content of the base register and the address part of the instruction are added.

3.3.2 Instruction Formats

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:

$$\text{ea} = r[x] + sxt(\text{disp})$$

Here 'ea' is a variable which contains $r[x]$. It refers to the program counter which is indexed. The $r[0]$ 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 summarized in Table 3.6.

Table 3.6 Opcode and Keywords

Opcode	Keywords
add	add, addw, addl, addq, addsd, addss, addub, addus, addwod, addqd, addusd, addsdw, addsdq, addusdw, addusdq
sub	sub, subw, subl, subq, subsd, subss, subub, subus, subwod, subqd, subusd, subsdw, subsdq, subusdw, subusdq
mul	mul, mull, mullw, mulsd, mulss, mulub, mulus, mulwod, mulqd, mulusd, mulsdw, mulsdq, mulusdw, mulusdq

NOTES

The first row shows the opcode, 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 the difference between the current location and addressed label.

NOTES

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. Memory reference instructions are used in multi-threaded 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.

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,

74	15	12	11	8	0	15	12	11	1	1	1	1	dst	0	x
----	----	----	----	---	---	----	----	----	---	---	---	---	-----	---	---

is the structure of short word 17 stored at address 74. Here dst refers to memory space where the value is stored.

Figure 3.7 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.

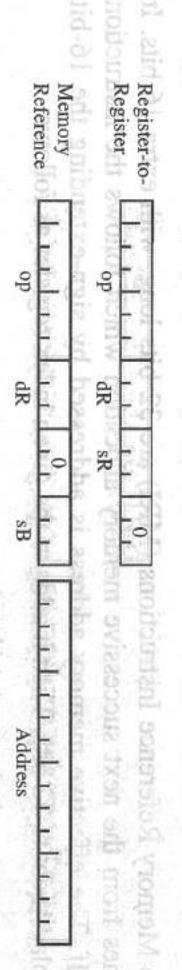


Fig. 3.7 Memory Reference Format

Figure 3.8 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 opcode are 00, 01 or 10. Instructions start with 11 is used for other instructions. The opcode has two parts in which the first part indicates the type of number and the second part shows the operation performed according to instruction as summarized in Table 3.7 and 3.8, respectively.

Table 3.7 1st Part of Opcode

Binary Representation	Type of Number	Bit Representation
000	Byte	8-bit integer
100	Halfword	16-bit integer
010	Integer	32-bit integer
011	Long	64-bit integer
1000	Medium	48-bit floating point
1001	Floating	32-bit floating point
1010	Double	64-bit floating point
1011	Quad	128-bit floating point

Table 3.8 1st Part of Opcode

Binary Representation	Type of Number
0000	Swap
0001	Compare
0010	Load
0011	Store
0100	Add
0101	Subtract
0110	Multiply
0111	Divide
1000	Insert
1001	Unsigned Compare
1010	Unsigned Load
1011	XOR
1100	AND
1101	OR
1110	Multiply Extensively
1111	Divide Extensively

Memory reference instruction, a type of instruction, requires two machine cycles, one to fetch the instruction and the other fetch the data at an address, and to execute the instruction.

Memory Reference Format
 Memory Reference Op DR op dR sX ssX SB Address
 Memory-to-Memory Op DR op dR sX ssX SB Address

Memory-to-Register Op DR op dX dB sX sB Address
 Register-to-Memory Op DR op dX dB sX sB Address
 Register-to-Register Op DR op dS op3 sR Address
 Scratchpad-to-Scratchpad Op DR op dS op3 sR Address
 Scratchpad-to-Memory Op DR op dS op3 sR Address
 Aux Register Memory Op DR op dBR sX ssX SB Address
 Reference acquire Memory Op DR op dBR sX ssX SB Address
 Reference acquire Memory Op DR op dBR sX ssX SB Address

Fig. 3.9 Memory Reference Instructions from Register-to-Register Format**NOTES**

NOTES

In Figure 3.9, ‘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 used for thirty-two based register. XOR, AND and OR perform the basic logical operations as summarized in Table 3.9.

Table 3.9 XOR, AND and OR Logic Operations

a	b	a XOR b	a AND b	a OR b
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	0	1	1

3.4 CPU STRUCTURE AND FUNCTION

Execution of programs is the main function of the computer. The programs or the set of instructions are stored in the computer’s main memory and are executed by the CPU. The CPU processes the set of instructions along with any calculations and comparisons to complete the task. Additionally, the Central Processing Unit or CPU controls and activates various other functions of the computer system. It also activates the peripherals to perform input and output functions.

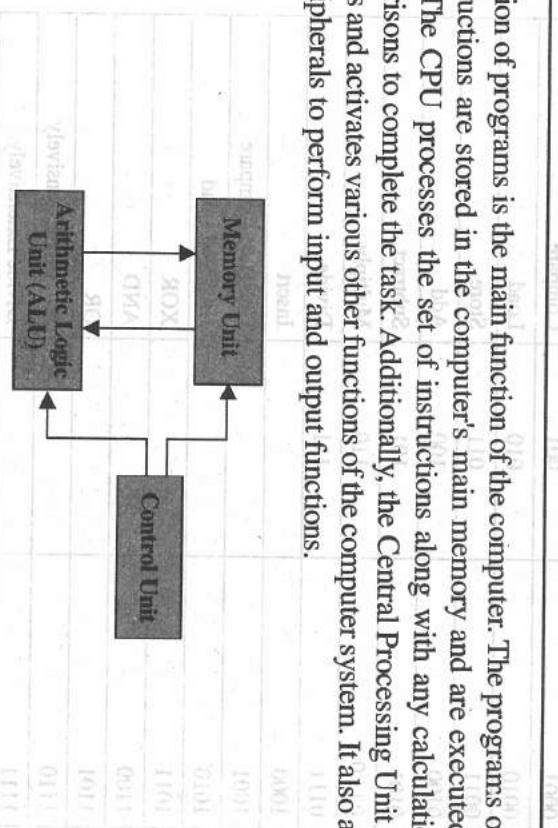


Fig. 3.10 Major Components of CPU

The CPU consists of three major components as shown in Figure 3.10—the register set (associated with the main memory) that stores the transitional data while processing the programs and commands, ALU which performs the necessary microoperations for processing the programs and commands, and the control unit that controls the transmitting of information amongst the registers and directs the ALU on the instructions to follow.

Control Unit

The control unit not only plays a major role in transmitting data from a device to the CPU and vice versa but also plays a significant role in the functioning of the CPU. It actually does not process the data but manages and coordinates the entire computer system including the input and the output devices. It retrieves and interprets the commands of the programs stored in the main memory and sends signals to other units of the system for execution. It does this through some special purpose registers and a decoder. The special purpose register called the instruction register holds the

Check Your Progress

- What is an instruction?
- What is instruction cycle?
- What is the role of address-field in instruction set?
- What is effective address?
- Why ‘complement accumulator’ is known as implied mode instruction?
- Why memory reference instructions are arranged as per the protocols of memory reference format?

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.

Arithmetic and Logic Unit

The Arithmetic and Logic Unit, or ALU is responsible for arithmetic and logic operations. This means that when the control unit encounters an instruction that involves an arithmetic operation (add, subtract, multiply, divide) or a logic operation (equal to, less than, greater than), it passes control to the ALU, which has the necessary circuitry to carry out these arithmetic and logic operations.

As an example, a comparison of two numbers (a logical operation) may require the control unit to load the two numbers in the requisite registers and then pass on the execution of the 'compare' function to the ALU.

Figure 3.11 represents the basic structure of a CPU.

Data Processing Unit

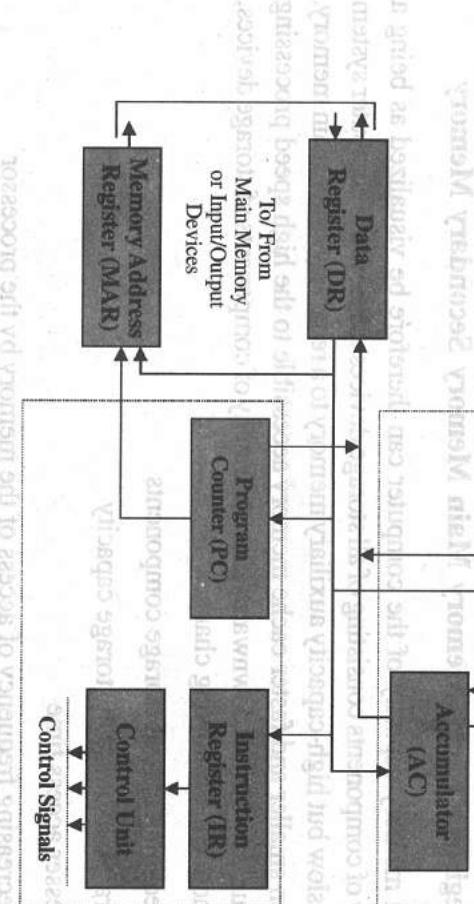


Fig. 3.11 Basic Structure of a CPU

Memory

Memory is used for storage and retrieval of instructions and data in a computer system. The CPU contains several registers for storing data and instructions. But, these can store only a few bytes. If all the instructions and data being executed by the CPU were to reside in secondary storage (like magnetic tapes and disks), and loaded into the registers of the CPU as the program execution proceeded, it would lead to the CPU being idle for most of the time, since the speed at which the CPU processes data is much higher than the speed at which data can be transferred from disks to registers. Every computer thus requires storage space where instructions and data of a program can reside temporarily when the program is being executed. This temporary storage

NOTES

area is built into the computer hardware and is known as the primary storage or main memory. Devices that provide backup storage (like magnetic tapes and disks) are called secondary storage or auxiliary memory. A memory system is mainly classified into the following categories:

NOTES

- **Internal Processor Memory** is a small set of high speed registers placed inside a processor and are used for storing temporary data while processing.
- **Primary Storage Memory** is the main memory of the computer which communicates directly with the processor. This memory is large in size and fast, but not as fast as the internal memory of the processor. It is actually a couple of integrated chips mounted on a printed circuit board which are plugged directly on the motherland. Random Access Memory (RAM) is an example of the primary storage memory.
- **Secondary Storage Memory** stores all the system software and application programs and are basically used for data backups. It is much larger in size and slower than the primary storage memory. Hard disk drives, floppy disk drives and flash drives are a few examples of secondary storage memory.
- **Cache Memory** is another category of memory which is being used by modern computer systems. It temporarily stores and supplies the data and instructions from the main memory to the internal memory (registers) to speed up the process.

CPU Registers Cache Memory Main Memory Secondary Memory

The total memory capacity of the computer can therefore be visualized as being a hierarchy of components consisting of all storage devices employed in computer system from the slow but high-capacity auxiliary memory to a relatively faster main memory, to an even smaller and faster cache memory accessible to the high speed processing logic. Thus, as we move downwards in the hierarchy of components of storage devices, we will notice the following changes:

- Declining cost of storage components
- Drastic increase in storage capacity
- Lesser access time
- Decreasing frequency of access of the memory by the processor

Memory Capacity: Capacity, in computer system, is defined in terms of the number of bytes that it can store in its main memory. This is usually stated in terms of Kilobytes (KB) which is 1024 bytes or Megabytes (MB) which is equal to 1024 KB (10,48,576 bytes). The rapidly increasing memory capacity of computer systems has resulted in defining the capacity in terms of Gigabytes (GB) which is 1024 MB (1,07,37,41,824 bytes). Thus a computer system having a memory of 256 MB is capable of storing $(256 \times 1024 \times 1024)$ 26, 84, 35, 456 bytes or characters.

The CPU processes data in binary form. An instruction is a basic operation performed by the CPU. Following are the prime functions of CPU:

- **Input:** Input is the process of entering data and programs in to the computer system. An input unit takes data from users to the computer in an organized manner for processing.

- Storage:** The process of saving data and instructions permanently is known as storage. Data has to be fed into the system before the actual processing starts. It is because the processing speed of CPU is so fast that the data has to be provided to CPU with the same speed. Therefore, the data is first stored in the storage unit for faster access and processing. This storage unit or the primary storage of the computer system is designed to do the defined functions. It provides space for storing data and instructions.

NOTES

- Processing:** The task of performing operations like arithmetic and logical operations is called processing. The CPU takes data and instructions from the storage unit and makes all sorts of calculations based on the instructions given and the type of data provided. It is then sent back to the storage unit.
- Output:** Output is the process of producing results from the data for getting useful information. Output can be displayed on the screen. The hard copy printout is also considered as output.
- Control:** Controlling of all operations like input, processing and output are performed by control unit. The control unit performs step-by-step processing of all operations inside the computer.

3.4.1 Processor Organization

Processor is an integrated circuit that contains the entire CPU on a single chip. It can be a single Large Scale Integration/Very Large Scale Integration (LSI/VLSI) chip in CPU. Memory distributed physically throughout the machine. Each processor has private memory. Data moves between processors, such as microprocessor 8086, microprocessor 8088, microprocessor 80286 and microprocessor 80386 but they do not perform complex numerical calculations and graphics designing. Microprocessor 8087 and microprocessor 82786 were launched for arithmetic applications and graphics designing, respectively. These microprocessors launched with co-processors and are arranged and operated in parallel with CPU. Using these co-processors, CPU first sends data and instructions to processors then processors generate results from it. Processors which are used for graphics applications known as graphics processors. Intel developed 740-3D graphics chip for enhancing the capability of 'graphics processors' to generate pixels and then process them efficiently. The processing speed of microprocessor is measured by Million Instructions per Second or MIPS and Floating-Point Instructions per Second or MFLOPS. In these days, System Performance Evaluation Committee or SPEC is being used to check the processor's performance. In CPU organization, a number of microprocessors are used to control input and output devices of a large computer. They are also used to control the performance of keyboard and Cathode Ray Tube (CRT) display unit. They are also used to control the operation of a printer. VLSI technology uses single-chip microcomputers containing high quality of microprocessors that is really a complete processor, such as CPU memory RAM, Read Only Memory or ROM, Erasable Programmable Read-Only Memory or EPROM and input and output lines. Table 3.10 summarizes the list of Intel processors and their capacity.

Table 3.10 summarizes the list of Intel processors and their capacity.

Central Processing Unit

Table 3.10 Intel Microprocessors

Microprocessor Name	Year	Pins	Capacity/Bit Length
4004	1971	16	1KB/4 bit
8085	1976	40	64KB/8 bit
8086	1978	40	1MB/16 bit
8088	1980	40	1MB/16 bit
80286	1982	68	16MB (Real) 4 GB (Virtual)/16 bit
80386	1985	100	4 GB (R) 64 TB (Virtual)/32 bit
80486	1989	168	4 GB (Real) 64 TB (Virtual)/32 bit
Pentium	1993	237 Pin Grid Array or PGA	4 GB (Real)/32 bit
Pentium Pro	1995	387 Pin PGA	64GB(Real)/32 bit
Pentium II	--	--	--
Celeron	1998	--	--
Pentium III	1999	--	64GB (Real)/32 bit
Pentium IV	2000	--	64GB (Real)-64TB (Virtual)/64 bit
Turion	2005	--	1TB-256TB/64 bit

In processor, some CPU processes data faster than others. A computer contains a system clock that emits pulses to establish the timing of all systems operations. The system clock operates at a speed quite different from a clock that keeps track of the time of the day. The system clock determines the speed at which the computer can execute an instruction and therefore limits the number of instructions the computer can complete within a specific amount of time. The time to complete an instruction execution cycle is measured in Megahertz (MHz) or millions of cycles per second. Figure 3.12 illustrates the Advanced Micro Devices or AMD Phenom II Socket AM3, Intel Core i7 LGA 1366 and Intel Core i5 LGA 1156 Processors.

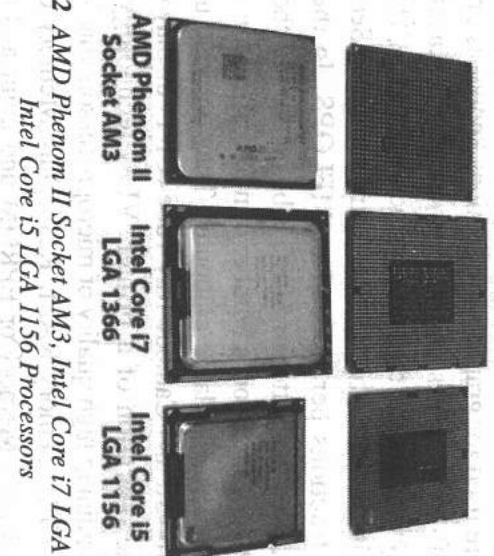


Fig. 3.12 AMD Phenom II Socket AM3, Intel Core i7 LGA 1366 and Intel Core i5 LGA 1156 Processors

Today, microprocessor speeds exceed up to 300 MHz. Some newer computer programs will not run on older processors, and some newer processors are not

compatible with older software. The faster the processor in a computer, the more quickly the computer will perform operations. Therefore, processor organization refers to:

- A microprocessor incorporates the functions of a computer's Central Processing Unit (CPU) on a single Integrated Circuit (IC) or at most a few integrated circuits. It is a multipurpose, programmable device that accepts digital data as input, processes it according to instructions stored in its memory and provides results as output.

NOTES

- A Graphics Processing Unit or GPU is a specialized circuit designed to rapidly manipulate and alter memory in such a way so as to accelerate the building of 3D images in a frame buffer intended for output to a display. GPUs are used in embedded systems, mobile phones, personal computers, workstations, and game consoles.
- A Physics Processing Unit (PPU) is a dedicated microprocessor which is designed to handle the calculations of physics. It is used in the physics engine of video games. Examples of calculations involving a PPU are collision detection, fluid dynamics, clothing simulation, finite element analysis, etc.
- A Digital Signal Processor (DSP) is a specialized microprocessor with an architecture optimized for the fast operational needs of digital signal processing.
- A network processor is an integrated circuit which is used for networking application domain.
- A Front End Processor (FEP) is a small sized computer which interfaces to the host computer a number of networks, and a number of peripheral devices, such as terminals, disk units, printers and tape units. Data is transferred between the host computer and the front end processor using a high speed parallel interface. Examples of front end processor are International Business Machines or IBM 3705 Communications Controller and the Burroughs Data Communications Processor.
- A coprocessor is a computer processor used to supplement the functions of the primary processor (CPU). Operations performed by the coprocessor may be floating point arithmetic, graphics, signal processing, string processing or encryption.
- Data processing refers to computer data processing, computer processes that convert data into information or knowledge.
- A Word processor is a computer application used for the production (including composition, editing, formatting, and printing) of any sort of printable material, such as Microsoft Word, WordPad, etc.
- Audio signal processing is the intentional alteration of auditory signals or sound.

Physical Memory Organization in Processor

Physical memory can be described as the total amount of memory installed in the computer. For example, if the computer has two 64MB memory modules installed, it

NOTES

has a total of 128MB of physical memory. It is also referred to as the physical storage or the real storage. The model of memory organization is determined by systems software designers. A 'flat' address space is consisted of a single array of up to 4GB. A 'segmented' address space is consisted of a collection of arrays up to 16,383 linear addresses. It spaces up to 4GB. Both models can provide memory protection for the computer system and memory sizes of computer are described as specific number of words. Processor memory sizes are given in K increments or roughly 1,000 word blocks. The exact size of 1K block is 1024 wordblocks. Physical memory management can be performed better with the help of following factors:

- **Access Time:** It is the time required to read from or write data to a particular memory address in the memory. It is the interval at which a request for data is initiated until the data is available for use. It can range from a few nanoseconds (ns) to microseconds (μ s).

• **Destructive Readout:** When data is read from memory, then in this process the stored data is extracted from memory and is erased from the source. Because the data is lost, the process is referred to as destructive readout. If it is desired to restore the same data at the same storage location, the word must be rewritten after reading. Read/write memory, such as core memory is an example of destructive readout.

• **Non-destructive Readout:** If the data in a memory is not destroyed in the reading process, the system has non-destructive readout. This means the data can be read over and over again without being rewritten. A flip-flop is an example of non-destructive readout. Sensing the output voltage reads from a given side of the flip-flop generally does not change the state of the flip-flop and the stored data is retained.

• **Volatile Memories:** These are memories that lose their contents when the power is turned off. A semiconductor memory is an example.

• **Non-volatile Memories:** These are memories that do not lose their contents when power is removed. Core memory is an example.

Processor organization helps in optimizing the performance based products. For example, software engineers need to know the processing ability of processors. They may need to optimize software for good performance at less expense. This mechanism can require quite detailed analysis of the computer organization. For example, in a multimedia decoder, the designers might need to arrange for most data to be processed in the fastest data path and the various components are assumed to be in place and task is to investigate the organizational structure to verify the computer parts operates. Processor organization also helps plan the selection of a processor for a particular project. Multimedia projects may need very rapid data access while supervisory software may need fast interrupts. Sometimes, certain tasks need additional components as well. For example, a computer capable of virtualization needs virtual memory hardware so that the memory of different simulated computers can be kept separated. The computer organization and features also affect the power consumption and the cost of the processor. 'Power consumption' is used as prime 'design factor' that is

considered as the design time of modern computers. Power efficiency can often be traded for performance or cost benefits. The measurement in this case is MIPS/W or Millions of Instructions per Watt. With the increasing power density of modern circuits as the number of transistors per chip scales, power efficiency has increased. Recent processor designs, such as the Intel Core 2 put more emphasis on increasing power efficiency.

NOTES

Memory management looks very similar to memory management in the 8086 CPU.

3.4.2 Register Organization

A bus organization for seven CPU registers is shown in Figure 3.13. 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 microoperations to be performed. The result of the microoperation 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.

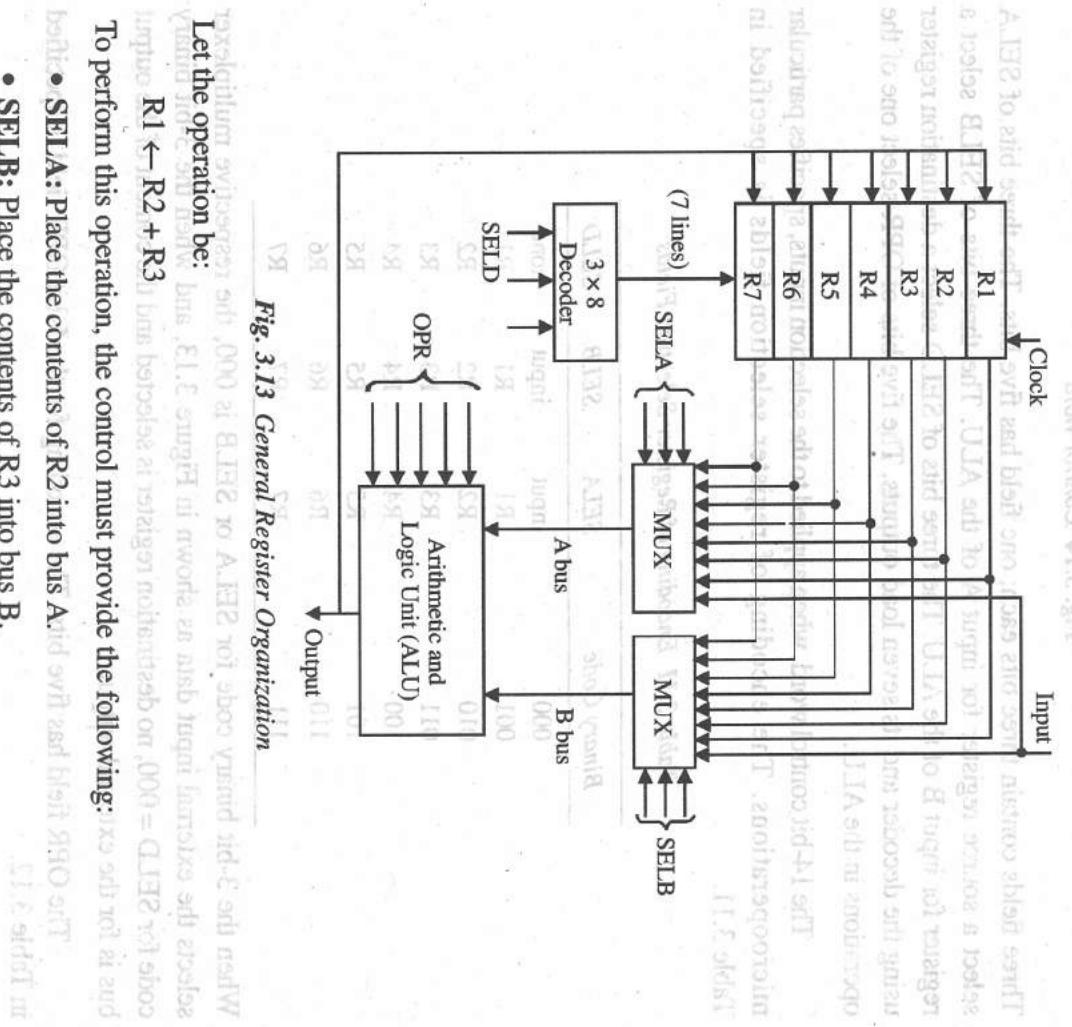


Fig. 3.13 General Register Organization

Let the operation be:

$R1 \leftarrow R2 + R3$

To perform this operation, the control must provide the following:

- SELA: Place the contents of R3 into bus A.
- SELB: Place the contents of R2 into bus B.

- ALU: Operation Selector OPR: Provide the arithmetic addition A + B.
- SELD: Transfer the contents of the output bus R1.

NOTES

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.

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 3.14. 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 microoperations. The encoding of register selection fields is specified in Table 3.11.

Table 3.11 Encoding of Register Selection Fields

Binary Code	SELA	SELB	SELD
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 3.13, 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 3.12.

Fig. 3.14 Control Word

Table 3.12 Encoding of ALU Operation

Central Processing Unit

OPR	Operation	Symbol	NOTES
00000	Transfer A	TSFA	
00001	Increment A	INCA	
00010	Addition	ADD	
00011	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	

Let the microoperation given by the statement be expressed as follows:

$$R1 \leftarrow R4 \wedge R5$$

This statement specifies R4 for input A of the ALU, R5 for input B of the ALU, and R1 as the destination register. The microoperation to be performed is the AND (\wedge) operation between R4 and R5. The control word for the above statement according to Tables 3.11 and 3.12 is summarized in Table 3.13.

Table 3.13 Control Word as per Tables 3.11 and 3.12

SELA	SELB	SELD	OPR
R1	R4	R5	AND

Thus, the control word is 001 100 101 01000.

3.4.3 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:

- Step 1: Fetch:** In this step, an instruction is loaded from the memory into the CPU registers. All the instructions must be fetched before they can be executed.
- Step 2: Decode:** In this step, the control unit decodes the instructions.
- Step 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.
- Step 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.

An instruction cycle sometimes called fetch-and-execute cycle, Fetch-Decode-eXecute or FDX cycle is the basic operation cycle of a computer. It is the process by which a computer retrieves a program instruction from its memory, determines what actions the instruction requires, and carries out those actions. This cycle is repeated continuously by the CPU until the computer is shut down. Figure 3.15 illustrates the instruction cycle.

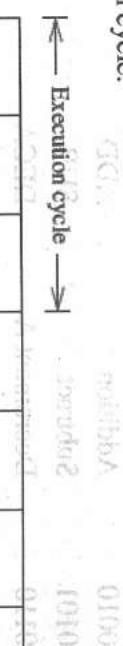


Fig. 3.15. Instruction Cycle

3.4.4 Instruction Pipelining

An instruction pipeline operates on a stream of instructions by overlapping the fetch, decode and execute phases of the instruction cycle.

An instruction pipeline reads the consecutive instruction from the memory when the previous instruction is executing in other segments. In this way, we can overlap the instructions fetch and execute phases and perform operations simultaneously. The instruction mainly involves the following sequences of steps:

- **Instruction Fetch:** This step is required to fetch the instruction from memory.
- **Instruction Decode:** This step is required to decode the instruction.
- **Calculate Address:** This step is required to calculate the effective address of operands.
- **Operand Fetch:** This step is required to fetch the operands from memory.
- **Operation Execution:** This step is required to execute the instruction.
- **Result Storing:** This step is required to store the result in proper place.

In a non-pipelined computer, all the above steps are performed for the execution of an instruction and then the next instruction is fetched from the 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 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 may require memory access at the same time, 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 3.16 where the decoding of the instruction and calculation of the effective address are combined into one segment. Also, the execution of an instruction is combined with the storing of result into one segment as most of the instructions place the result into the processor register after executing the instruction.

Segment 1 : Fetch instruction from memory
 Segment 2 : Decode instruction and calculate effective address.
 Segment 3 : Fetch operand from memory
 Segment 4 : Execute instruction

• If an interrupt occurs in segment 2, then the pipeline is flushed and the PC is updated.

• If an interrupt occurs in segment 3, then the pipeline is flushed and the PC is updated.

• If an interrupt occurs in segment 4, then the pipeline is flushed and the PC is updated.

• If an interrupt occurs in segment 1, then the pipeline is flushed and the PC is updated.

NOTES

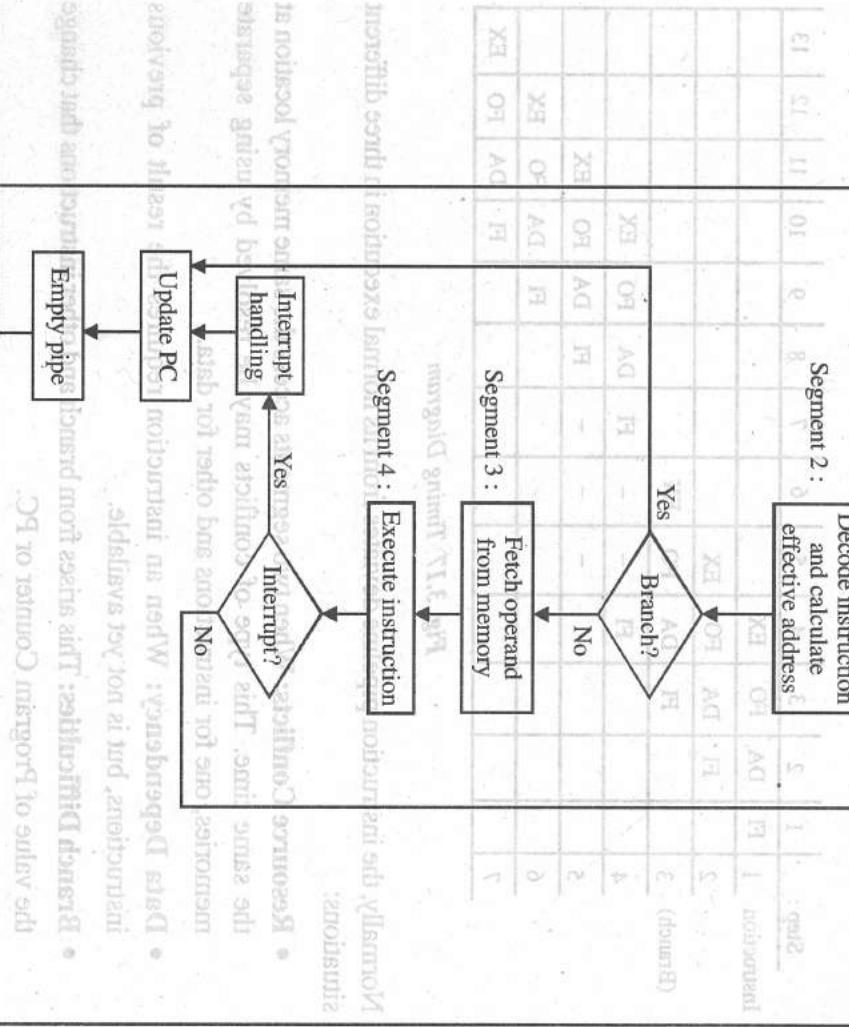


Fig. 3.16 Four-Segment Instruction Pipeline

If an instruction in a 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 3.17 shows the working of an instruction pipeline. Horizontal axis shows the time, which is divided into steps of equal duration. Vertical axis shows instructions. The four segments are abbreviated as follows:

- FI for fetch an instruction.
- DA for decodes instruction and calculate effective address.
- FO for fetch the operand.
- EX for execute the instruction.

Also, 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 Step 4, the instruction 1 is executed in segment EX, instruction 2 is executed in segment FO, instruction 3 is

being decoded in segment DA and instruction 4 is in segment FI. Let the instruction 3 is a branch instruction. When this instruction is decoded in segment DA, the transfer of other instructions from FI to DA is halted until the branch instruction is executed in Step 6. A new instruction is then fetched in Step 7 and pipeline continues until a new branch instruction is encountered again.

NOTES

Step :	1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction	FI	DA	FO	EX									
2		FI	DA	FO	EX								
(Branch)			FI	DA	FO	EX							
3				FI	—	—	FI	DA	FO	EX			
4					—	—	—	FI	DA	FO	EX		
5						—	—	FI	DA	FO	EX		
6							—	FI	DA	FO	EX		
7								FI	DA	FO	EX		

Fig. 3.17 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 the same time. This type of conflicts may be resolved by using separate memories, one for instructions and other for data.
- **Data Dependency:** When an instruction requires the result of previous instructions, but is not yet available.
- **Branch Difficulties:** This arises from branch and other instructions that change the value of Program Counter or PC.

3.5 REDUCED INSTRUCTION SET COMPUTING OR RISC

RISC, which stands for Reduced Instruction Set Computing, gives a new generation of faster and cheaper machines. The basic idea behind RISC is to use a small set of instructions with simple constructs so that it can execute them much faster within the CPU.

The major characteristics of a RISC processor are as follows:

- Relatively few and simple instruction types.
- Fixed and easily decoded instruction formats.
- Memory access limited mainly to load and store instructions.
- Relatively few addressing modes.
- A large number of register sets.
- Fast single cycle instruction execution.
- Hardwired rather than microprogrammed control.

- Use of instruction pipeline and cache memory.
- Use of compilers to optimize object code performance.

Simple and smaller instruction set leads to simple hardware and faster execution. The small instruction set mostly consists of register-to-register operations with simple load and store operations for memory access. All operations are done within the CPU registers. Thus, with the help of load instruction, first the operands are brought into processor registers and with the help of store instructions, the results are transferred to the memory. Also, the RISC processor uses only few addressing modes, and thus all instructions have only simple register addressing. The use of hardwired rather than microprogrammed control, helps in faster execution of operations. Another important aspect of the RISC instruction format is that the instruction lengths are fixed and are easy to decode. RISC processors have the ability to execute one instruction pipelining by overlapping the fetch, decode and execute phases of two or more instructions. A large number of registers in the RISC processor are useful for storing intermediate results and in minimizing the register memory operations by keeping the most frequently accessed operands in register.

RISC is a type of microprocessor that utilizes high optimized set of instructions. John Cocke of IBM, New York introduced RISC concept in 1974. The first RISC machine came into existence in the late 1970 and early 1980 from IBM, Stanford and UC Berkeley. Officially, David Patterson introduced the term RISC who was a teacher at the University of California in Berkeley. The Berkeley RISC 1 and Berkeley RISC 2, IBM 801 and Stanford MIPS all were designed as RISC machine. A RISC microprocessor is designed to perform a small number of computer instructions operating at a higher speed. Each instruction requires additional transistors and circuitry to enhance the system performance. The main objective of RISC machines is to make the processor fast by reducing its complex process and by adding more registers. They also contain some transistors to control circuitry functions. Using this mechanism, they follow some instruction formats and low semantic instructions. They save the time because of choosing the requirements for a selected programming task, such as designers choose 'C' high level language frequently. Most of the RISC processors allow quick frequently used data due to having large number of registers. These registers are dual-ported memory. The dual-ported memory allows two simultaneous accesses at different memory address to fetch source operands at every cycle. The low semantic instructions supports to RISC processors which have high bandwidth for keeping instructions flowing into the CPU. The RISC processors also support the cache memory to attain fast and enhance performance. They retain internal instruction pipeline for extra hardware and compiler requirement to manage the pipeline technique. The interrupts are also received for the pipeline resources that is saved and restored after processing. The RISC implementation strategies must have compilers, such as scheduling pipeline which avoid data hazards and filling branch which delay shots and managing allocation and spilling the registers which reduce the complexity of executing set of instructions. The RISC instructions for processors are closer to single step microprocessor. The features of RISC processors are as follows:

- RISC processors contain a CPI or Clocks per Instruction as one cycle. This is happened to optimize of each instruction on the CPU by a technique known as single cycle.

NOTES

NOTES

- These processors support pipelining technique which allows execution of instructions more efficiently. The RISC machines contain the fast chips so that they can use pipelining more efficiently. Pipelining is a design technique in which RISC machines process more than one instruction at a time and one processing instruction does not complete before starting the next instruction. Pipelining technique does not improve the latency of instructions and overall throughput. Sometimes, in complex set of instructions, pipelining process takes more than one clock to complete a phase.

• These processors need large number of registers that prevent a large amount of interactions with memory.

In RISC processors, the chips with multiple units are an important parameter that is required to the pipeline depth of each unit. Floating-point units are implemented with a deeper pipeline, taking into account the longer latency of floating-point operations. The pipelines of different depth are coordinated to avoid collisions at the exit of the pipelines, when more than one unit accesses the register file. In RISC processors, the maintenance of register file is an essential feature. There are three different ways to solve the scheduling problem for the usage of registers:

- The first solution is to schedule registers in software and to avoid collisions through a sophisticated compile time analysis.
- The second solution relies on the help of a special hardware ‘scoreboard’ that tracks the usage and availability of registers. Whenever request is sent to a register, which is not free, the scoreboard locks the request until the register is made available.
- The third solution comes from the mainframe world and was implemented by IBM in the RS/6000 processors, i.e., registers are dynamically renamed by the hardware. If two instructions need register R2 to generate a temporary result, one of the two gets access to this register and the other to a ‘copy’ of R2. The results are calculated and the real R2 is updated according to the sequential order of the calling instructions.

The RISC coprocessor architecture use own registers. Registers can be interchanged through memory. A common register file is accessed by all execution units which work with private registers. The Motorola 88000 is a processor with a common register file. The IBM RS/6000 uses private registers in its execution units. The word width is the very important feature of the RISC processor. Most RISC processors use 32-bit internal and external word width. The integer registers, the address and the data paths are assembled with the number of bits in word width. Some of the RISC processors, such as IBM RS/6000 use full-fledged 64-bit architecture.

RISC Machines

Following are the examples of RISC machines:

- **Million Instructions Per Second or MIPS Architecture:** MIPS works with microprocessor without interlocking pipeline stages. The objective of the MIPS designers is to produce a RISC processor with deep pipelining and pipeline interlocking controlled by software. If one instruction requires two cycles to

complete, it is the function of the compiler to schedule one instruction at a specific time. In this way, only pipeline bubbles during execution. This reduces the amount of hardware needed in the processor. The MIPS R2000 is a 32-bit processor with an off-chip split cache for instructions and data. A write buffer is used to handle all data writes to memory. The MIPS chip set follows radical coprocessor architecture. There are 32 general purpose integer registers and 16 separate 64-bit floating point registers. The floating point coprocessor contains add, divide and multiply unit. There are no condition code bits and no floating scoreboard.

- **Scalable Processor ARChitecture or SPARC:** The SPARC architecture was originally defined by Sun Microsystems but it is not a proprietary design. Any interested semiconductor company can get a license to build a SPARC processor in any desired technology. The SPARC is a 32-bit processor with an off-chip common cache. Three chips provide the functionality needed: one for the integer unit, one for the floating-point unit and another works as a cache controller and memory management unit. The SPARC design follows the coprocessor architectural paradigm. Floating-point unit and integer unit exchange information through memory and through some control lines. There is no prefetch buffer. A common integer register file with two read and one write port is used. The floating point unit provides 32 registers 32 bits wide. Instructions are decoded in parallel by the integer and floating point unit. Floating-point instructions are then started when the integer unit sets a control line. The SPARC is a RISC oriented design. The SPARC uses the concept of 'register Windows' in order to eliminate the load and stores to a stack associated with procedure calls. Register Windows are hardware oriented method to optimize register allocation. Another advantage of the SPARC is to programmed with instructions. The SPARC provides instructions which make easier to handle a 2-bit tag in each word of memory. In all other architectural respects, the SPARC is very similar to the MIPS machine but the number of addressing modes is higher, for example two addressing mode in the SPARC and one addressing mode in the MIPS processor.

- **IBM RS/6000 Architecture:** The IBM RS/6000 or Performance Optimization with Enhanced RISC (POWER) architecture shares with older RISC designs the streamlined approach to pipelined execution. But, the instruction set of the IBM processor is large and many special instructions are provided in order to speed up execution. The POWER chip set is indeed an impressive computing engine. The RS/6000 is a 32-bit processor. Split external caches are used in this processor. The processor follows Harvard architecture with separate buses for instructions and data. The data bus is 64 bits wide in order to read and store 64-bit floating point data in a single cycle. The RS/6000 architecture is one of multiple units and consists of three main blocks; one for control and branching, one for integer operations and another for floating-point. Integer operations then go through six pipeline stages and floating-point operations through eight pipeline stages. This approach of pipelining uncommon in workstations. Other RISC processors do not employ with pipelined floating-point units. The RS/6000 has one addressing mode and an additional auto increment mode. The

NOTES

as it is not auto increment mode is CISC processors but it was included in the RS/6000 to combat gain some speed trying to avoid compromising the pipeline flow.

- **Motorola 88000 Family:** The 88100 is a RISC processor, the first in the 88000 family and internal architecture. There are separate buses for instruction and data, i.e., the processor follows a Harvard architectural model. There is no prefetch buffer and the processor follows the multiple units approach. There is one integer unit and two floating-point units (adder and multiplier). The register file is common to all units and contains 32 registers of 32 bits. Register 0 is hardwired to 0 address. Registers can contain integer or floating point data. The M88100 uses three different addressing modes: register plus offset, register plus register, and register plus scaled register. The last two addressing modes provide easy access to arrays in memory. Two general purpose registers are concatenated when 64 bits floating-point data is needed.

• **Intel 860:** Intel developed the 80860 processor with embedded applications in mind. It was the first RISC chip of the semiconductor manufacturer in which silicon area was not spared. More than one million transistors were used in the final design. The Intel 860 is a 32-bit processor built with Harvard architecture. The bus to the instruction cache is 32 bits wide and the bus to the data cache is 128 bits wide making possible to access four words in parallel. The 'RISC core' contains thirty two 32-bit registers and one ALU. A scoreboard controls the allocation of general purpose registers. The floating-point register file contains 30 registers 32 bits wide.

- **Hewlett Packard's Precision Architecture:** When Hewlett-Packard charged their computer architects with designing new processor architecture, it unified the different product lines of Hewlett-Packard and with RISC design. The number of different instructions formats is larger than in other RISC machines. Twelve different combinations of operation code and register or constant fields are used with Scalable Processor ARChitecture or SPARC and MIPS processors.

7. What does special purpose register hold in the control unit?
8. What is an internal processor memory?
9. Name some of the examples of secondary storage memory.
10. On what an instruction pipeline operates?
11. What is the basic idea behind RISC?
12. Who introduced RISC concept?
13. What does 'RISC core' contain?

Check Your Progress

Innosim 32-bit processor with a non Harvard architecture. The Innos implementation does not provide a cache although there is an internal on-chip memory which must be explicitly addressed while system design. Transputer follows the coprocessor paradigm. The floating-point coprocessor operates with its private floating-point registers and the integer unit implements a stack model. A unique feature of the transputer is its 4 serial links which make it possible to connect arrays of transputers with little additional hardware. The number of different instructions in the transputer is greater than 128.

3.6 INSTRUCTION LEVEL PARALLELISM AND SUPERSCALAR PROCESSORS

Instruction Level Parallelism (ILP) is a measure of the number of instructions that can be performed during a single clock cycle. Very Long Instruction Word or VLIW

refers to a CPU architecture designed to take advantage of ILP. A processor that executes every instruction one after the other, i.e., a non-pipelined scalar architecture can use processor resources inefficiently, potentially leading to poor performance. The performance can be improved by executing different sub-steps of sequential instructions simultaneously (pipelining) or by executing multiple instructions entirely executing instructions in an order different from the order they appear in the program; this is called out-of-order execution. Superscalar uses hardware to detect instructions while VLIW uses off-line software (compiler) to perform this task. In superscalar, the instructions complete out-of-order and have to be reordered to avoid hazards. In VLIW, the compiler works in correct order and hence it is more predictable. Superscalar forms different execution bundles depending on the previous piece of software (different calls to a subroutine) while VLIW execution bundles are predefined in the instruction code. Both need high bandwidth to memory and a multiported register file to insert operands to the parallel units. Superscalar and superscalar pipelines are shown in Figure 3.18. Superscalar pipeline is more complex and requires more hardware resources in which data memory access unit only assumed in the pipeline diagrams. The VLIW pipeline resembles more the conventional RISC pipeline.

NOTES

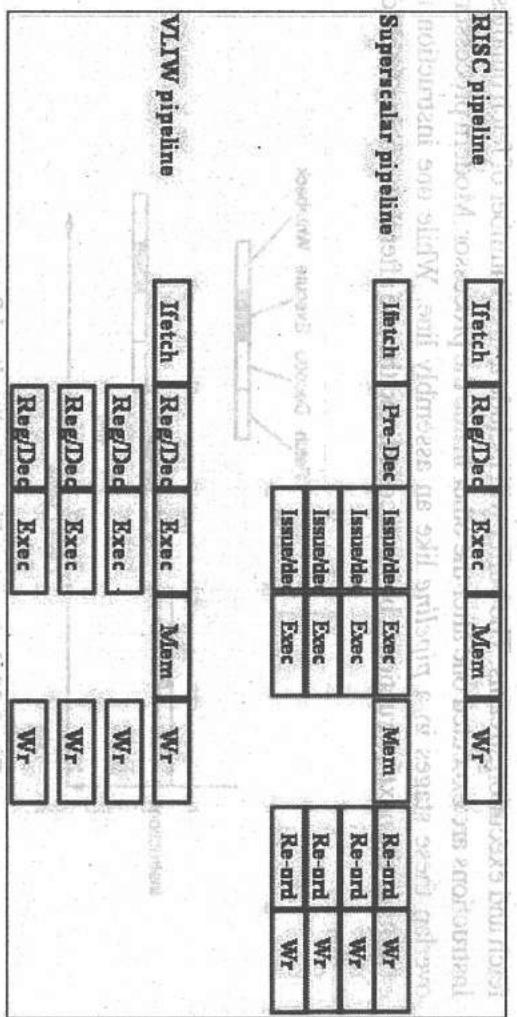


Fig. 3.18 RISC and Superscalar Pipeline

The drawbacks of these architectures can be summarized as follows. Superscalar lessens the implementation of complex hardware needed for dynamic instruction issue and reordering. The number of parallel units does not scale because of the dynamic ordering and register file problems. In VLIW, if orthogonality (setup for square matrix) is required because the register file complexity grows very quickly with the number of execution units. Thus, it does not scale well either. In VLIW there is also overhead from long instruction word if variable-length execution bundles are not supported. This task is done in the case of cost-effective embedded VLIW cores. Figure 3.19 illustrates the instruction coding problem. If all the No Operations or NOPs for the unused execution units will be coded in the instruction word, it will definitely be long because equally long independently of the units is exploited.

QUESTION: What is the main difference between superscalar and VLIW architectures?

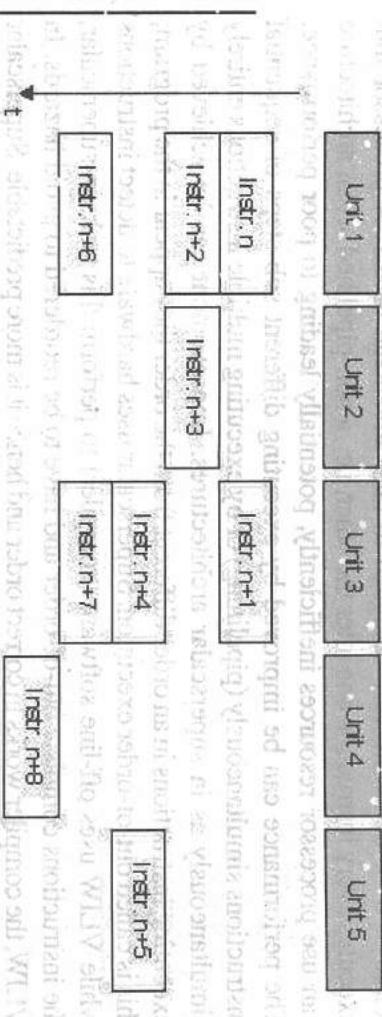
NOTES

Fig. 3.19 Instruction Coding Problem

If you separate the execution bundle (operations/instructions launched in parallel) from the fetch bundle (operations/instructions fetched from the memory in parallel), the concept of parallelism is usable in the algorithms without carrying the overhead of NOP operations. This requires that the border of the execution bundle is marked in the instructions. The principle of an instruction buffer is used to match the fetch and execution streams. The buffer will include a small number of fetch bundles. Instructions are executed one after the other inside the processor. Modern processors overlap these stages in a *pipeline* like an assembly line. While one instruction is executing, the next instruction is being decoded, and the one after that is being fetched.

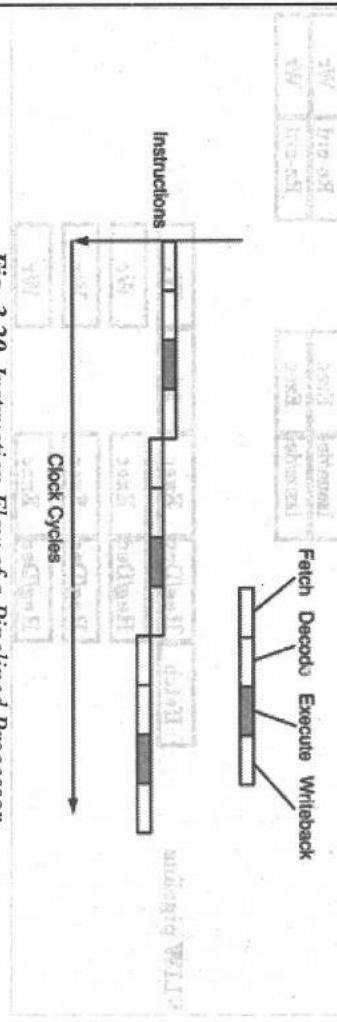


Fig. 3.20 Instruction Flow of a Pipelined Processor

Figure 3.20 illustrates instruction flow of a pipelined processor. Each pipeline stage consists of some combinatorial logic and possibly access to a register set and some form of high speed cache memory. The pipeline stages are separated by latches. A latch is a circuit that has two stable states and can be used to store state information. The circuit can be made to change state by signals applied to one or more control inputs and will have one or two outputs. It is the basic storage element in sequential logic. A common clock signal synchronizes the latches between each stage so that all the latches capture the results produced by the pipeline stages at the same time, in effect, the clock ‘pumps’ instructions decreases the pipeline. At the beginning of each clock cycle, the data and control information for a partially processed instruction is held in a pipeline latch and this information is used to form the inputs to the logic circuits for the next pipeline stage. During the clock cycle, the signals propagate through the combinatorial logic of the stage producing an output can be captured by

the next pipeline latch at the end of the clock cycle in a pipelined microarchitecture (refer Figure 3.21).¹³ Instructions can be fetched from memory or registers and then sent to an instruction decoder which then sends control signals to the pipeline stages. The pipeline stages are Fetch, Decode, Execute, and Writeback.

Fig. 3.21 A Pipelined Microarchitecture This diagram shows a four-stage pipeline: Fetch, Decode, Execute, and Writeback. It illustrates the flow of instructions over four clock cycles. The first instruction starts at the beginning of the first cycle. The second instruction begins its fetch stage in the first cycle and enters the decode stage in the second cycle. The third instruction begins its fetch stage in the second cycle and enters the decode stage in the third cycle. The fourth instruction begins its fetch stage in the third cycle and enters the decode stage in the fourth cycle. The writeback stage for each instruction occurs immediately after its execution stage. The final result of each instruction is written back to memory or registers at the end of the fourth cycle.

NOTES

Since the result from each instruction is available after the execute stage has completed, the next instruction is used that distinct value which waits for the committed result. This process occurs in destination register in the writeback stage (refer Figure 3.22). To implement this mechanism, forwarding lines called *bypasses*, are added by shifting backwards along the pipeline.

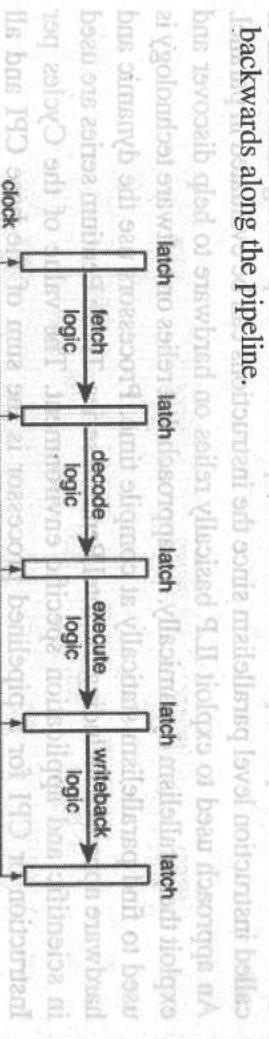


Fig. 3.22 A Pipelined Microarchitecture with Bypasses

In the pipeline stages, *execute* stage is really made up of several sets of logic gates which make them as *functional units* for each type of operation that the processor must be able to perform. Instruction level parallelism is a measure of the number of instructions that can be performed during a single clock cycle. Parallel instructions are a set of instructions that do not depend on each other to be executed. Instruction level parallelism is a critical technique used in computer architecture for processor and compiler design. Instruction level parallelism can improve the program execution performance by causing individual machine operations to execute in parallel. Consider the following set of statements:

Statement 1: $e = a + b$

Statement 2: $f = c + d$

Statement 3: $g = e \times f$

Statement 3 depends on the results of Statements 1 and 2, respectively so it cannot be calculated until both of them are completed. However, Statements 1 and 2 do not depend on any other operation so they can be calculated simultaneously. If we assume that each operation can be completed in one unit of time then these three instructions can be completed in a total of two units of time giving an instruction level parallelism of 3/2. A goal of compiler and processor designers is to identify and take advantage of as much instruction level parallelism as possible. Ordinary programs are typically

NOTES

written under a sequential execution model where instructions execute one after the other and in the order specified by the programmer. Instruction level parallelism allows the compiler and the processor to overlap the execution of multiple instructions or even to change the order in which instructions have to be executed. The pipeline is used to increase the performance of processors. The longer the pipeline, the harder to keep it full in all occasions. If a pipeline is full of useful instructions, it completes at most one every clock cycle and hence known as 'Flynn limit'. If multiple function units and multiple instructions have been fetched, then it is possible to start several instructions at once. The two approaches execute several instructions in parallel within a single processor are superscalar and VLIW. In superscalar, many prefetched instructions are dynamically issued to idle function units as possible. In VLIW, long instruction words are statically compiled with many operations and each for a different function unit. In such type of multiuse processors, the different types of function units are used as floating point, integer, multiply/divide, branch and load/store. There can be more than one of the same type, for example several load-store units. Each function unit can be itself pipelined. All processors use pipelining to overlap the execution of instructions and improve performance. This potential overlap among instructions is called instruction level parallelism since the instructions can be evaluated in parallel. An approach used to exploit ILP basically relies on hardware to help discover and exploit the parallelism dynamically. An approach that relies on software technology is used to find parallelism statically at compile time. Processors use the dynamic and hardware approach including the Intel Pentium series. These Pentium series are used in scientific and application specific environment. The value of the Cycles per Instruction or CPI for a pipelined processor is the sum of the base CPI and all contributions from stalls:

$$\text{Pipeline CPI} = \text{Ideal pipeline CPI} + \text{Structural stalls} + \text{Data hazard stalls} + \text{Control stalls}$$

The pipeline CPI is a measure of the maximum performance attainable by the implementation. By reducing each of the terms of the right hand side, the pipeline is minimized or alternatively increases the instructions per clock. The amount of parallelism, available within the basic block, a straight line code sequence with no branches in except to the entry and no branches out except at the exit, is small. For MIPS programs, the average dynamic branch frequency is often between 15 per cent and 25 per cent which means that three and six instructions can be executed between a pair of branches. Since these instructions are likely to depend upon one another, the amount of overlap you can exploit within a basic block is likely to be less than the average basic block size. The simplest way to increase the ILP is to exploit parallelism among iterations of a loop. This type of parallelism is often called loop-level parallelism. Following is an example of simple loop which adds two 1000 element arrays that is processed completely parallel. It is assigned to a larger set of hardware resources.

```

for (i=0; i<1000; i=i+1)
    for (j=0; j<1000; j=j+1)
        x[i]=x[i]+y[j];
    
```

The iteration of the loop can overlap with any other iteration within each loop is little or no opportunity for overlap. There are a number of techniques that is to be examined for converting such loop-level parallelism into instruction-level parallelism. Basically,

such techniques work by unrolling the loop either statistically by the compiler or dynamically by the hardware. An alternative method used for exploiting loop-level parallelism by operating on data items in parallel. For example, the above code sequence could execute in four instructions on some vector processors, two instructions to load the vectors x and y from memory, one instruction to add the two vectors and an instruction to store back the result vector. These instructions can be pipelined and have relatively long latencies by these latencies can be overlapped. Vector instruction sets are used in graphics, digital signal processing and multimedia applications. Instruction level parallelism in Superscalar processors has complex bypassing hardware that forwards the results of each instruction to all of the execution units to reduce the delay between dependent instructions. The instructions that make up a program are handled in superscalar processors by the instruction issue logic which issues instructions to the units in parallel.

Superscalar Processors

Several instructions are executed in the same clock cycle by the processor. Such processors are capable of achieving an instruction execution throughput of more than one instruction per cycle. They are known as superscalar processors. Superscalar processing is used for fast microprocessors. By exploiting instruction-level parallelism, superscalar processors are capable of executing more than one instruction in a clock cycle. Superscalar processing, the ability to initiate multiple instructions during the same clock cycle, is known as latest architectural innovations aimed at producing fast microprocessors. Superscalar microprocessors are now being designed and produced by all the microprocessor vendors for high-end products. The superscalar methods have been applied to a spectrum of instruction sets, ranging from the Digital Equipment Corporation or DEC Alpha, the newest RISC instruction set to the decidedly non-RISC Intel x86 instruction set. A superscalar processor fetches and decodes the incoming instruction stream several instructions at a time. As part of the instruction fetching process, the outcomes of conditional branch instructions are usually predicted in advance to ensure an uninterrupted stream of instructions. The incoming instruction stream is then analysed for data dependencies and instructions are distributed to functional units according to instruction type. Next, instructions are initiated for execution in parallel based primarily on the availability of operand data rather than their original program sequence. This important feature, present in many superscalar implementations, is referred to as dynamic instruction scheduling. Upon completion, instruction results are re-sequenced so that they can be used to update the process state in the correct (original) program order in the event that an interrupt condition occurs. A superscalar CPU architecture implements a form of parallelism called instruction-level parallelism within a single processor. It allows fast CPU throughput for a given clock rate. A superscalar processor executes more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to redundant functional units on the processor. Each functional unit is not a separate CPU core but an execution resource is accessed within a single CPU, such as an arithmetic logic unit, a bit shifter or a multiplier. Figure 3.23 illustrates the microarchitecture or hardware organization of a typical superscalar processor. The prime parts of the

NOTES

NOTES

microarchitecture are instruction fetch and branch prediction, decode and register dependence analysis, issue and execution, memory operation analysis and execution, and instruction reorder and commit. These phases are listed in the same order as instructions flow through them. The underlying pipeline stages are to some extent a lower level logic design issue depending on how much work is done in each pipeline stage. This will affect the clock period and causes some very important design trade-offs regarding the degree of pipelining and the width of parallel instruction issue.

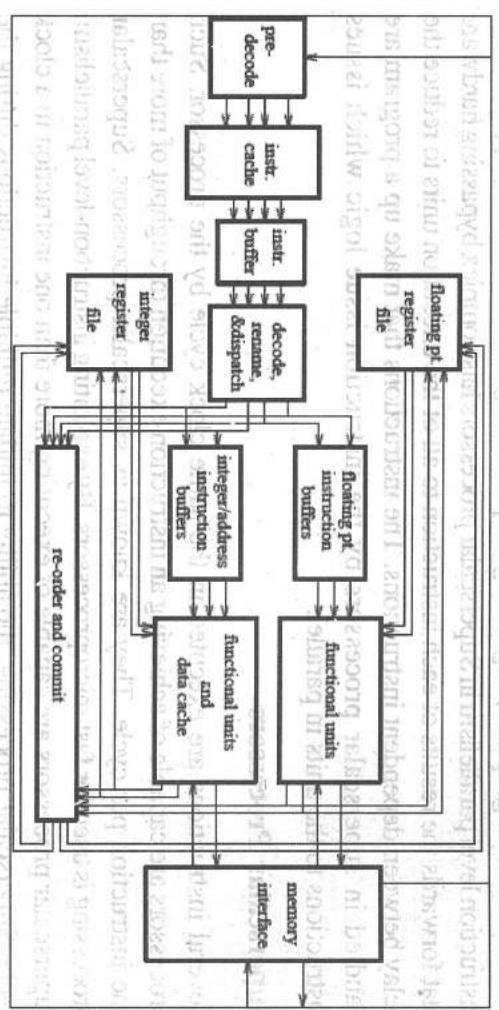


Fig. 3.23 Organization of a Superscalar Processor

Multiple paths connecting units are used to illustrate a typical level of parallelism. For example, four instructions can be fetched in parallel, two floating-point instructions can complete in parallel, etc. While a superscalar CPU is also pipelined, pipelining and superscalar architecture are considered different performance enhancement techniques. The superscalar technique associated with several identifying characteristics within a given CPU core is implemented as follows:

- Instructions are issued from a sequential instruction stream.
- CPU hardware dynamically checks for data dependencies between instructions at runtime.
- The CPU accepts multiple instructions per clock cycle.

The simplest processors are scalar processors. Each instruction executed by a scalar processor typically manipulates one or two data items at a time. Also, each instruction executed by a vector processor operates simultaneously on many data items. An analogy is the difference between scalar and vector arithmetic. In superscalar, each instruction processes one data item but there are multiple redundant functional units within each CPU thus multiple instructions can be processed with separate data items concurrently. Superscalar CPU design improves the instruction dispatcher accuracy and allows it to keep the multiple functional units at all times. This has become increasingly important when the number of units increased. While early superscalar CPUs had two ALUs and a single Floating-Point Unit or FPU but in modern design, such as the PowerPC

970 includes four ALUs, two FPUs and two SIMD units. If the dispatcher is ineffective at keeping all of these units fed with instructions, the performance of the system will decrease. A superscalar processor usually sustains an execution rate in excess of one instruction per machine cycle. But, processing multiple instructions concurrently does not make architecture superscalar since pipelined multiprocessor or multi-core architectures also achieve that but with different methods. In a superscalar CPU, the dispatcher reads instructions from memory and decides which is run in parallel dispatching them to redundant functional units contained inside a single CPU. Therefore, a superscalar processor can be envisioned having multiple parallel pipelines and each of which is processing instructions simultaneously from a single instruction thread.

NOTES

Superscalar Microprocessors

The current superscalar microprocessors are MIPS R10000, DEC Alpha 21164 and AMD K5. They are discussed below:

MIPS R10000

The MIPS R10000 microprocessor uses dynamic scheduling. It fetches four instructions at a time from the instruction cache. These instructions have been pre-decoded when they are put into the cache. The pre-decode method generates four additional bits per instruction which helps to determine the instruction type immediately after the instructions are fetched from the cache. After being fetched, branch instructions are predicted. The prediction table is contained within the instruction cache mechanism. The instruction cache holds 512 lines and there are 512 entries in the prediction table. Each entry in the prediction table holds a 2-bit counter value that encodes history information used to make the prediction. When a branch is predicted to be taken, it takes a cycle to redirect instruction fetching. All register designators are renamed using physical register files that are twice the size of the logical files, for example 64 physical and 32 logical files. The destination register is assigned for an unused physical register from the free list and the map is updated to reflect the new logical-to-physical mapping. Operand registers are given the correct designators by reading the map. Then, up to four instructions at a time are dispatched into one of three instruction queues memory, integer and floating-point.

Alpha 21164

The Alpha 21164 is an example of a simple superscalar processor that features dynamic instruction scheduling with the help of high clock rate (refer Figure 3.24). Instructions are fetched from an 8KB instruction cache four at a time. These instructions are placed in one of two instruction buffers, each capable of holding four instructions. Instructions are issued from the buffers in program order and a buffer must be completely emptied before the next buffer can be used. This restricts the instruction issue rate but it greatly simplifies the necessary control logic. Branches are predicted using a table that is associated with the instruction cache. There is a branch history entry for each instruction in the cache and each entry is a 2-bit counter.

Figure 3.24 shows the Alpha 21164 microprocessor architecture. It consists of an instruction cache, two instruction buffers, a branch predictor, and a central processing unit. The instruction cache is 8KB and feeds into two instruction buffers. Each buffer can hold four instructions. The branch predictor is a table-based predictor. The central processing unit contains four ALUs, two FPUs, and two SIMD units. The buffers feed into the central processing unit, which then issues instructions to memory and functional units.

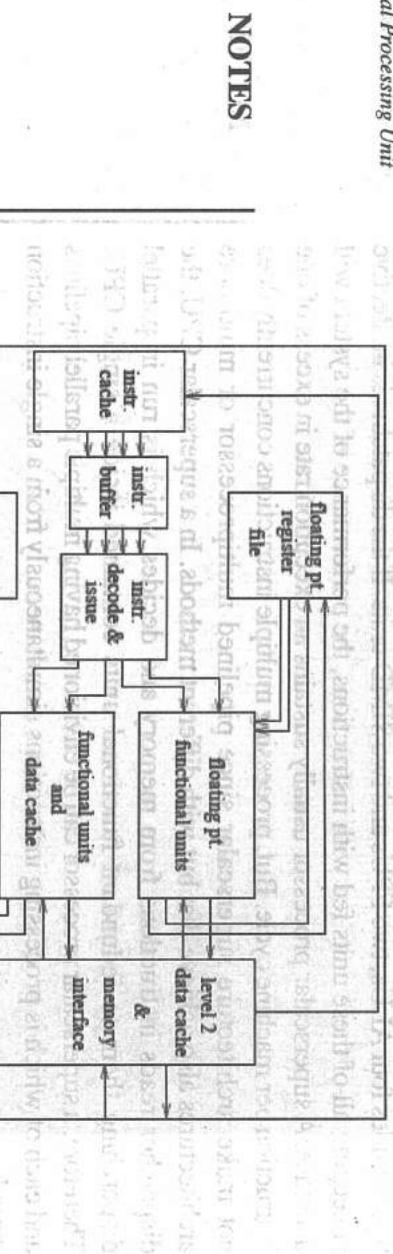


Fig. 3.24 DEC Alpha 21164 Superscalar Organization

There are four functional units in which two integer ALUs, a floating point adder and a floating point multiplier work together. An integer ALUs are not identical in which only one can perform shifts and integer multiplies but the other is the only one that can evaluate branches. The 21164 has two levels of cache memory on the chip. There is a pair of small and fast primary caches of 8KB and each one for instructions and one for data. The secondary cache, shared by instructions and data, is 96KB. The small direct mapped primary cache allows single cycle cache which accesses at a very high clock rate. The primary cache can sustain a number of outstanding misses. There is a six entry Miss Address File (MAF) that contains the address and target register for a load that misses. If the address is to the same line is linked with another address in the MAF, the two are merged into the same entry. Hence, the MAF can hold many more than 6 cache misses. To provide a sequential state at the time of an interrupt, it does not issue out of order and keeps instructions in sequence as they flow down the pipeline. The final pipeline stage updates the register file in original program sequence.

AMD K5

The AMD K5 (refer Figure 3.25) implements an instruction set that was not originally designed for fast pipelined implementation. It is an example of a complex instruction set, i.e., Intel x86. The Intel x86 instruction set uses variable length instructions. One instruction must be decoded and its length established before the beginning of the next can be found. Consequently, instructions are sequentially pre-decoded as they are placed into the instruction cache. There are five pre-decode bits per instruction byte. These bits indicate information whether the byte is the beginning or end of an instruction and identifies bytes holding opcodes and operands. Instructions are then fetched from the instruction cache at a rate of up to 16 bytes per cycle. These instruction bytes are placed into a 16 element byte queue where they wait for dispatch. There is one prediction entry per cache line. A single bit reflects the direction taken by the previous execution of the branch. The prediction entry also contains a pointer to the target instruction and this information indicates where in the instruction cache the target can be found. This mechanism reduces the delay for fetching a predicted target instruction.

NOTES

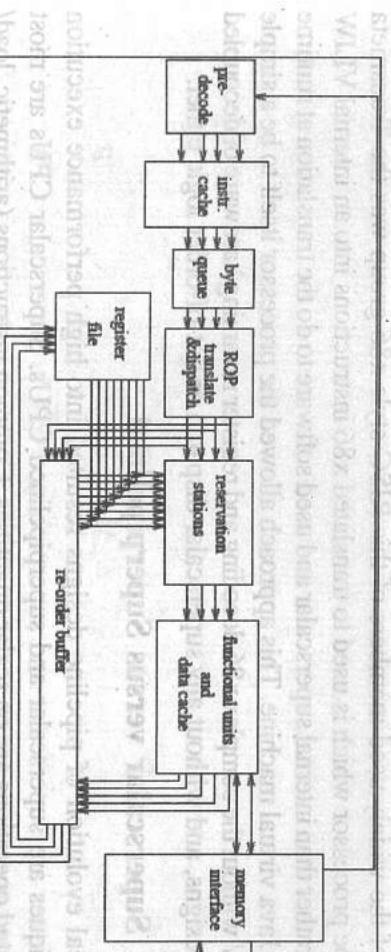


Fig. 3.25 AMD K5 Superscalar Implementation of the Intel x86 Instruction Set

The six functional units contain two integer ALUs, one floating-point unit, two load/store units and a branch unit. One of the integer ALUs is used to perform integer divides. The reservation stations are partitioned among the functional units as illustrated in Figure 3.25. Except the Floating Point Unit (FPU), each of these units has two reservation stations in which the FPU has only one. Based on availability of data, Raster

Operations (ROPs) are issued from the reservation stations to their associated functional units. There are enough register ports and data paths to support up to 4 ROP issues per cycle. There is an 8KB data cache which has 4 banks. Dual load and stores are allowed, provided they are to different banks. A 16 entry reorder buffer is used to maintain a precise process state when there is an exception. There are bypasses from the buffer and an associative lookup is required to find the data if it is complete but holds at the register write operation. This is a reorder buffer method of renaming. The reorder buffer is also used to recover from incorrect branch predictions. While the original Pentium, a superscalar x86, was the complex x86 instruction set. Complex addressing modes and a minimal number of registers meant that few instructions could be executed in parallel due to potential dependencies. The micro-instructions (micro-ops) are often called ‘uops’. For these ‘decoupled’ superscalar x86 processors, register renaming is critical due to the 8 registers of the x86 architecture in 32-bit mode (64-bit mode added another 8 registers) as shown in Figure 3.26.

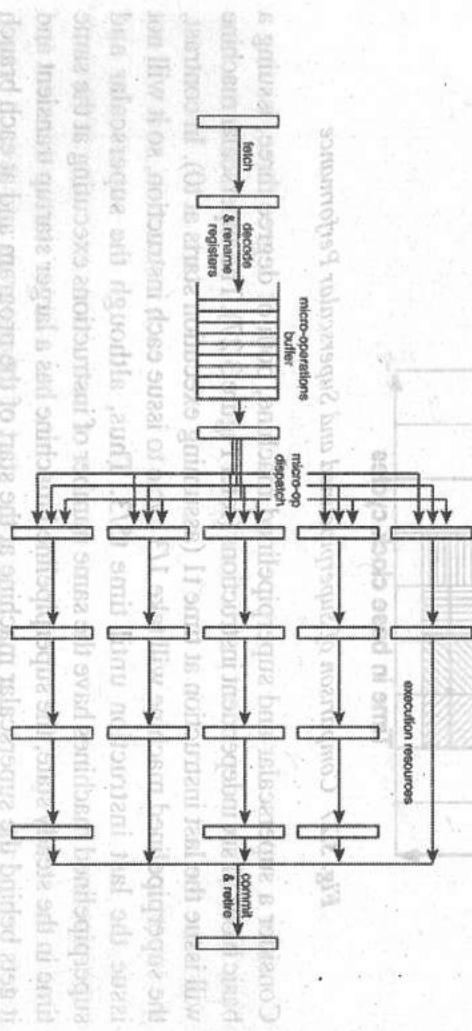


Fig.3.26 A Decoupled Microarchitecture

NOTES

NOTES

One of the widely used members of this RISC style x86 group was the Transmeta Crusoe processor which is used to translate x86 instructions into an internal VLIW form rather than internal superscalar and used software to do the translation at runtime like a Java virtual machine. This approach allowed the processor itself to be a simple VLIW without the complex x86 decoding and register renaming hardware of decoupled x86 designs, and without any superscalar dispatch or out-of-order logic either.

3.6.1 Superscalar versus Superpipelined

Logical evolution of pipeline designs resulted into high performance execution techniques are superscalar and superpipelined CPUs. Superscalar CPUs are most executed operations are on scalar quantities. Common instructions (arithmetic, load/store, conditional branch) can be initiated and executed independently. The superscalar CPU has more than one pipelined functional unit, for example ALU which can operate in parallel. Superpipelined CPUs result from the observation that a large number of pipeline operations do not require a full clock cycle to complete. Dividing the clock cycle into smaller subcycles and subdividing the ‘macro’ pipeline stages into smaller and faster substages means that although the time to complete individual instructions does not change the perceived throughput increases.

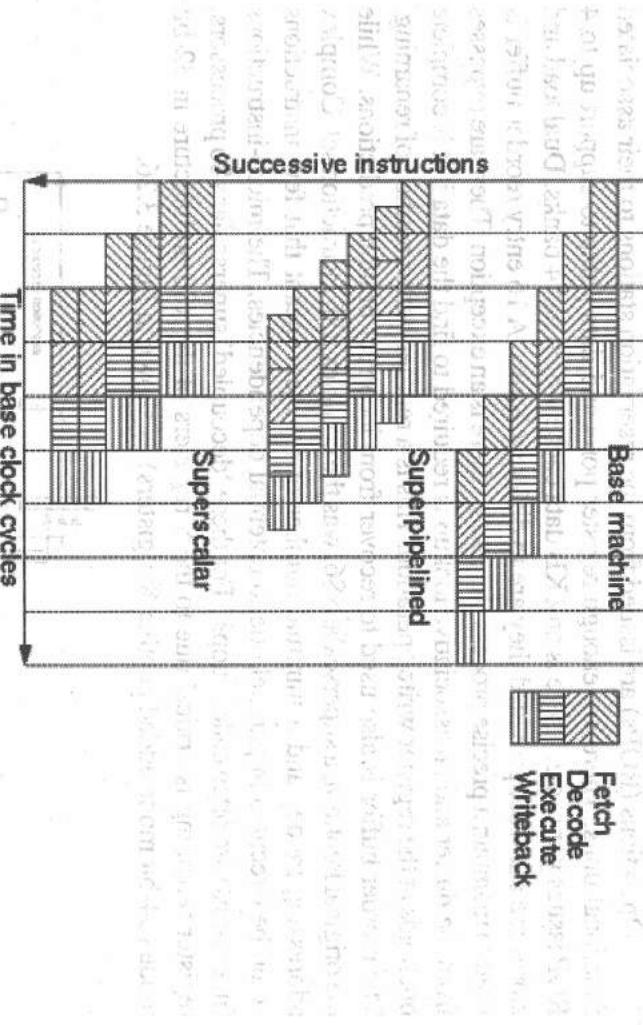


Fig. 3.27 Comparison of Superpipelined and Superscalar Performance

Consider a superscalar and superpipelined machine, both of degree three, issuing a basic block of six independent instructions (refer Figure 3.27). The superscalar machine will issue the last instruction at time t_1 (assuming execution starts at t_0). In contrast, the superpipelined machine will take $1/3$ cycle to issue each instruction, so it will not issue the last instruction until time $t_5/3$. Thus, although the superscalar and superpipelined machines have the same number of instructions executing at the same time in the steady state, the superpipelined machine has a larger startup transient and it gets behind the superscalar machine at the start of the program and at each branch

target. This effect diminishes as the degree of the superpipelined machine increases and all of the issuable instructions are issued closer and closer together. This effect is seen in Figure 3.28 as the superpipelined performance approaches that of the ideal superscalar machine with increasing degree.

2.5

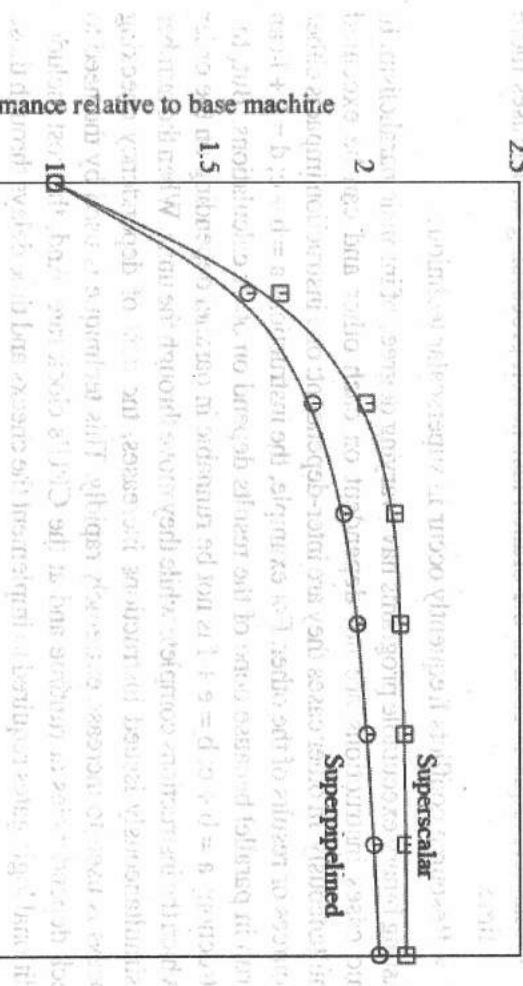


Fig. 3.28 Supersymmetry

Another difference between superscalar and superpipelined machines involves operation latencies that are non-integer multiples of a base machine cycle time. In particular, consider operations which can be performed in less time than a base machine cycle set by the integer add latency, such as logical operations or register-to-register moves. In a base or superscalar machine these operations will require an entire clock because that is by definition the smallest time unit. In a superpipelined machine these instructions might be executed in one superpipelined cycle. Then, in a superscalar machine of degree 3 the latency of a logical or move operation might be 3 times longer than in a superpipelined machine of degree 3. Since, the latency is longer for the superscalar machine, the superpipelined machine will perform better than a superscalar machine of equal degree. In general, when the inherent operation latency is divided by the clock period, the remainder is less on average for machines with shorter clock periods.

3.6.2 Limitations

Following are considered as 'limitations' in available performance improvement from superscalar techniques:

NOTES

- The degree of intrinsic parallelism in the instruction stream is implemented in a limited amount of instruction-level parallelism.
- Superscalar technique supports the complexity and time cost of the dispatcher and associated dependency checking logic.
- This technique is used in the branch-instruction processing which uses more time.

Existing binary executable programs have varying degrees of intrinsic parallelism. In some cases instructions are not dependent on each other and can be executed simultaneously. In other cases they are inter-dependent: one instruction impacts either resources or results of the other. For example, the instructions $a = b + c$; $d = e + f$ can be run in parallel because none of the results depend on other calculations. But, the instructions $a = b + c$; $b = e + f$ is not be runnable in parallel depending on the order in which the instructions complete while they move through the units. When the number of simultaneously issued instructions increases, the cost of dependency checking process is used to increase extremely rapidly. This technique is used by the need to check dependencies at runtime and at the CPU's clock rate. And, the cost includes additional logic gates required to implement the checks and time delays through those gates.

3.6.3 Instruction Issue Policy

Since superscalar processors have to issue multiple instructions per cycle, the first task necessarily is parallel decoding. Clearly, decoding in superscalar processors is a considerably more complex task than in the case of scalar processors and becomes even more sophisticated as the issue rate increases. Higher issue rates, however, can unduly lengthen the decoding cycle or can give rise to multiple decoding cycles unless decoding is enhanced. An increasingly common method of enhancement is predecoding. This is a partial, decoding performed in advance of common decoding while instructions are loaded, into the instruction cache. Therefore, in order to achieve higher performance, superscalar processors have introduced intricate instruction issue policies, involving advanced techniques, such as shelving, register renaming and speculative branch processing. As a consequence, the instruction issue policy used becomes crucial for achieving higher, processor performance.

Shelving

The limitations of the direct issue mode initially applied in superscalar processors raised the need for a more sophisticated issue scheme early on. In the direct issue mode, executable instructions are issued from an issue window directly to the EUs. In an n -way superscalar processor the issue window comprises the last n entries of the instruction buffer (I-buffer). In each clock cycle decoded instructions in this window are checked for dependencies on previous instructions still in execution. In the absence of dependencies all n instructions are executable and will be issued directly to the EUs. However, each dependent instruction in the window gives rise to an issue blockage. Depending on how effectively issue blockages are handled in the processor, they decrease more or less severely the sustained issue rate and cause an issue bottleneck. The direct issue mode severely limits the performance of the processor.

Because of this, high performance processors are forced to employ a more advanced issue mode, such as shelving. Shelving, also known as shelved issue or indirect issue, avoids the limitations of the direct issue mode by utilizing following techniques at the same time:

- First technique supports the decoupling of dependency which is checked from issuing instruction issue.
- Second technique substantially widens the window which is scanned in the processor for executable instructions in each clock cycle.

Shelving is implemented as follows: Instructions are first issued without checking for dependencies on previous instructions still in execution into special buffers called shelving buffers that are provided in front of the EUs as illustrated in Figure 3.29.

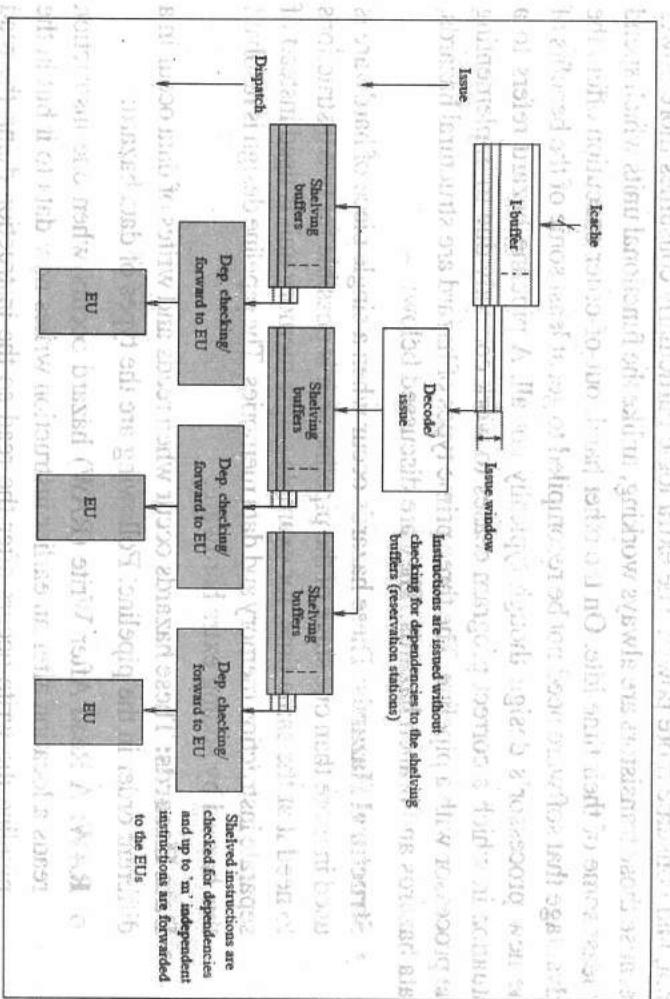


Fig. 3.29 Principle of Shelving

Figure 3.29 illustrates the principle of shelving, assuming a 4-way superscalar processor and individual reservation stations in front of each EU.

Register Renaming

If branches and long latency instructions are going to cause bubbles in the pipelines, then the empty cycles can be used to do other work. To achieve this, the instructions in the program must be reordered so that while one instruction is waiting, other instructions can execute. For example, it might be possible to find other instructions from further down in the program and put them between the two instructions in the earlier multiply example. There are two ways to do this. One approach is to do the reordering in hardware at runtime. Doing dynamic instruction scheduling (reordering) in the processor means the dispatch logic must be enhanced to look at groups of instructions and dispatch them out of order as best it can to use the processor's functional units. This approach is called out-of-order execution. If the processor is going to execute instructions out of order, it will need to keep in mind the dependencies

NOTES

NOTES

between those instructions. This can be made easier by not dealing with the raw architecturally defined registers instead using a set of renamed registers. For example, a store of a register into memory, followed by a load of some other piece of memory into the same register, represent different values and need not go into the same physical register. Furthermore, if these different instructions are mapped to different physical registers they can be executed in parallel which is the whole point of out-of-order execution. So, the processor must keep a mapping of the instructions in flight at any moment and the physical registers they use. This process is called register renaming. As an added bonus, it becomes possible to work with a potentially larger set of real registers in an attempt to extract even more parallelism out of the code. All of this dependency analysis, register renaming and out-of-order execution adds a lot of complex logic to the processor, making it harder to design, larger in terms of chip area, and consume more power. The extra logic particularly consumes more power because those transistors are always working, unlike the functional units which spend at least some of their time idle. On the other hand, out-of-order execution offers the advantage that software need not be recompiled to get at least some of the benefits of the new processor's design though typically not all. A pipeline hazard refers to a situation in which a correct program ceases to work correctly due to implementing the processor with a pipeline. The three prime types of hazard are structural hazards, data hazards and branch hazards. They are discussed below:

- **Structural Hazards:** These hazards occur when a single piece of hardware is used in more than one stage of the pipeline so it is possible for two instructions to need it at the same time, for example single memory unit is used instead of separate instruction memory and data memories. The pipeline design is resolved this hazard by adding extra hardware.
 - **Data Hazards:** These hazards occur when reads and writes of data occur in a different order in the pipeline. Following are the types of data hazard:
 - **RAW:** A Read After Write (RAW) hazard occurs when one instruction reads a location after an earlier instruction writes new data to it but in the pipeline the write occurs after the read so the instruction doing the read gets stale data.
 - **WAR:** A Write After Read (WAR) hazard is the reverse of a RAW. In this type of code, write occurs after a read but the pipeline causes write to happen first.
 - **WAW:** A Write After Write (WAW) hazard is a situation in which two writes occur out of order. It is only considered as a WAW hazard when there is no read in between. If there is, then we have a RAW and WAR hazard to resolve.
 - **Control Hazards:** These hazards occur when a decision needs to be made but the information needed to make the decision is not available yet. A control hazard is considered separately because different techniques can be employed to resolve it.
- The Tomasulo algorithm is a hardware algorithm developed in 1967 by Robert Tomasulo from IBM. This algorithm allows sequential instructions that will be stalled due to certain dependencies to execute non-sequentially execution. It was first

implemented for the IBM System/360 Model 91's floating-point unit. This algorithm utilizes register renaming where scoreboardding resolves Write-after-Write (WAW) and Write-after-Read (WAR) hazards by stalling. Register renaming allows the continual issuing of instructions. The Tomasulo algorithm also uses a Common Data Bus (CDB) on which computed values are broadcast to all the reservation stations that may need it. Tomasulo algorithm is used in decoding the instruction, allocating a RS, renaming source register, renaming destination register, reading register file and dispatching the decoded and renamed registers. Following approaches are used for the implementation of Tomasulo's algorithm while register renaming:

- Instructions are issued sequentially so that the effects of a sequence of instructions such as exceptions raised by these instructions occur in the same order as they will in a non-pipelined processor.
- All general-purpose and reservation station registers hold either real or virtual values. If a real value is unavailable to a destination register during the issue stage, then a virtual value is initially used. The functional unit that is computing the real value is assigned as the virtual value. The virtual register values are converted to real values as soon as the designated functional unit completes its computation.
- Functional units use reservation stations with multiple slots (areas). Each slot holds information needed to execute a single instruction, including the operation and the operands. The functional unit begins processing when it is free and when all source operands needed for an instruction are real.

In computer processing, if a processor changes the value of an operand and at a subsequent time, fetches the operand and obtains the old rather than the new value of the operand, then it is called stale data. Hazards are used to stall the pipeline. A cache miss stalls all the instructions on pipeline both before and after the instruction causing the miss. Eliminating a hazard requires some instructions in the pipeline that is to be allowed to proceed while other instructions are delayed. When the instruction is stalled, all the instructions issued later than the stalled instruction are also stalled. Instructions issued earlier than the stalled instruction must continue since the hazard will never clear. A hazard causes pipeline insert bubbles. Examples of early-out-of-order designs included the MIPS R10000, Alpha 21264 and to some extent the entire POWER/PowerPC line with their reservation stations. In these days, almost all high performance processors are out-of-order designs, with the notable exceptions of UltraSPARC-II/IV and POWER6. Most low-power processors, such as ARM11, Cortex-A8 and Atom are in-order designs because out-of-order logic consumes a lot of power for a relatively small performance gain. Register renaming is a widely used technique to remove false data dependencies (WAR and WAW dependencies) between register operands of subsequent instructions in straight-line code. The principle of register renaming is straightforward. If the processor encounters an instruction which addresses a destination register, it writes the result of the instruction temporarily into a dynamically allocated rename buffer rather than into the specified destination register. The following examples show the WAR dependency:

```
i1: ad ... r2, ...; [... ??(r2) + (...)]
i2: mul r2, ...; [r2 ??(..) * (...)]
```

NOTES

The destination register of i2 (r2) will be renamed, for example r33. Then after the renaming of r2, instruction i2 becomes:

i2': mul r33,...,.; [r33 ?r(..) *(..)]

NOTES

The result will be written into r33 instead of into r2 as per above instruction. This resolves the previous WAR dependency between i1 and i2. In subsequent instructions, however, references to source registers must be redirected to the rename buffers allocated to them. Figure 3.30 illustrates the state transition diagram of the rename registers. During initialization the processor sets all rename registers into the 'available' state. When the processor allocates a rename register to an issued instruction, the state of the allocated register will be changed to 'allocated, not valid' and its valid bit will be reset. When this instruction is finished, the newly produced result is written into the rename register, and the state of the associated rename register becomes 'allocated, valid'. Finally, while the instruction completes, the result held temporarily in the rename register is written into the specified architectural register or into the addressed memory location. Thus, the allocated rename register can be reclaimed. Its state is then changed to 'available'. In this case, a recovery procedure is needed, and the state of the concerned rename registers will be changed from the 'allocated, not valid' or 'allocated, valid' state to the 'available' state and the corresponding mappings between architectural and rename registers will be deleted.

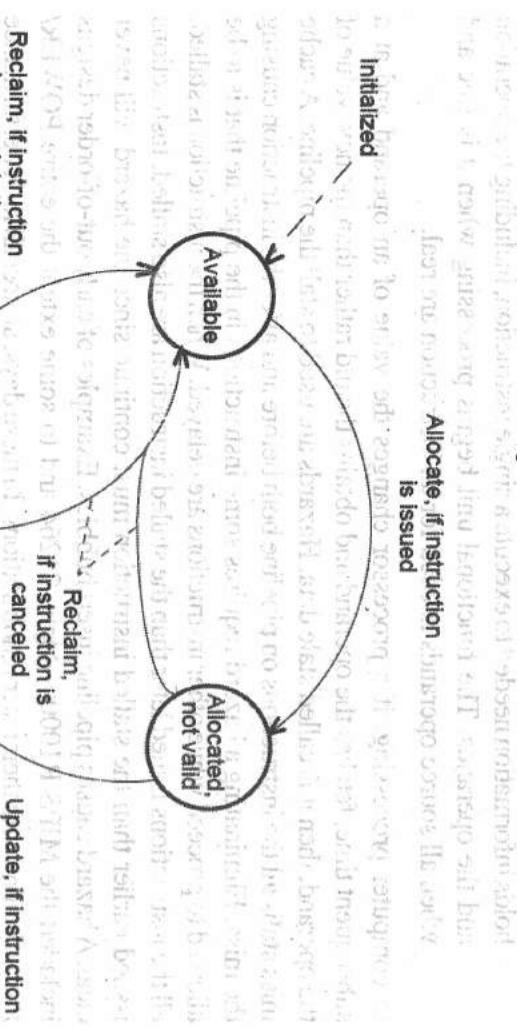


Fig. 3.30 State Transition Diagram of the Rename Registers

Design Space of Register Renaming Techniques

The design space of register renaming has four main dimensions: the scope of register renaming, the layout of the rename registers, the method of register mapping and the rename rate, as indicated in Figure 3.31.

in which stages can see Register renaming techniques used to keep architectural registers from being modified while they are being renamed. Renamed buffers keep register results temporarily until instructions complete. While using renaming the processor maps referenced architectural registers to given rename buffers. Destination registers occurring in instructions are then mapped to particular rename buffers during instruction issue, and the established mappings are maintained until their invalidation. There are two possibilities for keeping track of the actual mapping of particular architectural registers to allocated rename buffers. The processor can use a mapping table for this or can track the actual register mapping within the rename buffers themselves. As its name suggests, the rename rate stands for the maximum number of renames that a processor is able to perform in a cycle. Basically, the processor should be able to rename all instructions issued in the same cycle in order to avoid performance degradation. Thus, the rename rate should equal the issue rate.

register renaming

rename buffers

register mapping

NOTES

Fig. 3.31 Design Space of Register Renaming Techniques

The scope of register renaming indicates how extensively the processor makes use of renaming. Rename buffers establish the actual framework for renaming. Rename buffers keep register results temporarily until instructions complete. While using renaming the processor maps referenced architectural registers to given rename buffers. Destination registers occurring in instructions are then mapped to particular rename buffers during instruction issue, and the established mappings are maintained until their invalidation. There are two possibilities for keeping track of the actual mapping of particular architectural registers to allocated rename buffers. The processor can use a mapping table for this or can track the actual register mapping within the rename buffers themselves. As its name suggests, the rename rate stands for the maximum number of renames that a processor is able to perform in a cycle. Basically, the processor should be able to rename all instructions issued in the same cycle in order to avoid performance degradation. Thus, the rename rate should equal the issue rate.

Branch Prediction

Branch predictor is a digital circuit that determines a decision which way a branch will go, for example an ‘if-then-else’ structure decides the path of instruction flow. The purpose of the branch predictor is to improve the flow in the instruction pipeline. Branch predictors are crucial in pipelined microprocessors for achieving high performance. Two-way branching is usually implemented with a conditional jump instruction. Branch prediction increases the number of instructions available for the scheduler to issue and also instruction level parallelism. It allows useful work to be completed while waiting for the branch to resolve. When a branch is predicted, no speculative state may commit. Squash instructions in the pipeline must not allow stores in the pipeline to occur. It cannot allow stores which would not have happened to commit. It handles exceptions appropriately. Branch prediction research focusses on improving the performance of pipelined microprocessors by accurately predicting ahead of time whether or not a change in control flow will occur. Changes in control flow or branches affect processor performance because many processor cycles must be wasted flushing the pipeline and reading in the correct instructions when programs do not behave as the processor expects them to. Imagine that a branch instruction has moved through the fetch and decode stages and is now being executed. This execution stage is the first time that the processor knows whether or not the branch will be taken. In general the result of this decision is based on a compare between two other data elements, for example ‘IF X > Y THEN...’. The problem arises because until

NOTES

this comparison occurs, the processor does not know the next correct instruction to execute. The stages prior to the execution cycle have already begun speculatively processing instructions that follow the branch, yet if the branch is taken, these are not the correct instructions. Therefore, all the stages before the execution cycle must be flushed and instruction fetch must proceed from the target location of the taken branch. This flushing of the pipeline wastes many cycles of execution time, thereby decreasing the performance of the processor. In an effort to save these wasted cycles, processor designers try to predict the direction of each branch instruction to execute next. If the prediction is incorrect, however, the pipeline must be flushed, and the correct instruction read into the pipeline. This incorrect prediction is known as a misprediction. One of the simplest branch predictors which track the behaviour of individual branches is the bimodal predictor. Bimodal branch predictors take advantage of the fact that a branch can either be taken or not taken. This bimodal distribution of branch behavior allows branch predictor designers to represent a given branch occurrence with a single bit. Following are the two common types of branch prediction:

- **Local Branch Prediction:** A local branch predictor has a separate history buffer for each conditional jump instruction. It may use a two-level adaptive predictor. The history buffer is separate for each conditional jump instruction while the pattern history table may be separate as well or it may be shared between all conditional jumps. The Intel Pentium MMX, Pentium II and Pentium III have local branch predictors with a local 4-bit history and a local pattern history table with 16 entries for each conditional jump.

• **Global Branch Prediction:** A global branch predictor does not keep a separate history record for each conditional jump. Instead it keeps a shared history of all conditional jumps. The advantage of a shared history is that any correlation between different conditional jumps is utilized in the prediction making. The disadvantage is that the history is diluted by irrelevant information in case the different conditional jumps are uncorrelated, and that the history buffer may not include any bits from the same branch in case there are many other branches in between. It may use a two-level adaptive predictor. The history buffer must be longer in order to make a good prediction. The size of the pattern history table grows exponentially with the size of the history buffer. Hence, the big pattern history table must be shared between all conditional jumps.

Bimodal branch prediction is the simplest 2-bit counter based dynamic prediction scheme. The branch history table is indexed by the low order address bits in the program counter. Correlated branch prediction schemes include common correlation, gselect, global and local. Since the bimodal scheme takes advantage of the bimodal distribution of branch behaviour, it does not perform well when branches have strong dynamic behaviour. Correlated prediction schemes are designed to take advantage of relationship between different branch instructions certain repetitive branch pattern of several consecutive branches. Figure 3.32 illustrates the predictor table which includes branch history.

is being decided which of the two branches to take. The history of recent branches is stored in a shift register. The predictor takes the PC as input and outputs the branch prediction.

NOTES

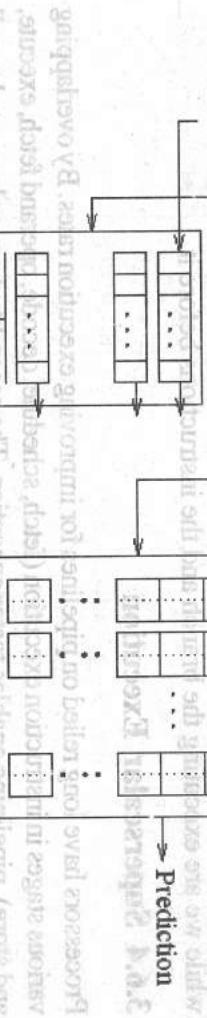


Fig. 3.32 Branch History and Predictor Table

Branch History Table

Branch Predictor Table

One correlation based predictor uses two branch history tables. The first table records the history of recent branches global history. Each entry is implemented using a shift register. The second table records the branch history for each branch. It is organized as a matrix with rows and columns. Each entry is a 2-bit counter. The pc determines which shift register in the first table and which row of the 2-bit counters of the second table should be used. The chosen global shift register indexes the appropriate counter from the selected row of counters. Prediction is made based the selected counter. The selected shift register and the 2-bit counter will be updated afterwards accordingly. Sharing index scheme is referred to as gshare. It was proposed by McFarling. This scheme is similar to the bimodal scheme. It XORs a j bits global history shift register with i bits of the pc before indexing the counter table. Different dynamic schemes use different branch history information. Many schemes work well on one type of programs and do not work well on another type of programs. The selective scheme uses two different predictors. Each of two predictors makes prediction independently. A third table is used to track the performance of the two subpredictors and arbitrates which prediction should be used as the final prediction. Selective scheme can perform well on different types of programs. The implementation costs of different schemes are shown in the following table. In the table, i is the number of pc bits for indexing the counter table row, j is the number of pc bits for indexing the shift register table and k is the number of bits of the shift register (refer Figure 3.32). Table 3.14 summarizes the dynamic predictor implementation cost.

Table 3.14 Dynamic Predictor Implementation Cost

Scheme Name	i	j	k	Buffer Size
bimodal	variable	Not Available	Not Available	$2 * 2^j$
correlation	variable	1	2	$1 + 2 * 4 * 2^i$
gselect	variable	1	variable	$k + 2 * 2^{j(i+k)}$
global	1	1	variable	$k + 2 * 2^k$
local	variable	variable	variable	$k * 2^j + 2 * 2^{(i+k)}$
gshare	variable	Not Available	variable	$k + 2 * 2^j$

NOTES

Branch prediction is a common hardware technique. In the scheme, the branch predictor is used to maintain a table of two-bit entries. Low-order bits of a branch's address provide the index into this table. This two-bit prediction scheme correctly handles loops whose branch is usually taken but not always. Branch prediction is often used to keep a pipeline full where fetch and decode instructions are fetched after a branch while we are executing the branch and the instructions before it.

3.6.4 Superscalar Execution

Processors have long relied on pipelines for improving execution rates. By overlapping various stages in instruction execution (fetch, schedule, decode, operand fetch, execute, and store), pipelining enables faster execution. The assembly-line analogy works well for understanding pipelines. For example, if the assembly of a car takes 100 time units which can be broken into 10 pipelined stages of 10 units each then a single assembly line can produce a car every 10 time units. This represents a 10-fold speedup over producing cars entirely serially, one after the other. It is also evident from this example that to increase the speed of a single pipeline one would break down the tasks into smaller and smaller units thus lengthening the pipeline and increasing overlap in execution. In the context of processors, this enables faster clock rates since the tasks are now smaller. For example, the Pentium 4, which operates at 2.0GHz, has a 20 stage pipeline. The speed of a single pipeline is ultimately limited by the largest atomic task in the pipeline. Long instruction pipelines therefore need effective techniques for predicting branch destinations so that pipelines can be speculatively filled. The penalty of a misprediction increases as the pipelines become deeper since a larger number of instructions need to be flushed. These factors place limitations on the depth of a processor pipeline and the resulting performance gains. An obvious way to improve instruction execution rate beyond this level is to use multiple pipelines. During each clock cycle, multiple instructions are piped into the processor in parallel. As their name suggests, superscalar machines were originally developed as an alternative to vector machines. A superscalar machine of degree n can issue n instructions per cycle. Superscalar execution of instructions is illustrated in Figure 3.33.

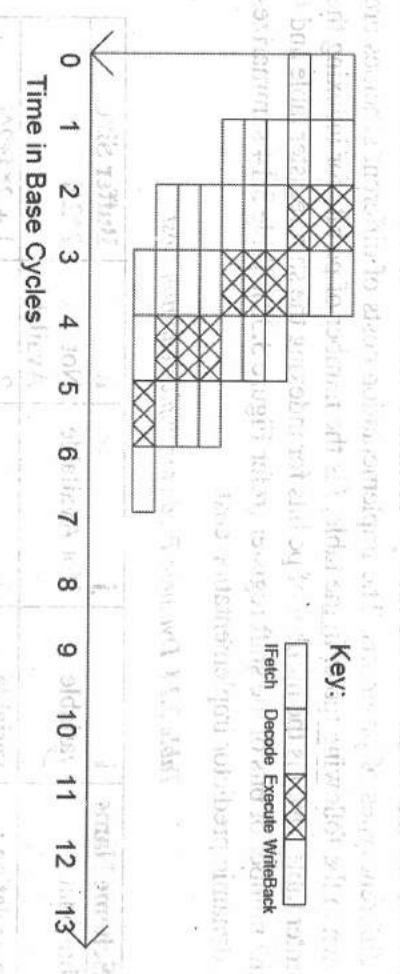


Fig. 3.33 Execution in A Superscalar Machine Containing $n=3$

A superscalar processor of degree n can issue n instructions per cycle. In this sense, the base scalar processor, implemented either in RISC or CISC, has $n=1$. In order to fully utilize a superscalar processor of degree n , n instructions must be executable in parallel. This situation may not be true in all clock cycles. In that case, some of the

pipelines may be stalling in a wait state. In a superscalar processor, the simple operation latency should require only one cycle, as in the base scalar processor. In order to utilize a superscalar machine of degree n , there must be n instructions executable in parallel at all times. If an instruction-level parallelism of n is not available, stalls and dead time will result where instructions are forced to wait for the results of prior instructions. Formalizing a superscalar machine according to definitions is as follows:

- Instructions issued per cycle = n .
- Simple operation latency measured in cycles = 1.
- Instruction-level parallelism required to fully utilize = n .

A superscalar machine can attain the same performance as a machine with vector hardware. Consider the operations performed when a vector machine executes a vector load chained into a vector add, with one element loaded and added per cycle. The vector machine performs four operations: load, floating-point add, a fixed-point add to generate the next load address, and a compare and branch to see if we have loaded and added the last vector element. Figure 3.34 illustrates the process of superscalar execution.

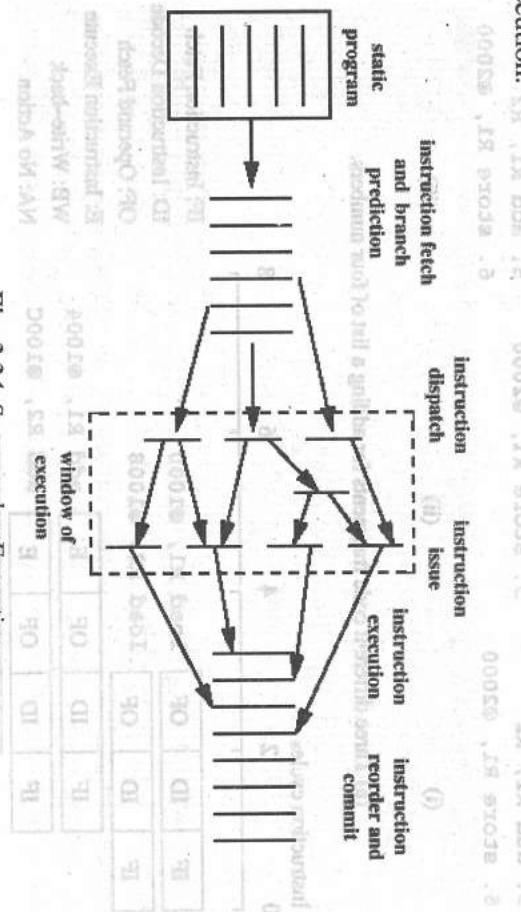


Fig. 3.34 Superscalar Execution

Static program is sent to instruction fetch and branch prediction. This program is segregated to instruction dispatch and instruction issue as per defined instructions in the program. After passing through window of execution, it is sent to instruction reorder and commit. An application begins as a high level language program; it is then compiled into the static machine level program or the program binary. The static program in essence describes a set of executions, each corresponding to a particular set of data that is given to the program. Implicit in the static program is the sequencing model, the order in which the instructions are to be executed. As a static program executes with a specific set of input data, the sequence of executed instructions forms a dynamic instruction stream. As long as instructions to be executed are consecutive, static instructions can be entered into the dynamic sequence simply by incrementing the program counter, which points to the next instruction to be executed. When there is a conditional branch or jump, however, the program counter may be updated to a non-consecutive address. An instruction is said to be control dependent on its preceding dynamic instructions, because the flow of program control must pass through preceding

NOTES

instructions first. A superscalar machine that can issue a fixed-point, floating-point, load and a branch all in one cycle achieves the same effective parallelism. These instructions are executed on multiple functional units. This process can be illustrated with the help of an example.

NOTES

Example of Superscalar Execution

Consider a processor with two pipelines and the ability to simultaneously issue two instructions. These processors are sometimes also referred to as superpipelined processors. The ability of a processor to issue multiple instructions in the same cycle is referred to as superscalar execution. Since the architecture illustrated in Figure 3.35 allows two issues per clock cycle, it is also referred to as two-way superscalar or dual issue execution.

ifT	1. load R1, @1000	1. load R1, @1000	1. load R1, @1000
bbs	2. load R2, @1008	2. add R1, @1004	2. add R1, @1004
bbr	3. add R1, @1004	3. add R1, @1008	3. load R2, @1008
isloc	4. add R2, @100C	4. add R1, @100C	4. add R2, @100C
isloc	5. add R1, R2	5. store R1, @2000	5. add R1, R2
	6. store R1, @2000	6. store R1, @2000	

(a) Three different code fragments for adding a list of four numbers.

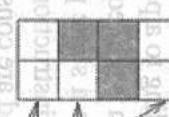


IF	ID	OF	load R1, @1000
IF	ID	OF	load R2, @1008
IF	ID	OF	E
IF	ID	OF	E
IF	ID	NA	E

IF: Instruction Fetch
ID: Instruction Decode
OF: Operand Fetch
E: Instruction Execute
WB: Write-back
NA: No Action

ai	ifT	ifT	ifT	ifT	ifT	ifT	store R1, @2000
ai	ifT	ifT	ifT	ifT	ifT	ifT	store R1, @2000
ai	ifT	ifT	ifT	ifT	ifT	ifT	store R1, @2000
ai	ifT	ifT	ifT	ifT	ifT	ifT	store R1, @2000
ai	ifT	ifT	ifT	ifT	ifT	ifT	store R1, @2000

IF: Instruction Fetch
ID: Instruction Decode
OF: Operand Fetch
E: Instruction Execute
WB: Write-back



Horizontal waste
Vertical waste

Full issue slots

Empty issue slots

(b) Execution schedule for code fragment (i) above.

Consider the execution of the first code fragment in Figure 3.35 for adding four numbers. The first and second instructions are independent and therefore can be issued

Fig. 3.35 Example of a Two-Way Superscalar Execution of Instructions

concurrently. This approach is illustrated in the simultaneous issue of the instructions load R1, @1000 and load R2, @1008 at $t = 0$. The instructions are fetched, decoded, and the operands are fetched. The next two instructions, such as add R1, @1004 and add R2, @100C are independent although they must be executed after the first two instructions. Consequently, they can be issued concurrently at $t = 1$ since the processors are pipelined. These instructions terminate at $t = 5$. The next two instructions, add R1, R2 and store R1, @2000 cannot be executed concurrently since the result of the former (contents of register R1) is used by the latter. Therefore, only the add instruction is issued at $t = 2$ and the store instruction at $t = 3$. The instruction ‘add R1, R2’ can be executed only after the previous two instructions have been executed. The instruction schedule is illustrated in Figure 3.35 (b). The schedule assumes that each memory access takes a single cycle. First, instructions in a program may be related to each other as illustrated in Figure 3.35. The results of an instruction may be required for subsequent instructions. This is referred to as *true data dependency*. For example, consider the second code fragment in Figure 3.35 for adding four numbers. There is a true data dependency between load R1, @1000 and add R1, @1004, and similarly between subsequent instructions. Dependencies of this type must be resolved before simultaneous issue of instructions. This has two implications. First, since the resolution is done at runtime, it must be supported in hardware. The complexity of this hardware can be high. Second, the amount of instruction level parallelism in a program is often limited and is a function of coding technique. In the second code fragment, there can be no simultaneous issue, leading to poor resource utilization. The three code fragments in Figure 3.35 (a) also illustrate that in many cases it is possible to extract more parallelism by reordering the instructions and by altering the code. Notice that in this example the code reorganization corresponds to exposing parallelism in a form that can be used by the instruction issue mechanism. Another source of dependency between instructions results from the finite resources shared by various pipelines. As an example, consider the co-scheduling of two floating point operations on a dual issue machine with a single floating point unit. Although there might be no data dependencies between the instructions, they cannot be scheduled together since both need the floating point unit. This form of dependency in which two instructions compete for a single processor resource is referred to as *resource dependency*. The flow of control through a program enforces a third form of dependency between instructions. Consider the execution of a conditional branch instruction. Since the branch destination is known only at the point of execution, scheduling instructions *a priori* across branches may lead to errors. These dependencies are referred to as *branch dependencies* or *procedural dependencies* and are typically handled by speculatively scheduling across branches and rolling back in case of errors. Studies of typical traces have shown that on average, a branch instruction is encountered between every five to six instructions. Therefore, just as in populating instruction pipelines, accurate branch prediction is critical for efficient superscalar execution. The ability of a processor to detect and schedule concurrent instructions is critical to superscalar performance. For instance, consider the third code fragment in Figure 3.35 which also computes the sum of four numbers. The reader will note that this is merely a semantically equivalent reordering of the first code fragment. However, in this case, there is a data dependency between the first two instructions are load R1, @1000 and add R1, @1004. Therefore, these instructions cannot be issued simultaneously. However, if the processor had the ability to look ahead, it would realize that it is possible to schedule the third instruction load

NOTES

R2, @1008 with the first instruction. In the next issue cycle, instructions two and four can be scheduled, and so on. In this way, the same execution schedule can be derived for the first and third code fragments. However, the processor needs the ability to issue instructions '*out-of-order*' to accomplish desired reordering. The parallelism available in '*in-order*' issue of instructions can be highly limited as illustrated by this example. Most current microprocessors are capable of out-of-order issue and completion. This model, also referred to as *dynamic instruction issue*, exploits maximum instruction level parallelism. The processor uses a window of instructions from which it selects instructions for simultaneous issue. This window corresponds to the look-ahead of the scheduler. The performance of superscalar architectures is limited by the available instruction level parallelism. Consider the example in Figure 3.35. For simplicity of discussion, let us ignore the pipelining aspects of the example and focus on the execution aspects of the program. Assuming two execution units (multiply-add units), the Figure 3.35 illustrates that there are several zero-issue cycles (cycles in which the floating point unit is idle). These are essentially wasted cycles from the point of view of the execution unit. If, during a particular cycle, no instructions are issued on the execution units, it is referred to as *vertical waste*; if only part of the execution units is used during a cycle, it is termed *horizontal waste*. In the example, we have two cycles of vertical waste and one cycle with horizontal waste. In all, only three of the eight available cycles are used for computation. This implies that the code fragment will yield no more than three-eighths of the peak rated FLoating point Operations per Second or FLOPS count of the processor. Often, due to limited parallelism, resource dependencies, or the inability of a processor to extract parallelism, the resources of superscalar processors are heavily underutilized. Current microprocessors typically support up to four-issue superscalar execution.

3.6.5 Superscalar Implementation

For a superscalar implementation to sustain the execution of multiple instructions per cycle, the fetch phase must be able to fetch multiple instructions per cycle from the cache memory. To support this high instruction fetch bandwidth, it has become almost mandatory to separate the instruction cache from the data cache. A typical superscalar processor fetches and decodes the incoming instruction stream several instructions at a time. As part of the instruction fetching process, the outcomes of conditional branch instructions are usually predicted in advance to ensure an uninterrupted stream of instructions. The incoming instruction stream is then analysed for data dependences and instructions are distributed to functional units, often according to instruction type. Next, instructions are initiated for execution in parallel, based primarily on the availability of operand data, rather than their original program sequence. This important feature, present in many superscalar implementations, is referred to as dynamic instruction scheduling. Upon completion, instruction results are re-sequenced so that they can be used to update the process state in the correct (original) program order in the event that an interrupt condition occurs. Because individual instructions are the entities being executed in parallel, superscalar processors exploit what is referred to as instruction level parallelism. CISC or a RISC scalar processor can be improved with a superscalar or vector architecture. In a superscalar processor, multiple instruction pipelines are used. This implies that multiple instructions are issued per cycle and multiple results are generated per cycle. A vector processor executes vector instructions

NOTES

on arrays of data. Thus, each instruction involves a string of repeated operations which are essential for pipelining with one result per cycle. Following factors are used in superscalar processors:

Superscalar Processors: Superscalar processors are designed to exploit more instruction-level parallelism in user programs. Only independent instructions can be executed in parallel without causing a wait state. The amount of instruction-level parallelism varies widely depending on the type of code being executed. It has been observed that the average value is around 2 for code without loop unrolling. Therefore, for these codes there is not much benefit gained from building a machine that can issue more than three instructions per cycle.⁵⁵

Pipelining in Superscalar Processors: Superscalar processors were originally developed as an alternative to vector processors. Multiple instruction pipelines are used. The instruction cache supplies multiple instructions per fetch. However, the actual number is constrained by data dependences and resource conflicts among instructions that are simultaneously decoded. Multiple functional units are built into the integer unit and into the floating-point unit. Multiple data buses exist among the functional units. In theory, all functional units can be simultaneously used if conflicts and dependences do not exist among them during a given cycle.

Representative Superscalar Processors: A number of commercially available processors have been implemented with the superscalar architecture including the IBM RS/6000, DEC 21064, and Intel i960CA processors. Due to the reduced CPI and higher clock rates used, most superscalar processors out-perform scalar processors. The maximum number of instructions issued per cycle ranges from two to five in these four superscalar processors. Typically, the register files in the IU or Integer Unit and FPU or Floating-Point Unit each have 32 registers. Most superscalar processors implement both the IU and the FPU on the same chip. The superscalar degree is low due to limited instruction parallelism that can be exploited in ordinary programs. Besides the register files, reservation stations and reorder buffers can be used to establish instruction windows. The purpose is to support instruction look ahead and internal data forwarding, which are needed to schedule multiple instructions through the multiple pipelines simultaneously.

3.6.6 Instruction Level Parallelism and Machine Parallelism

An instruction level parallelism and machine parallelism are the two techniques are shown to be equivalent ways of exploiting instruction level parallelism. The average degree of superpipelining metric is introduced to account for the non-unit operation latencies present in most machines. A first order estimate is based on the average degree of superpipelining. A second order model corrects for the effects of non-uniformities in instruction-level and machine parallelism for different machine pipelines: CRAY-1, Multifitan and a dual issue superscalar machine. The instruction level parallelism is measured primarily within basic blocks except for limited amounts of loop unrolling in the numeric benchmarks. Inter-block instruction level parallelism depends on the particular implementation of branch prediction, trace scheduling or software pipelining used to make inter-block parallelism accessible. Once the inter-block instruction level parallelism made available by these techniques is quantified, the performance estimation techniques developed in this paper are applicable based on the Instruction level parallelism present.

NOTES

NOTES

Machine parallelism is based on the concepts of superscalar and superpipelined machines. Superscalar machines can issue several instructions per cycle. Superpipelined machines can issue only one instruction per cycle but they have cycle times shorter than the latencies of their functional units. Machine parallelism interface allows us to specify details about the pipeline, functional units, cache and register set. Machine level parallelism includes the following types of parallelism:

- **Data Parallelism:** Data parallelism refers to program loops which focus on distributing the data across different computing nodes to be processed in parallel. Parallelizing loops often leads to similar operation sequences or functions being performed on elements of a large data structure.
- **Task Parallelism:** Task parallelism is the characteristic of a parallel program that completely different calculations can be performed on either the same or different sets of data.

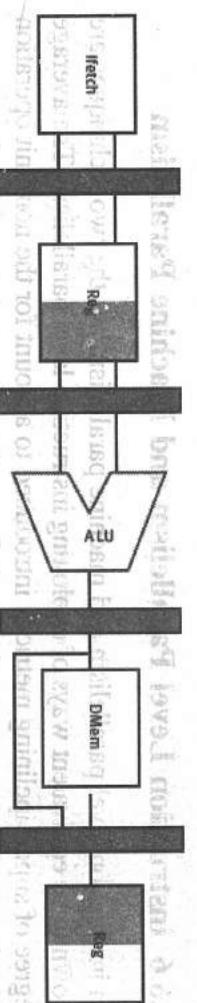
Superscalar machines may have an upper limit on the number of instructions that may be issued in the same cycle, independent of the availability of functional units. Following are the ways to execute instructions in parallel:

- Operation latency is the time (in cycles) until the result of an instruction is available for use as an operand in a subsequent instruction. For example, if the result of an 'Add' instruction can be used as an operand of an instruction that is issued in the cycle after the Add is issued, we say that the Add has an operation latency of one.

- Simple operations are executed by the machine. Operations, such as integer add, logical ops, loads, stores, branches and even floating-point addition and multiplication are simple operations.

- Instruction class represents a group of instructions all issued to the same type of functional unit.

The language system then optimizes the code, allocates registers, and schedules the instructions for the pipeline, all according to this specification. The simulator executes the program according to the same specification.



Check Your Progress

14. What is a latch?
15. Name some of the superscalar microprocessors.
16. Write one characteristic of Alpha 21164.
17. Who developed Tomasulo algorithm?
18. Why is register renaming used in superscalar technique?
19. Name the factors on which inter-block instruction level parallelism depends.
20. What does machine parallelism interface allow?

Machine parallelism of a processor is a measure of the ability of the processor to take advantage of the ILP of the program which is determined by the number of instructions that can be fetched and executed at the same time. A perfect machine with infinite machine parallelism can achieve the ILP of a program. To achieve high performance, need both ILP and machine parallelism. Parallel instructions are a set of instructions that do not depend on each other to be executed. Figure 3.36 illustrates the structure of machine parallelism which contains pipeline execution representation, such as ifetch (fetch instructions), reg (register), etc.

3.7 SUMMARY

Central Processing Unit

- 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. An instruction guides the CPU to perform work accordingly.
- 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.
- 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 in the instruction in the operand field.
- Address-field provides operands on which operation is to be performed or provides the addresses of CPU register or main memory addresses which store the operands.
- A program counter keeps track of the instructions in the program stored in the memory. The program counter stores the address of the command to be run next and is increased each time an instruction is fetched from memory.
- The command identifies a register in the CPU whose contents offer the address of the memory location where the operand is stored, i.e. the selected register comprises the address of the operand rather than the operand itself. In this mode, the register acts as the memory address register.
- 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.
- 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 0 and 99 separated by spaces without formatted text and symbols.
- The control unit not only plays a major role in transmitting data from a device to the CPU and vice versa but also plays a significant role in the functioning of the CPU. It actually does not process the data but manages and coordinates the entire computer system including the input and the output devices.
- The ALU is responsible for arithmetic and logic operations. This means that when the control unit encounters an instruction that involves an arithmetic operation (add, subtract, multiply, divide) or a logic operation (equal to, less than, greater than), it passes control to the ALU, which has the necessary circuitry to carry out these arithmetic and logic operations.
- Primary storage memory is the main memory of the computer which communicates directly with the processor. This memory is large in size and fast, but not as fast as the internal memory of the processor. It is actually a couple of integrated chips mounted on a printed circuit board which are plugged directly on the motherboard.
- Secondary storage memory stores all the system software and application programs and is basically used for data backups. It is much larger in size and

NOTES

slower than the primary storage memory. Hard disk drives, floppy disk drives and flash drives are a few examples of secondary storage memory.

- 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.

NOTES

- 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.
- An instruction pipeline reads the consecutive instruction from the memory when the previous instruction is executing in other segments. In this way, we can overlap the instructions fetch and execute phases and perform operations simultaneously.
- If an instruction in a 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.
- RISC, which stands for Reduced Instruction Set Computer, gives a new generation of faster and cheaper machines. The basic idea behind RISC is to use a small set of instructions with simple constructs so that it can execute them much faster within the CPU.
- The SPARC is a RISC oriented design. The SPARC uses the concept of 'register Windows' in order to eliminate the load and stores to a stack associated with procedure calls.
- The Intel 860 is a 32-bit processor built with Harvard architecture. The bus to the instruction cache is 32 bits wide and the bus to the data cache is 128 bits wide making possible to access four words in parallel.
- Transputer is a 32-bit processor with a non Harvard architecture. The Immos implementation does not provide a cache although there is an internal on-chip memory which must be explicitly addressed.
- Instruction Level Parallelism or ILP is a measure of the number of instructions that can be performed during a single clock cycle. Parallel instructions are a set of instructions that do not depend on each other to be executed.
- Machine parallelism is based on the concepts of superscalar and superpipelined machines. Superscalar machines can issue several instructions per cycle. Superpipelined machines can issue only one instruction per cycle but they have cycle times shorter than the latencies of their functional units.

3.8 KEY TERMS

- **Instruction:** A command given to a computer to perform a specified operation on some given data
- **Instruction sequencing:** A method by which instructions are selected for execution

- **Processor's instruction set:** A collection of different instructions that the processor can execute
- **Memory operand:** An address in memory used for those data which are to be processed
- **Exchange:** An operation which is used to swap contents of source and destination in instruction set
- **Internal processor memory:** A small set of high speed registers placed inside a processor

NOTES

- **Control word:** A group of binary bits assigned to perform a specified operation
- **Resource conflicts:** A situation in which when two segments access the same memory location at the same time
- **RISC:** A type of microprocessor that is utilized high optimized set of instructions
- **Intel 860:** A 32-bit processor built with Harvard architecture
- **Register windows:** Hardware oriented method to optimize register allocation
- **Transputer:** A 32-bit processor with a non Harvard architecture
- **Latch:** A circuit that has two stable states and can be used to store state information

3.9 ANSWERS TO 'CHECK YOUR PROGRESS'

1. An instruction is a command given to a computer to perform a specified operation on some given data.
2. The sequence of operations performed by the CPU in processing an instruction is known as an instruction cycle.
3. Address-field provides operands on which operation is to be preformed or provides the addresses of CPU register or main memory addresses which store the operands.
4. The address of an operand is known as effective address.
5. 'Complement Accumulator' is an implied mode instruction because the operand in the accumulator register is implied in the definition of the instruction.
6. 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.
7. 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.
8. Internal processor memory is a small set of high speed registers placed inside a processor and are used for storing temporary data while processing.
9. Hard disk drives, floppy disk drives and flash drives are a few examples of secondary storage memory.
10. An instruction pipeline operates on a stream of instructions by overlapping the fetch, decode and execute phases of the instruction cycle.

- The basic idea behind RISC is to use a small set of instructions with simple constructs.

NOTES

- John Cocke of IBM, New York introduced RISC concept in 1974.
- The 'RISC core' contains thirty two 32-bit registers and one ALU.
- A latch is a circuit that has two stable states and can be used to store state information.
- The current superscalar microprocessors are MIPS R10000, DEC Alpha 21164 and AMD K5.
- The Alpha 21164 is an example of a simple superscalar processor that avails the advantages of dynamic instruction scheduling with the help of high clock rate.
- The Tomasulo algorithm is a hardware algorithm developed in 1967 by Robert Tomasulo from IBM.
- Register renaming is a widely used technique to remove false data dependencies (WAR and WAW dependencies) between register operands of subsequent instructions in straight-line code.
- Inter-block instruction level parallelism depends on the particular implementation of branch prediction, trace scheduling or software pipelining used to make interblock parallelism accessible.
- Machine parallelism interface allows us to specify details about the pipeline, functional units, cache and register set.

3.10 QUESTIONS AND EXERCISES

Short-Answer Questions

- What is an opcode?
- Write the steps which are required to execute an instruction.
- What are the merits and demerits of direct addressing mode?
- What is index addressing mode?
- Write the function of control unit.
- Name the stages involved in an instruction cycle.
- What are resource conflicts?
- Write the major characteristics of RISC processor.
- What is shelving?
- Name the types of data hazards.
- What do 'correlated branch prediction schemes' include in branch prediction?

Long-Answer Questions

- What is an instruction set? Explain with the help of suitable examples.
- Explain the types of operations used in an instruction set.

3. Discuss the addressing modes with the help of examples.
4. Describe the instruction formats with the help of illustrations.
5. Explain the structure of CPU with the help of diagram.
6. Discuss the functions of CPU with the help of examples.
7. Explain four-segment instruction pipeline with the help of examples.
8. Discuss the RISC microprocessors with the help of examples.
9. Discuss the current superscalar microprocessors with the help of illustrations and examples.
10. Describe register renaming with the help of suitable example.
11. Explain the concept of branch prediction with the help of examples.

NOTES

- Tanenbaum, Andrew S. 1999. *Structured Computer Organization*, 4th edition. New Jersey: Prentice-Hall Inc.
- Wilkinson. 1996. *Computer Architecture: Design and Performance*, 2nd edition. Hertfordshire: Prentice-Hall.
- Mano, M. Morris. 1993. *Computer System Architecture*, 3th edition. New Jersey: Prentice-Hall Inc.
- Stallings, William. 2006. *Computer Organization and Architecture*, 7th edition. New Jersey: Prentice-Hall Inc.
- Hamacher, V.C., Z.G. Vranesic and S.G. Zaky. 2002. *Computer Organization*, 5th edition. New York: McGraw-Hill International Edition.
- Conte, T.M. and C.E. Grimaldi. 1995. *Fast Simulation of Computer Architecture*. Boston: Kluwer Academic Publishers.
- Gilmore, M. 1996. *Microprocessors: Principles and Applications*, 2nd edition. New York: McGraw-Hill.

UNIT 4 CONTROL UNIT

Control Unit

Structure

an organism consisting of sub-cellular components.

NOTES

- Expresses the major phases of processing.
- Details about the procedure.

4.0 Introduction

4.1 Unit Objectives

4.2 Control Unit Operation: Basic Concepts

4.2.1 Functions of Control Unit

4.2.2 Control of the Processor

4.2.3 Hardwired Implementation

4.2.4 Microprogrammed Control

4.2.5 Microinstructions

4.3 Microoperation

4.3.1 Data Manipulation Instructions

4.4 Summary

4.5 Key Terms

4.6 Answers to 'Check Your Progress'

4.7 Questions and Exercises

4.8 Further Reading

4.0 INTRODUCTION

In the previous unit, you learnt about the central processing unit. In this unit, you will learn about control unit. The control unit controls the instructions. Hardwired control units are constructed using digital circuits and once formed cannot be changed. These control units are implemented alongwith sequential logic units. They feature a finite number of gates that can act as a generator of specific results based on the instructions that are used to invoke those responses. Outputs of the controller are organized in microinstructions and they can be easily replaced. You will also learn about microprogrammed control. A microprogrammed control unit itself decodes and executes instructions to execute microprograms. Finally, you will learn about microoperation. The operations that are executed on data stored in registers are called microoperations. A microoperation is an elementary operation performed on the information stored in one or more registers. In the central processing unit, microoperations are comprehensive low level instructions specifically used in some designs to implement complex machine instructions. The most common microoperations are register transfer microoperation, arithmetic microoperation, logic microoperation and shift microoperation. Register transfer is used to transfer binary information from one register to another. Arithmetic microoperation performs arithmetic operations on numeric data stored in registers. Logic microoperation performs bit manipulation operations on non-numeric data stored in registers. Shift microoperation performs shift operations on data stored in registers.

4.1 UNIT OBJECTIVES

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

- Understand the various control unit operations
- Explain the functions of control unit

- Discuss the process low hardwired are implemented
- Describe what microprogrammed control is
- Explain the significance of microoperation
- Discuss the various types of data manipulation instructions

NOTES

4.2 CONTROL UNIT OPERATION: BASIC CONCEPTS

A Control Unit or CU in general is a central part of the machinery that controls its operation. It is specifically used in the area of computer design. The control unit coordinates the input and output devices of a computer system. It fetches the code of all of the instructions in the microprograms. It directs the operation of the other units by providing timing and control signals. Figure 4.1 illustrates the structure of control unit.

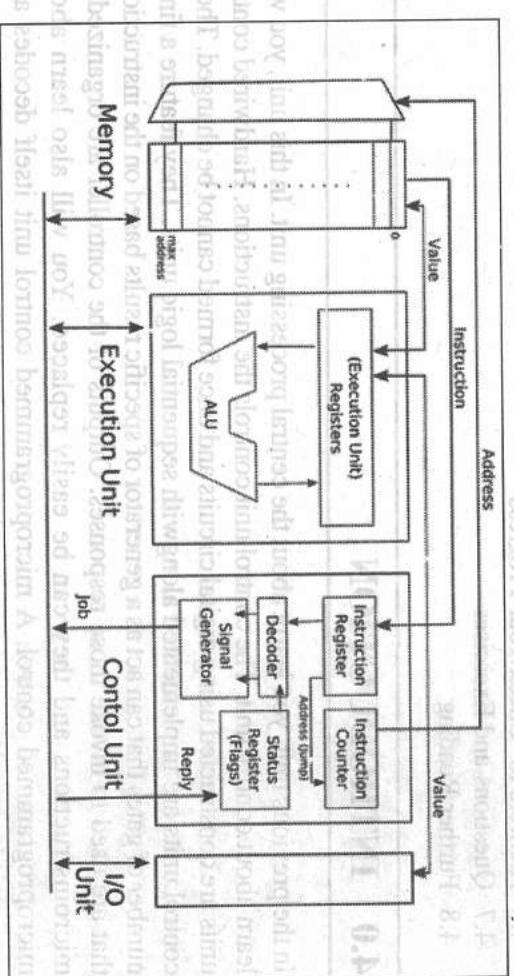


Fig. 4.1 Structure of Control Unit

The control unit interprets instructions of the program and controls the following parts of the processor:

1. **Instruction Register:** The function of instruction register is to store actual instructions which are loaded from the memory.
2. **Instruction Counter:** The function of instruction counter is to store memory address of the following instruction and is incremented automatically after every fetch command.
3. **Status Register:** The status register (flags) keeps the instructions about the result of the recent operations.
4. **Decoder:** The function of decoder is to read the actual instruction and uses the flags as per defined instructions.
5. **Signal Generator:** It is used for the commands that are generated and issued for other parts of the processor.

- **Execution Unit:** The execution unit is the core unit of the processor. Registers, which are very fast memory, components are used to keep the operands that are needed for the shift, arithmetic and logical operations.

Zilog's 8-bit 32032 processor has a 32-bit ALU, a 32-bit multiplier and a 32-bit shifter.

NOTES

- **Memory:** The memory stores both data and instructions of the actual program. Data can be read and written at its specified address.

Motorola 68000 processor has a 32-bit address bus, a 32-bit data bus and a 32-bit control bus.

- **Input/Output Unit:** Input/Output or I/O unit performs all the I/O operations. Input is the process of entering data and programs into the computer system. The task of performing operations like arithmetic and logical operations is called processing. Output is the process of producing results from the data for getting useful information.

The CU primarily executes the following two tasks:

1. Instruction Interpretation
2. Instruction Sequencing

Instruction Interpretation

Instruction Interpretation

Instruction Interpretation

- CU reads instructions from memory using PC.
- It recognizes the instruction type, obtains the necessary operands and then routes to the appropriate functional units of the execution unit.
- It issues necessary signals to perform the desired operation.
- The results are routed to the specific destination.

Instruction Sequencing

The CU determines the address of the next instruction to be executed and loads it to PC. The CU is designed by using one of the following three techniques:

- **Hardwired Control:** It has a processor that generates signals or instructions to be implemented in correct sequence. This method was previously used for physically connecting typical components, such as gates and flip-flops. Z8000 is a 16-bit microprocessor introduced by Zilog in 1979.
- **Microprogramming:** This type of CUs include a control ROM for translating the instructions. Intel 8086 is a 16-bit microprogrammed microprocessor.
- **Nanoprogramming:** Typically, in most of the microprogrammed processors, an instruction is fetched from memory which is then interpreted by a microprogram stored in a single Control Memory or CM. Although in some microprogrammed processors the microinstructions are not directly used by the decoders for generating control signals but they are used for accessing second control memory termed as nanoControl Memory or nCM. Thus, there are two levels of control memories, a higher level termed as microControl Memory or μCM which stores the microinstructions and a lower level called nCM which stores the nanoinstructions. In this case the decoder uses these nanoinstructions through μCM for generating control signals. The key advantage of using nanoprogramming is that it reduces total size of required control memory due to its design flexibility. Motorola's 68000, 68020 and 68030 are examples of nanoprogrammed processors.

Width of Registers in a Microprocessor

The number of bits a CPU uses to represent integer numbers is termed as ‘register width’, ‘word size’, ‘bit width’, ‘data path width’ or ‘integer precision’. This number is considered as one of the most important characteristics of a CPU. Registers are generally measured by the number of bits they can hold, for example, an ‘8-bit register’, a ‘16-bit register’ or a ‘32-bit register’. A processor typically contains several types of registers and is classified on the basis of their content or instructions that operate on them.

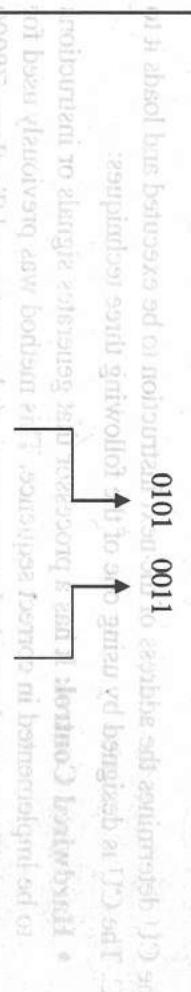
8-Bit Registers

An 8-bit register is 8 bits wide, and can hold 1 byte of data shown in Figure 4.2. Most of the computers nowadays do not display data of an 8-bit register in binary format. Instead, will use a hexadecimal display.



Fig. 4.2 8 Bit Register Model

However, it is useful to separate the 8 bits into two groups of 4 bits each. The left group of 4 bits is called the upper nibble and the right group of 4 bits is called the lower nibble. This is illustrated in Figure 4.3.



Example 4.1: If a register contains the binary number shown in Figure 4.4 then what would appear in the hexadecimal display for that register?

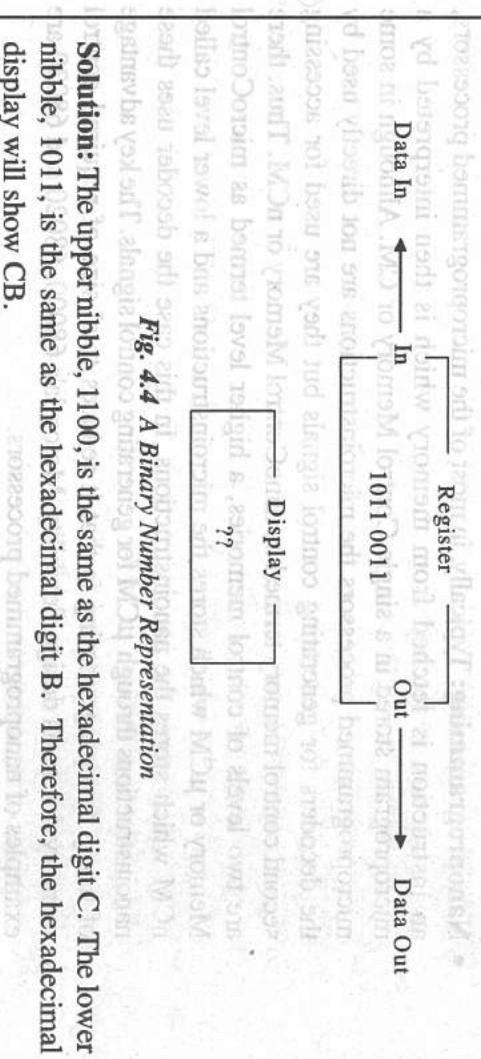


Fig. 4.4 A Binary Number Representation

Solution: The upper nibble, 1100, is the same as the hexadecimal digit C. The lower nibble, 1011, is the same as the hexadecimal digit B. Therefore, the hexadecimal display will show CB.

16-Bit Registers

A 16-bit register is 16 bits wide which is illustrated in Figure 4.5. As you can see, the 16 bits are again separated into groups of 4 bits. Each nibble or group of 4 bits is represented in the display as 1 hexadecimal digit.

NOTES



Fig. 4.5 16-Bit Register Model

Example 4.2: What are the binary contents of the register when the output displayed is as shown in Figure 4.6?

Solution: The far left digit (called the *most significant digit*) B has a binary equivalent of 1011. The F would be represented by 1111. The 3 would be represented by 0011 and the hexadecimal digit C would be represented by 1100 in binary. Putting the four nibbles together produces 1011.1111.0011.1100 which constitutes the binary contents of this register.

4.2.1 Functions of Control Unit

The functions performed by the control unit vary greatly by the internal architecture of the CPU, since the control unit really implements this architecture. On a Complex Instruction Set Computer or CISC processor that executes x86 instructions natively the control unit performs the tasks of fetching, decoding, managing execution and then storing the results. On x86 processor with a RISC (Reduced Instruction Set Computer) core, the control unit has significantly more work to do. It manages the translation of x86 instructions to RISC micro-instructions, manages scheduling the micro-instructions between the various execution units and produces the output. Further, the control unit is divided into various subunits to perform specific tasks, such as a scheduling unit is used to handle scheduling. Following are the functions of control unit:

- 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.

NOTES

- 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.

4.2.2 Control of the Processor

Control address register is a processor register which is used to change or control the general behaviour of a CPU or other digital device. Common tasks performed by control registers include interrupt control, switching the addressing mode, paging control and coprocessor control. Control memory is a Random Access Memory or RAM consisting of addressable storage registers. It is primarily used in minicomputers and mainframe computers and it is also used as a temporary storage for data. Access to control memory data requires less time than to main memory; it speeds up CPU operation by reducing the number of memory references for data storage and retrieval. Access is performed as part of a control section sequence while the master clock oscillator is running. The memory controller is considered as a digital circuit which is used to manage the flow of data going to and from the main memory. It can be a separate chip or integrated into another chip, such as a microprocessor. This is also called a Memory Chip Controller (MCC). Memory controllers contain the logic necessary to read and write to DRAM and also used to refresh the DRAM by sending current through the entire device. Without constant refreshes, DRAM will lose the data written to it as the capacitors leak their charge within a fraction of a second. The operations of reading and writing to DRAM is performed by selecting the row and column data addresses of the DRAM as the inputs to the multiplexer circuit where the demultiplexer on the DRAM uses the converted inputs to select the correct memory location and return the data which is then passed back through a multiplexer to consolidate the data in order to reduce the required bus width for the operation. The control memory address register specifies the address of the microinstruction. The control data register holds the microinstruction read from memory. The microinstruction contains a control word that specifies one or more microoperations for the data processor. Microinstructions are stored in control memory in groups with each group specifying a routine (refer Figure 4.7). Each computer instruction has its own microprogram routine to generate the microoperations. The hardware that controls the address sequencing of the control memory must be capable of sequencing the microinstructions within a routine and be able to branch from one routine to another. Following steps are required where the control must undergo during the execution of a single computer instruction:

- Load an initial address into the CAR when power is turned on in the computer.

This address is usually the address of the first microinstruction that activates the instruction fetch routine to hold the instruction.

- The control memory then goes through the routine to determine the effective address of the operand where Address Register or AR holds operand address.

- The next step is to generate the microoperations that execute the instruction by considering the opcode and applying a mapping.

- After execution, control must return to the fetch routine by executing an unconditional branch.

NOTES

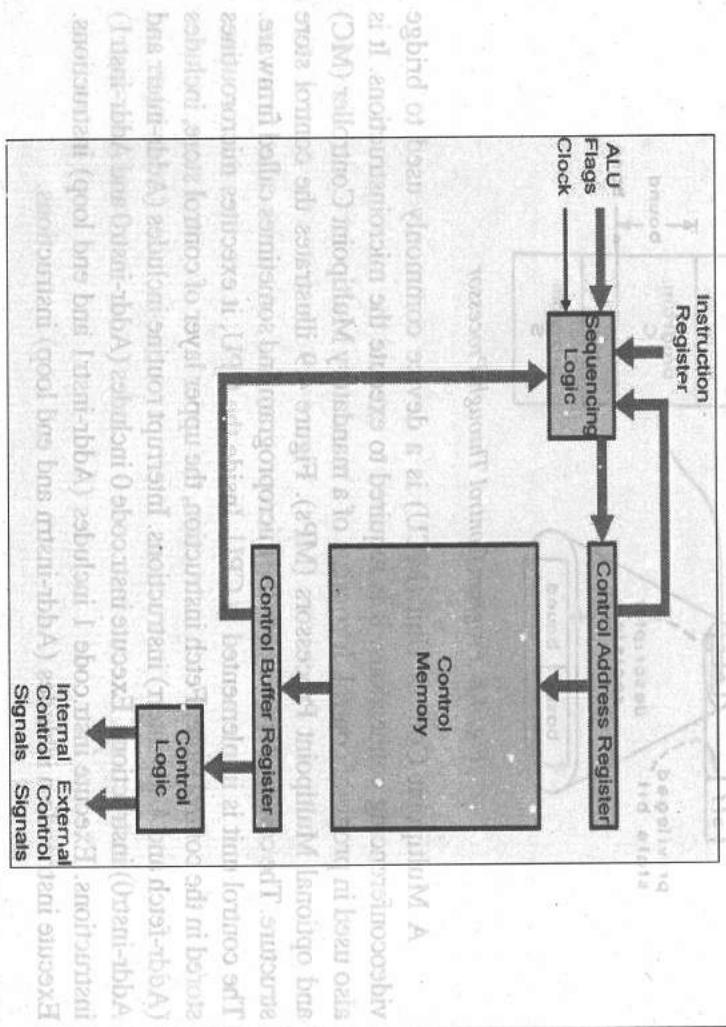


Fig. 4.7 Control Signals and Control Memory

Sequence login unit issues read command whereas word specified in control address register is read into control buffer register. The control buffer register generates control signals and next address information. The sequence login loads new address into control buffer register based on next address information from control buffer register and ALU flags. Each processor operates under the control of an instruction stream issued by its own control unit, i.e., each processor is capable of executing its own program on a different data. This means that the processors operate asynchronously. This process is done by different operations by different data at the same time. Controlling operation is performed by a special hardware register known as a descriptor register which is typically used in multiprogramming system. In Figure 4.8, all memory references by the processor are checked by an extra section of hardware that is interposed in the path to the memory. The descriptor register controls exactly which part of memory is accessible. The descriptor register contains two components: a *base* value and a *bound* value. The base is the lowest numbered address the program may use and the bound is the number of locations beyond the base that may be used.

The *base* contains the lowest address of the memory block. The *bound* contains the highest address of the memory block. The *base* and *bound* are used to calculate the effective address of the memory block.

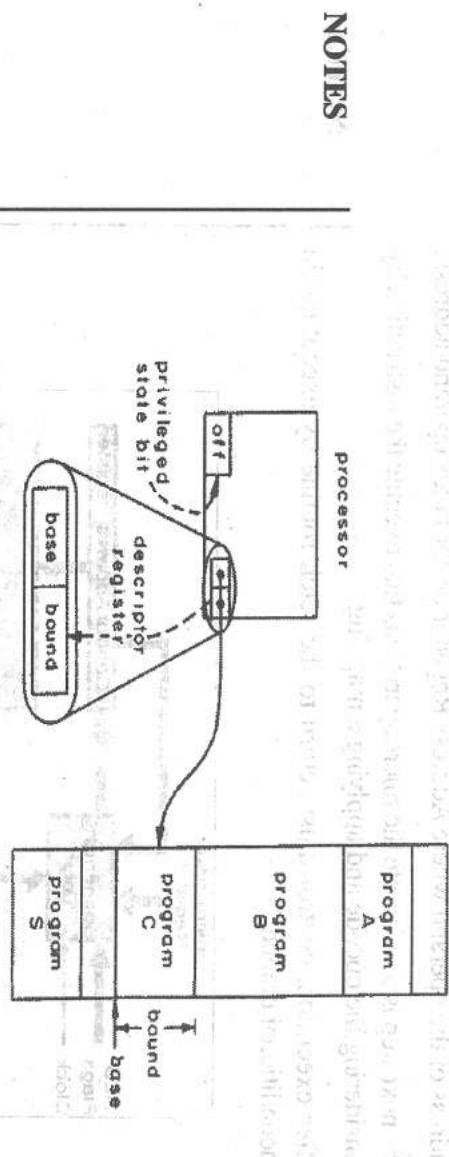


Fig. 4.8 Program Control Through Processor

A Multipoint Control Unit (MCU) is a device commonly used to bridge videoconferencing connections. It is required to execute the microinstructions. It is also used in program control. It consists of a mandatory Multipoint Controller (MC) and optional Multipoint Processors (MPs). Figure 4.9 illustrates the control store structure. The control store contains the microprogram and sometimes called firmware. The control unit is implemented in CPU. Inside the CPU, it executes microroutines stored in the control store. Fetch instruction, the upper layer of control store, includes (Addr-fetch and Addr-interr) instructions. Interrupt routine includes (Addr-instr0 and Addr-instr1) instructions. Execute instr.code 1 includes (Addr-instr1 and end loop) instructions. Execute instr.code n includes (Addr-instrn and end loop) instructions.

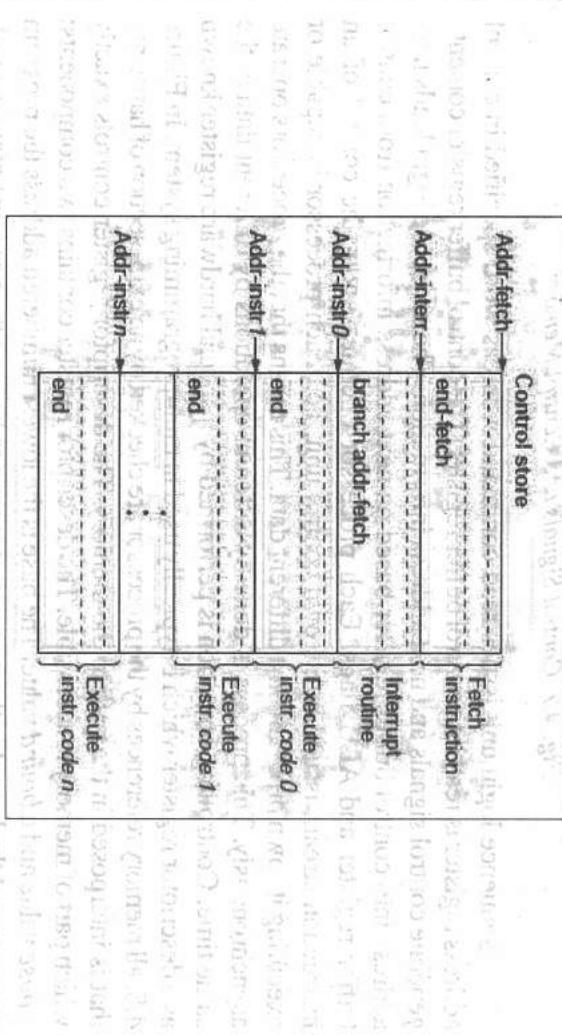


Fig. 4.9 Control Store Organization

The microroutines contain branches which have to be interpreted by the microprogrammed controller. The sequencer is controlling the right execution sequence

NOTES

of microinstructions. The sequencer is a small unit of the control unit. A set of control signals activates the microoperations which have to be executed in a given control step. Control units can be implemented hardwired or microprogrammed.

4.2.3 Hardwired Implementation

In a hardwired control unit, hardware (combinational and sequential circuits) generates the control signals using logic design. These hardware circuits include gates, flip-flops, decoders, multiplexers and other digital circuits. These circuits are designed to implement the control logic. Here the instruction is divided into fields that combine to drive the control lines. Thus, the control signals are generated by transforming the input logic into a set of output signals using these combinational circuits.

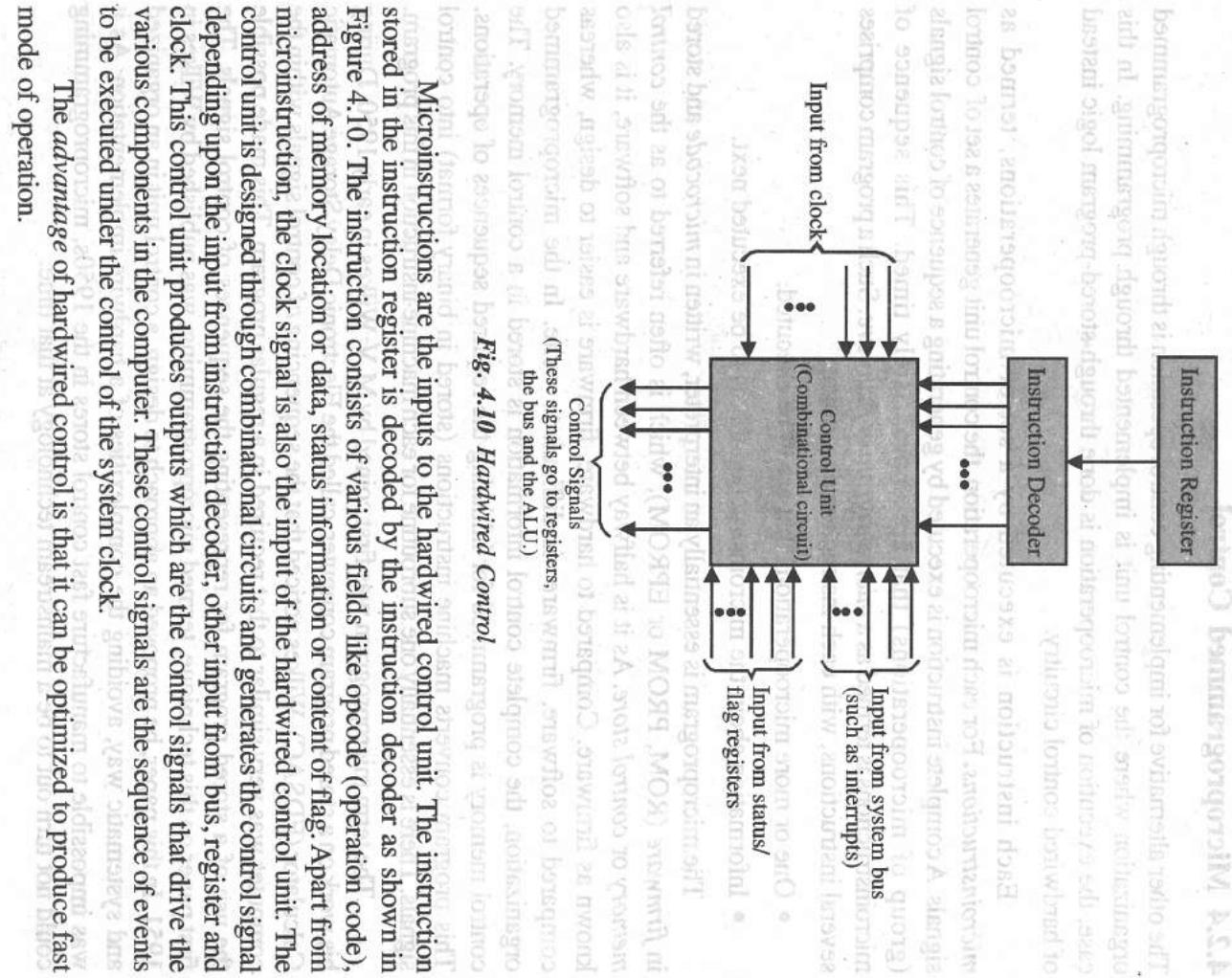


Fig. 4.10 Hardwired Control

Microinstructions are the inputs to the hardwired control unit. The instruction stored in the instruction register is decoded by the instruction decoder as shown in Figure 4.10. The instruction consists of various fields like opcode (operation code), address of memory location or data, status information or content of flag. Apart from microinstruction, the clock signal is also the input of the hardwired control unit. The control unit is designed through combinational circuits and generates the control signal depending upon the input from instruction decoder, other input from bus, register and clock. This control unit produces outputs which are the control signals that drive the various components in the computer. These control signals are the sequence of events to be executed under the control of the system clock.

The *advantage* of hardwired control is that it can be optimized to produce fast mode of operation.

The *disadvantage* of hardwired control is that the instruction set and the control logic are directly tied by special circuits that are complex and difficult to design or modify. Also, the number of different microoperations available in a given system is finite. If someone decides later to extend the instruction set, the physical components in the computer must be changed. Thus, modification in the design of hardwired control organizations requires changes in wiring among the various components. Hence, the design does not show any flexibility. Further, since the number of control lines in today's computer exceeds a hundred, it is not easy to design, test and implement a hardwired control system. Also, this is prohibitively expensive as not only new chips need to be fabricated but also the old ones must be located and replaced.

NOTES

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:

- 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 International Business Machines or 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. The System/360 is so named because it handled 'all 360 degrees of computing', i.e., it was aimed to provide an expandable system that would serve every data processing need. Typically, the IBM System/360 (S/360) was a mainframe computer system family broadcasted by IBM. It was the first family of computers specifically designed to enclose the complete range of applications, from small to large, both commercial and scientific. 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 8086 and Motorola 68000 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 and execute the instruction and prepare it again for fetching the next instruction from the main memory. To execute an instruction by 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 4.11):

- **Instruction Register (IR):** This holds the instruction to be executed.
- **Microinstruction Address Generation:** This generates the address where next the instruction (to be executed) is stored in the control memory. This address is given stored in instruction register and produces an output which is the address of the instruction in 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.

A microprogram flow chart is shown to explain how the control signals are generated. The flowchart starts with the instruction being fetched from the main memory and decoded by the microinstruction decoder. The decoder generates control signals for the control store and the microinstruction buffer. The control store then generates the microinstruction address, which is used to fetch the microinstruction from the control store. The microinstruction is then decoded by the microinstruction decoder, which generates control signals for the various functional units of the processor.

NOTES

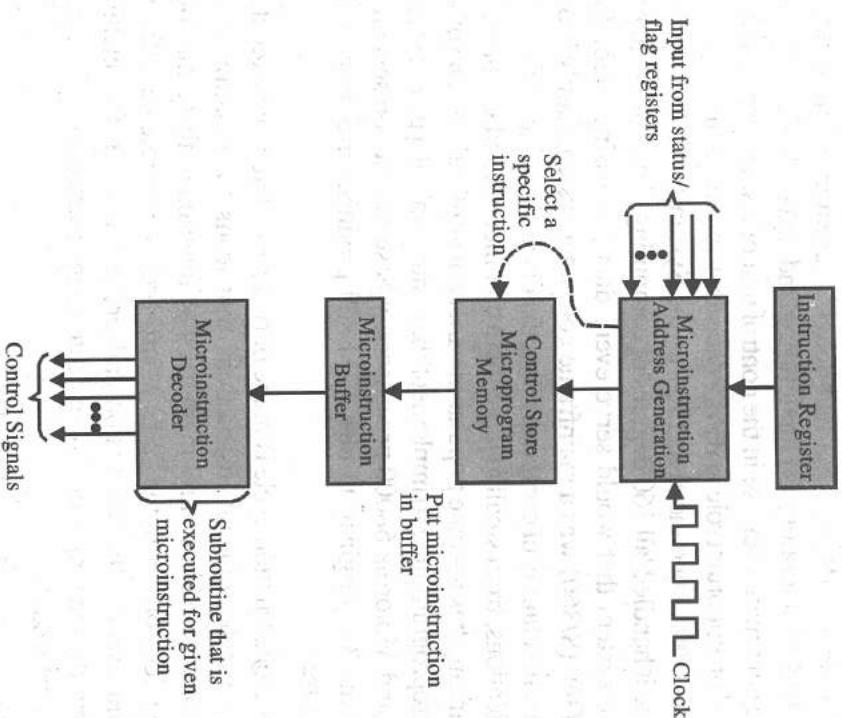
NOTES

Fig. 4.11 Microprogrammed Control Unit

Writable Control Store

Microprograms are usually stored in read only memory, since alteration in it is usually not required once a control unit is produced. However, it is not always so. Some computers allow the modification and they map part of the control store addresses onto Random Access Memory or RAM. In such a system, the main memory stores both control instructions and the instructions and data that programmers use. As many machines have complex instructions which may be required to be changed, one may need a writable store. The reason for change is simple. Since the microprogram for a complex processor will also be complex, it may be filled with bugs. Hence, having a writable control will help in revising the machine control and updating it. Also, this provides the added flexibility to assembly language programmers in the sense that the latter can write their own microcode so that they can extend the machine's 'native' instruction set with some special purpose instructions, especially for machines designed for special purpose (embedded systems). In such a memory instruction, one just needs to modify the various opcodes of the microprograms to have a changed set of control instructions. A computer with a writable control memory is termed as *dynamically microprogrammable* since it is possible to change the content of the control memory for such computers.

The control memory refers to a set of words in which each word represents a microinstruction. The computers that have writable microprogrammed control unit use RAM as two separate memories. One is used as the main memory and the other

is used as the control memory. The content of the main memory may change during the execution of a program but control memory holds fixed microprogram that cannot be changed by a normal user. Once we start the machine, it executes a 'boot' microprogram. This boot program is a sequence of microinstructions that are stored in ROM. It loads the rest of the microcodes into RAM from an external device, such as a hard disk.

Hardwired vs Microprogrammed Computers

Nowadays most computers are microprogrammed because microprogramming provides flexibility to their functionality. In case of a hardwired control unit, once the control unit is designed and built, it is virtually impossible to alter its architecture and instruction set. However, in the case of a microprogrammed computer, it is possible to change the computer's instruction set by altering the microprogram stored in its control memory. For example, the basic computer uses the four-bit opcode resulting in 16 possible instructions. If it is required to add seven more instructions to the instruction set, one can easily do it in case of microprogrammed control unit by expanding the microprogram stored in its control memory. To implement it in a hardwired version computer, one would require a complete redesigning of hardware to redesign the controller circuit.

Advantages

The advantages of a microprogrammed control unit over a hardwired unit are as follows:

- In the microprogrammed control unit, the task of designing the computer is simplified as the process of specifying the architecture and instruction set requires software design as opposed to hardware design. Thus, it helps to design more powerful instruction sets, especially those required in the Complex Instruction Set Computer (CISC) architecture.
- It eases the implementation of a code compatible family of computers like CISC.
- If any modification in the instruction set is required, the microprogram is simply updated and no change is required in the actual hardware.
- Since the microprograms are relatively easy to modify, microprogrammed control units are considered more flexible in comparison to hardwired control units.
- Microprogramming allows the convenient hardware/software trade-offs. If what you want is not implemented in hardware (for example, your machine has no multiplication statement), it can be implemented in the microcode.
- It offers a systematic control unit design.
- It has the ability to emulate other computer systems.
- As microprogram for a complex processor is complex, it can be used for debugging the system through microdiagnostics.
- It involves a low marginal cost for letting additional microinstructions fit within the control store.
- Using a writable control store makes possible to reconfigure the processor.

NOTES

Disadvantages

As far as speed is considered, hardwiring is faster than microprogramming because in the latter, one has to fetch the microinstructions from the control memory. Fetching from memory always requires more time compared to processing. Also, due to the control memory and its access circuitry, the cost of designing the microprogrammed control system is higher. However, with the availability of cheaper memory technologies, including high speed memories, these disadvantages are gradually becoming insignificant. Nevertheless, for certain applications, hardwired computers are preferred to microprogrammed system due to following reasons:

- Microprogramming is uneconomical for small systems as a high cost is involved in designing control memory and the microinstruction sequencing logic.
- All instructions have to pass an additional level of interpretation, slowing down the program execution.
- Control store access time limits the instruction cycle.
- Heavy investment is needed in the actual development since appropriate tools are required.

4.2.5 Microinstructions

The execution of every machine instruction is done with the help of a sequence of microinstructions known as a routine. Each computer has its own microprogram routine that is stored in the control memory. Thus, control memory is used for storing the microprograms, which are basically a sequence of microinstructions. These microinstructions consist of a sequence of microoperations that need to be performed during each cycle (fetch, indirect, execute, interrupt) and also specify the sequencing of these cycles. A set of control signal lines is provided by a control unit, with each signal line representing a zero or one. Microinstructions are responsible for generating control signals from the control unit. For implementing a desired microoperation, these control signals are sent to the desired control lines. Control signals comprise enabling signals and operation selection signals. The former are used for sending or receiving of data at the registers, while the latter decide the operation to be performed. For implementing a register-to-register transfer operation. For example, using microinstruction, the control signals enable the source register which sends its output and also enables the destination register to accept the input by the respective control signal lines. Thus, each microinstruction is capable of generating a set of control signals on the control lines. These control signals, in turn, implement one or more microoperations.

A control word refers to a set of control signals in which each bit represents a single control line. It can be programmed to perform various operations on the various components of the system. Each word in the control memory contains within it a microinstruction. A sequence of microinstructions constitutes a micropogram. Since usually the microprogram once made does not need alteration, ROM can be used for storing. The instruction is hence determined by the content of the word in ROM at a given address.

NOTES

As many microprogrammings may require identical codes, it is beneficial to store these codes as subroutines. Thus, the simplest way to organize control memory is to arrange microinstructions for the various subroutines of the machine instruction in the memory. The microinstructions in each routine are to be executed sequentially. Each routine ends with branch or jump instruction, determining where to go next. There is a special execution cycle routine whose only purpose is to find one of the machine instruction routines (AND, ADD, and so on) to be executed next, depending on the current opcode. Table 4.1 summarizes the control unit organization.

Table 4.1 Control Unit Organization

Jump to Indirect or Execute	Fetch Cycle Routine
Jump to Execute	Indirect Cycle Routine
Jump to Fetch	Interrupt Cycle Routine
Jump to Opcode Routine	Execute Cycle Begin
Jump to Fetch or Interrupt	AND Routine
Jump to Fetch or Interrupt	ADD Routine

This instruction causes branching to a specified main memory address if Arithmetic and Logic Unit or ALU's last result is zero, thereby checking the state of the zero flag. The pseudocode for this microprogram can be as follows:

Test 'Zero Flag': The zero flag can be set as branch to zero; else it is unconditional and branch to non-zero.

Zero: This microcode replaces the program counter with the address provided in the instruction.

Non-Zero: This microcode, which may set flags, if desired, indicates that the branching has not taken place.

Execution of Microinstructions

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, an MCU should have the following components (refer Figure 4.12).

- **Microprogram Sequencer:** It generates the address of the next microinstruction that needs to be retrieved from the control memory.
- **Control Address Register or CAR:** Its function is to hold the address of control memory generated by microprogram sequencer.
- **Control Memory or CM:** It stores all microprograms that are constituted of control words. It is usually a ROM.

Table 4.2 illustrates the sequence of control words in control memory for a simple microinstruction.

Control Word Address	Control Word
1	Write enable
2	To register
3	Write enable of CAR
4	Write enable of ROM
5	Write enable of ROM

NOTES

Instruction Register

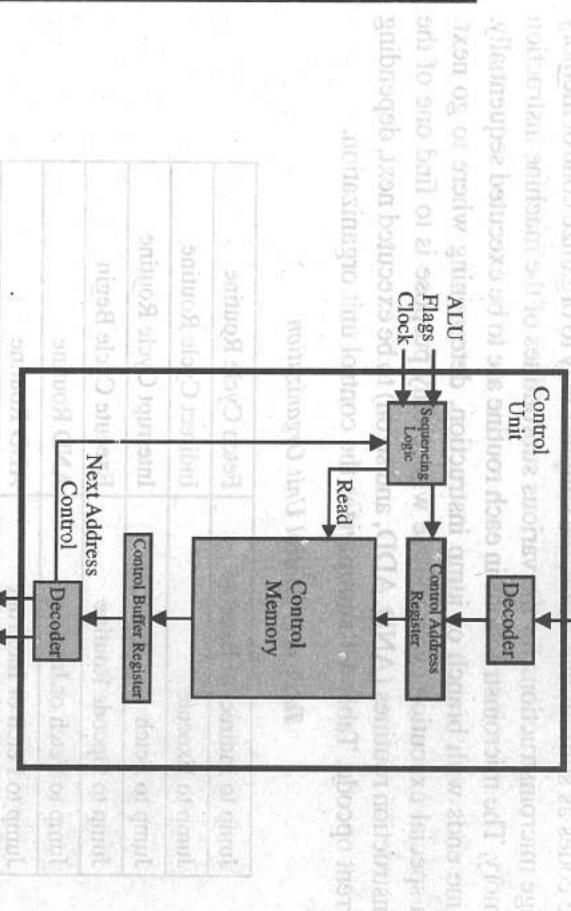


Fig. 4.12 Fundamental Components of a Microprogrammed Unit

• **Control Buffer Register or 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:

Check Your Progress

1. Write the advantage of hardwired control.
2. Who coined the term 'micropogram'?
3. How the execution of every machine instruction is done?
4. What does 'control word' refers?
5. Write the three basic functions of control buffer register.

Step 1: The instructions are fetched from main memory and are stored in IR.

Step 2: Opcode is decoded.

Step 3: To execute an instruction, the microprogram sequencer issues a READ command for control memory.

Step 4: Microinstructions are retrieved from CM, from the address specified in CAR.

Step 5: The microinstruction is read from the control memory and is transferred to a control buffer register.

Step 6: The content of control buffer register generates control signal and the address information for the sequencing logic unit.

Step 7: The microprogram sequencer loads a new address into CAR based on the next address information from control buffer register and ALU flags.

4.3 MICROOPERATION

Control Unit

In the central processing unit of a computer system, microoperations or micro-ops or µops are detailed low level instructions specifically used to execute complex machine instructions. Typically, a microoperation is a basic operation that can be defined on the data stored in registers. The various types of microoperations include data transfer microoperation, arithmetic microoperation, logic microoperation and shift microoperation.

Arithmetic microoperations are related to the basics of arithmetic operations

used for addition, subtraction, increment, decrement and complement functions. To add the contents of two registers R1 and R2 and then to place the result in a third register R0, we write the syntax $R0 \leftarrow R1 + R2$. Here the symbol '+' (plus for addition) is used for addition and it refers to the real binary addition which is performed using a binary ripple carry adder/subtractor. In simple register transfer, arithmetic microoperations are performed depending on the condition specified in the syntax.

For example, $R0 \leftarrow R1 + R2$ if $K = 1$.

This microoperation will be only performed if $K = 1$.

(bnsun We specify the condition as, $K : R0 \leftarrow R1 + R2$.

This microoperation will be only performed if $K = 1$.

Arithmetic microoperations perform various arithmetic operations on the data stored in registers. Arithmetic operations can be done only on numerics (including numeric subfields, numeric arrays, numeric array elements, numeric table elements, numeric named constants, numeric figurative constants, and numeric literals). In general, arithmetic operations are performed using the packed-decimal format. This means that the fields are first converted to packed decimal format prior to performing the arithmetic operation, and then converted back to their specified format (if necessary) prior to placing the result in the result field. The exceptions are as follows:

- If all operands are float, then the operation will use float arithmetic.
- If all are integer or integer and unsigned, then the operation will use integer arithmetic.
- If any operands are float, then the remaining operands are converted to float.

Besides the above mentioned basic arithmetic microoperations, i.e., Addition, Subtraction, Increment and Decrement there are some additional arithmetic microoperations for specific purposes. These include 'add with carry', 'subtract with borrow' and transfer/load. Table 4.2 illustrates the arithmetic microoperations and their functions.



NOTES

Arithmetic Microoperation	Function
$R3 \leftarrow R1 + R2$	Contents of R1 plus R2 transferred to R3
$R3 \leftarrow R1 - R2$	Contents of R1 minus R2 transferred to R3
$R2 \leftarrow R2'$	Complement the contents of R2
$R2' \leftarrow R2 + R2$'s complement	Complement the contents of R2 (negate)
$R3 \leftarrow R1 + R2' + 1$	Subtraction
$R1 \leftarrow R1 + 1$	Increment by one
$R1 \leftarrow R1 - 1$	Decrement by one

Figure 4.13 illustrates binary adder, subtractor, incrementer which use logic gates to perform arithmetic microoperations.

The binary adder or bit-serial adder is a digital circuit that performs binary addition bit-by-bit. The serial full adder has three single-bit inputs for the numbers to be added and the carry in. There are two single-bit outputs for the sum and carry out. The carry-in signal is the previously calculated carry-out signal. The addition is performed by adding each bit, lowest to highest, one per clock cycle. Binary subtractor includes half-subtractor and full-subtractor. The half subtractor is a combinational circuit which is used to perform subtraction of two bits. It has two inputs, X (minuend) and Y (subtrahend) and two outputs D (difference) and B (borrow). The full-subtractor is a combinational circuit which is used to perform subtraction of three bits. It has three inputs, X (minuend) and Y (subtrahend) and Z (subtrahend) and two outputs D (difference) and B (borrow). In digital circuits, a binary adder-subtractor is a circuit that is used in adding or subtracting binary numbers. Figure 4.13 represents a circuit that adds or subtracts depending on a control signal. Constructed circuit is used to perform both addition and subtraction at the same time. The addition and subtraction operations can be combined into one common circuit called binary adder-subtractor. These operations can be done by including an exclusive-OR gate with each full adder in binary adder.

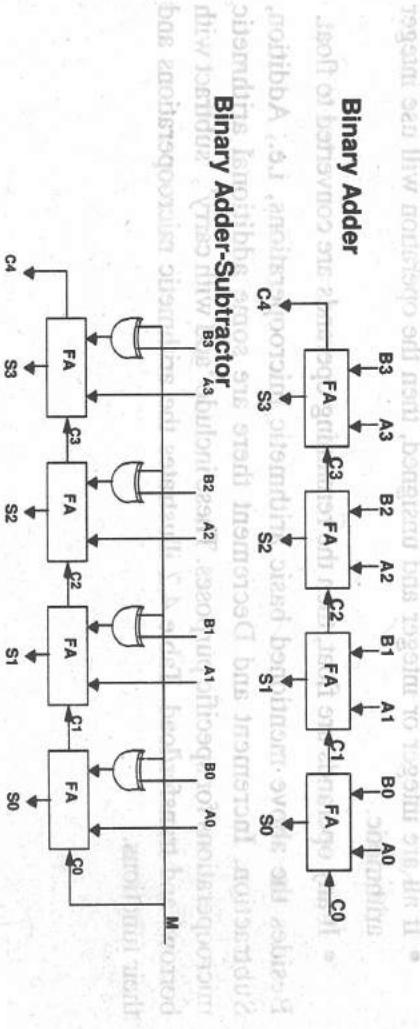


Fig. 4.13 Binary Adder and Binary Adder-Subtractor

Here, Full Adder = 2-Bits Sum + Previous Carry

Therefore, C0 represents input carry and C4 represents output carry.

Binary Incrementer

Figure 4.14 illustrates a binary incrementer circuit where 1 is added to binary input information of n digits to provide binary output information characterized in that output information of the lowest digit is produced as inverted input information by an inverter circuit, and that output information of each of the second-lowest to n th digits is produced by passing either input information of the particular digit or its inverted signal from an inverter circuit through a corresponding one of transfer gate transistor paths which are controlled by the information of the digits lower than the particular digit.

NOTES

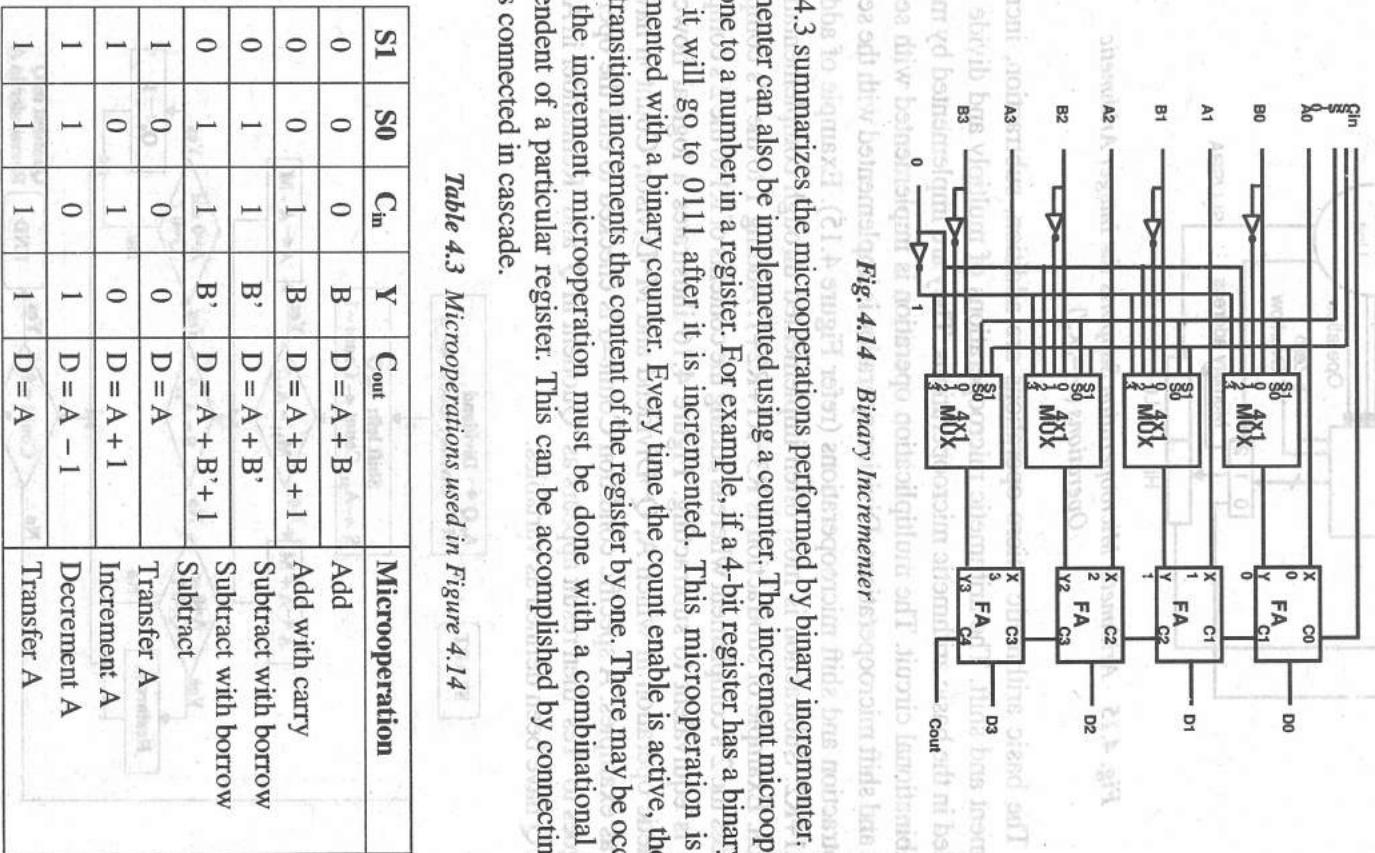


Fig. 4.14 Binary Incrementer

Table 4.3 summarizes the microoperations performed by binary incrementer. Binary incrementer can also be implemented using a counter. The increment microoperation adds one to a number in a register. For example, if a 4-bit register has a binary value 0110, it will go to 0111 after it is incremented. This microoperation is easily implemented with a binary counter. Every time the count enable is active, the clock pulse transition increments the content of the register by one. There may be occasions when the increment microoperation must be done with a combinational circuit independent of a particular register. This can be accomplished by connecting half-adders connected in cascade.

Table 4.3 Microoperations used in Figure 4.14

S1	S0	C _m	Y	C _{out}	Microoperation
0	0	0	B	D = A + B	Add
0	0	1	B	D = A + B + 1	Add with carry
0	1	1	B'	D = A + B'	Subtract with borrow
0	1	1	B'	D = A + B' + 1	Subtract
1	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	D = A		Transfer A

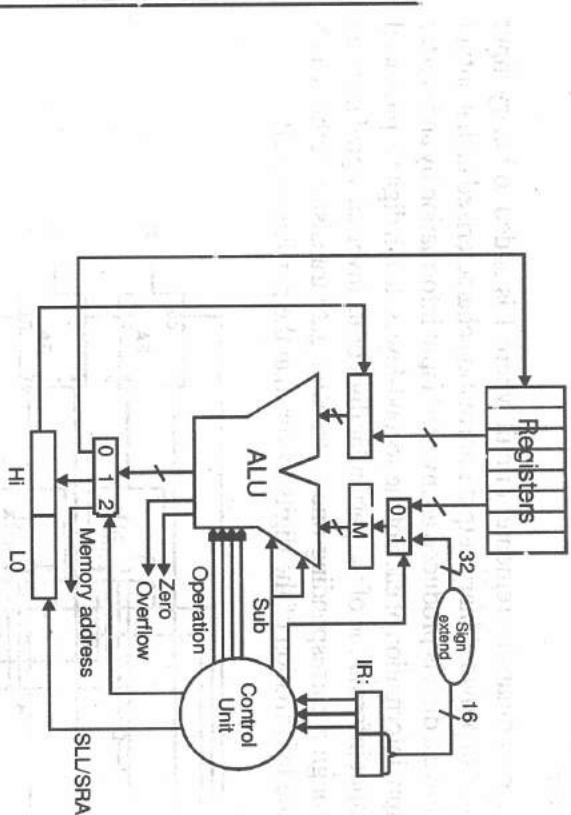
NOTES

Fig. 4.15 Arithmetic Microoperation Supports the Integer Arithmetic Operations (+, -, ×, /)

The basic arithmetic micro operations are addition, subtraction, increment, decrement and shift. The arithmetic microoperations of multiply and divide are not included in the basic arithmetic microoperations. They are implemented by means of a combinational circuit. The multiplication operation is implemented with sequence of add and shift microoperations. Division operation is implemented with the sequence of subtraction and shift microoperations (refer Figure 4.15). Example of addition is R3-R1+R2. Subtraction is most often implemented through complementation and addition. Example of subtraction is R3-R1+R2+1. Adding 1 to the 1's complement produces the 2's complement whereas adding the contents of R1 to the 2's complement of R2 is equivalent to subtracting. Figure 4.16 illustrates a logical flowchart of arithmetic operation in which A, Q-Dividend and M-Divisor, Count-n have been taken as examples. A specific condition Count=0 is checked to end the operation if flow goes to 'Yes' then result appears as 'Quotient in Q' and 'Remainder in A' where A and Q have been defined as variables.

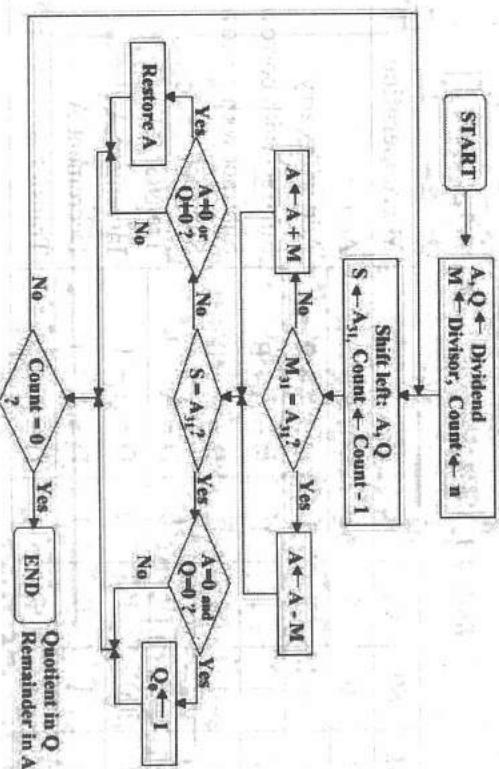


Fig. 4.16 Logical Flowchart of Arithmetic Operation

Arithmetic Operations

Arithmetic microoperations perform various arithmetic tasks in terms of addition, subtraction, 1's complement, 2's complement, etc. The arithmetic operations are performed in the Arithmetic and Logic Unit (ALU) with the help of various logic and arithmetic operators. The mode input **M** selects M=0 in the logic unit to perform logical operations and selects M=1 in arithmetic unit to perform arithmetic operations. Table 4.4 shows how the arithmetic microoperations are performed in ALU.

Table 4.4 Logic and Arithmetic Operations performed by the ALU

M = 0 Logic Operations			
S1	S0	C0	Function
0	0	X	A_iB_i
0	1	X	$A_i + B_i$
1	0	X	$A_i \oplus B_i$
1	1	X	$(A_i \oplus B_i)'$

M = 1 Arithmetic Operations			
S1	S0	C0	Function
0	0	0	$A \bmod 2^k$ or A is transferred
0	0	1	$A + 1$ since 1 is added to A
0	1	0	$A + B$
0	1	1	$A + B + 1$
1	0	0	$A + B'$
1	0	1	B is subtracted from A (i.e. $B' + A + 1$)
1	1	0	$B' + B$
1	1	1	$B - A$ (or $A' + B + 1$)

Given below are some examples of arithmetic operations.

- $(A + \bar{B})$
- $(A + \bar{B} + 1)$
- $(A + \bar{B} - 1)$
- $(\bar{A} + B + 1)$

Table 4.5 summarizes a list of arithmetic operations and their representations.

Table 4.5 Arithmetic Operations and their Functions

Arithmetic Operation	Functions
ADD	$R3 \leftarrow R1 + R2$
SUBTRACT	$R3 \leftarrow R1 - R2$
COMPLEMENT (1'S COMPLEMENT)	$R2 \leftarrow R2$ ($R2 \leftarrow \bar{R}_2$)
2'S COMPLEMENT	$R2 \leftarrow R2 + 1$ ($R2 \leftarrow \bar{R}_2 + 1$)
SUBTRACT (2 C)	$R3 \leftarrow R1 + R2 + 1$ ($R3 \leftarrow R1 + \bar{R}_2 + 1$)
INCREMENT	$R1 \leftarrow R1 + 1$
DECREMENT	$R1 \leftarrow R1 - 1$
NEGATE	2'S COMPLEMENT
MULTIPLICATION	Shift Left
DIVISION	Shift Right

NOTES

In Table 4.5, the statement SUBTRACT (2 C) R3←R1+R2+1 is the subtract operation performed by adding the 2'S COMPLEMENT of the number to be subtracted, i.e., $a - b$ is implemented as $a + (-b)$. Here is an example, which presents to you a clear picture of various mathematical operations. Let R1 be 1001 1100 and R2 be 0101 0110, so the result of mathematical operations are summarized in Table 4.6:

Table 4.6 Result Table of Various Arithmetic Operations of R1 (1001 1100) and R2 (0101 0110) Input Values

Function	Result
ADD	1111 0010
SUBTRACT	0100 0110
COMPLEMENT	1010 1001
2's COMPLEMENT	1010 1010
SUBTRACT (2 C)	0100 0110
INCREMENT	1001 1101
DECREMENT	1001 1011

Take another example of subtraction in arithmetic microoperation. In this, $A - B = A + 2^{\text{'}}\text{s complement of } B$ and the 2's complement of $B = \sim B + 1$. Here, $\sim B = B \wedge '1'$ that means B exclusive OR '1' where '1' is n 1's. Truth Table 4.7 shows the subtraction arithmetic operation.

Table 4.7 Resulted Truth Table

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Logic Microoperations

Logic microoperations refer to the bitwise manipulation of the contents of a register. It uses NOT, AND, OR and eXclusive-OR (XOR). They are quite useful for masking bits. They perform operations such as clearing, setting and complementing string of bits in registers. These operations operate each bit of register as binary variable, such as:

P: R1 R1 R2

The above statement is executed on the individual and separate bits. It also specifies a logic microoperation to be executed on the individual bits of the registers provided that the control variable P = 1. Assume that each register has four bits. Let the content of R1 be 1010 and the content of R2 be 1100. The XOR microoperation stated above symbolizes the following logic computation:

1010 Content of R1	1100 Content of R2	1110 Content of R1 after P = 1
--------------------	--------------------	--------------------------------

The content of R1 after the execution of the microoperation, is equal to the bit-by-bit XOR operation on pairs of bits in R2 and previous values of R1. There are some symbols to denote logic microoperations OR, NOT (complement) and AND. They are distinguished from the corresponding symbols which are used to express Boolean functions. The symbol (\vee) will be used to denote an OR microoperation and the symbol (\wedge) to denote an AND microoperation. The complement microoperation is the same as the l's complement and uses a bar on top of the symbol that denotes the register name.

The + symbol has two meanings; it will be possible to distinguish between them by noting where the symbol occurs. When the symbol + occurs in a microoperation, it will denote an arithmetic plus. When it occurs in a control (or Boolean) function, it will denote an OR operation. For example, P + Q. The + between P and Q is an OR operation between two binary variables of a control function. The characteristics are as follows:

- Logic microoperations are bit-wise operations because they work individually on the bits of data.
- They are useful for bit manipulations of binary data.
- They make logical decisions based on the bit value.
- There are sixteen logical functions which are defined over two binary variables, such as A and B.

Table 4.8 summarizes the functions of the sixteen logic microoperations. The other logical microoperations are created with the help of these combinations.

Table 4.8 represents the truth table which shows the sixteen functions of the corresponding sixteen logic microoperations.

Inputs		Outputs	
A	B	$A \wedge B$	0
1	0	$A = A \wedge B$	0
0	1	$B = A \wedge B$	0
1	1	$B = A \wedge B$	1
Complement		\bar{A}	1
Complement		\bar{B}	0
OR		$A \vee B$	1
XOR		$A \oplus B$	0
NOT		\bar{A}	0
AND		$A \wedge B$	0
Complement		$\bar{A} \wedge B$	0
Complement		$A \wedge \bar{B}$	0
NOT		$\bar{A} \wedge \bar{B}$	1
XOR		$A \oplus \bar{B}$	1
OR		$A \vee \bar{B}$	1
Complement		$\bar{A} \vee B$	1
Complement		$\bar{A} \vee \bar{B}$	1

NOTES

Table 4.8 Truth Table Shows the Sixteen Logic Microoperations

0000	F0 = 0	$F \leftarrow 0$	Clear
0001	F1 = xy	$F \leftarrow A \wedge B$	AND
0010	F2 = x'y	$F \leftarrow A \wedge B'$	NOT
0011	F3 = x	$F \leftarrow A$	Transfer A
0100	F4 = x'y	$F \leftarrow A' \wedge B$	NOT for B
0101	F5 = y	$F \leftarrow B$	Transfer B
0110	F6 = x \oplus y	$F \leftarrow A \oplus B$	Exclusive - OR
0111	F7 = x + y	$F \leftarrow (A \vee B)$	OR
1000	F8 = (x + y)'	$F \leftarrow A \vee B'$	NOR
1001	F9 = (x \oplus y)'	$F \leftarrow (A \oplus B)'$	Exclusive - NOR
1010	F10 = y'	$F \leftarrow B$	Complement B
1011	F11 = x + y'	$F \leftarrow A \vee B$	Complement A
1100	F12 = x'	$F \leftarrow A'$	
1101	F13 = x' + y	$F \leftarrow A' \vee B$	
1110	F14 = (xy)'	$F \leftarrow (A \wedge B)'$	NAND
1111	F15 = 1	$F \leftarrow \text{all } 1's$	Set to all 1's

Hardware Implementation for Logic Microoperation

Logic operations are basically the binary operations which are performed on the string of bits stored in the registers. For a logic microoperation, each bit of a register is treated as a variable. Some of the common logic microoperations are AND, OR, NOT or Complement, Exclusive OR, NOR, NAND. For implementation, let us first consider a question how many logic operations can be performed with two binary variables. We can have possible combination of input of two variables. These are 00, 01, 10 and 11. Now, for all these input combination we can have 16 output combination. This implies that for two variables we can have 16 logical operations. An arithmetic circuit is normally implemented using parallel adder circuits. Figure 4.17 illustrates an example of circuit where S1 and S0 are defined as binary input values whereas 'Output' is generated as per defined 'Operations'.

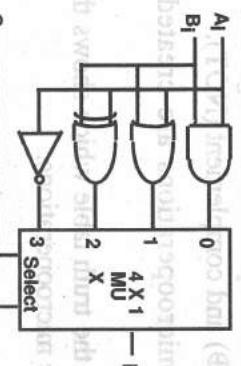


Fig. 4.17 Example of Circuit

Table 4.9 summarizes the function table of corresponding variables S1 and S0.

Table 4.9 Function Table of Corresponding Variables S1 and S0

S1	S0	Output	Operation
0	0	$F = A \wedge B$	AND
0	1	$F = A \vee B$	OR
1	0	$F = A \oplus B$	XOR
1	1	$F = A'$	Complement

Applications of Logic Microoperations

Control Unit

Logic microoperations are used to manipulate either the words in a register or the individual bits. Here is an example which shows how the data in a register B is used to modify the contents of register A.

Table 4.10 summarizes the operations and notations performed in registers A and B.

Table 4.10 Operations Performed in Registers A and B

Description	Notation
Selective-Set	$A \leftarrow A + B$
Selective-Complement	$A \leftarrow A \bullet B'$
Selective-Clear	$A \leftarrow A \bullet B$
Mask (Delete) $\oplus A \rightarrow A$	$1 + A \leftarrow A \bullet B$
Clear	$A \leftarrow A \oplus B$
Insert	$A \leftarrow (A \bullet B) + C$
Compare	

Logic Microoperation's Selective Set

In the selective set operation, the selected bit pattern of B is used in A. It is set as follows:

$$1100A$$

$$\underline{1010B}$$

$$(Langkah 5)$$

If 1 is set to B, the A gets also set to 1, otherwise A retains to keep the previous value.

Logic Microoperation's Selective Complement

The bit pattern in B is used to set the complement for bits in A. It is set as follows:

$$1100A$$

$$\underline{1010B}$$

$$(Langkah 6)$$

If a bit in B is set to 1, that same position in A gets complemented from its original value, otherwise it is unchanged.

Logic Microoperation's Selective Clear

In selective clear operation, the bit pattern in B is used to clear certain bits in A. It is set as follows:

$$1100A$$

$$\underline{1010B}$$

$$(Langkah 7)$$

If a bit in B is set to 1, A gets set to 0 at the same position, otherwise it is unchanged.

NOTES

Logic Microoperation's Mask Operations

In a mask operation, the bit pattern in B is used to clear certain bits in A:

$$\begin{array}{r} \text{Input at } B \text{ register is } 0000\ 0000\ 1100\ A \\ \text{A register initial value is } 1010\ B \\ \hline 1010\ B \end{array}$$

A register is initialized to 1010B. A register value is 000000001100A. A register value is 1010B.

If a bit in B is set to 0, A gets set to 0 at the same position, otherwise it is unchanged.

Logic Microoperation's Clear Operations

In a clear operation, if the bits in the same position in A and B are the same, they are cleared in A, otherwise they are set in A.

$$\begin{array}{r} \text{Input at } B \text{ register is } 0000\ 0000\ 1010\ B \\ \text{A register initial value is } 0110\ A + 1 \\ \hline 0110\ A + 1 \quad (A \leftarrow A \oplus B) \end{array}$$

B register value is 000000001010B. A register initial value is 0110A + 1. A register value is 0110A + 1.

Logic Microoperation's Insert Operations

Insert is an operation used to introduce a specific bit pattern into register A. It does not affect or change the other bit positions. It is done as follows.

- The desired bit positions are cleared using a mask operation.
- An OR operation is then used to introduce the new bits into the desired positions. For example, if 1010 is placed into the low order four bits of A, then the possible positions on different variants are summarized in Table 4.11.

Table 4.11 Logic OR Microoperation

1101 1000 1011 0001	$A \rightarrow R$	A (Original)
1101 1000 1011 1010		A (Desired)
1101 1000 1011 0001		A (Original)
1111 1111 1111 0000		Mask
1101 1000 1011 0000	$(A \oplus M \rightarrow A)$	A (Intermediate)
0000 0000 0000 1010	$I + A \oplus M$	Added Bits
1101 1000 1011 1010		A (Desired)

Shift Microoperations

Shift microoperations are associated with serial transfer of data. These are specifically defined in conjunction with arithmetic, logic and other data-processing operations. Depending on the case, the contents of a register can be shifted to the left or the right. First the bits are shifted and the first flip-flop receives its binary information from the serial input. During a shift-left operation the input bits are transferred in serial to the rightmost position while during a shift-right operation the input bits are serially transferred to the leftmost position. The process of information transferred through the serial input determines the type of shift operation.

There are six types of shift microoperations that are used to shift the bits of a register. The shift register that shifts the bits of a register one place left is **shl**, one place right is **shr**; one place left with the leftmost bit being circled back to the right is **cil** and **cir** which functions similarly for the right. **ashl** shifts all the bits except the sign bit of a register to the left but not into the sign bit and **ashr** which shifts all bits excluding the sign bit to the right.

The shift that transfers 0 through the serial input A is called logical shift. The symbols used are **shl** and **shr** for logical shift-left and shift-right microoperations, respectively.

For example,

R1 shl R1 specifies a 1-bit shift to the left of the content of the register.

R2 shr R2 specifies a 1-bit shift to the right of the content of the register.

cil and **cir** are the register symbols.

There are three types of shifts. They are as follows:

1. Logical Shift

During a logical shift the bit is transferred to the end position through the serial input and is assumed to be 0. In logical shift, the data entering by serial input to left most or right most flip-flop depending on right or left shift operations, respectively is 0.

2. Circular Shift

A circular shift is a rotate operation that circulates the bits of the register without losing information around the two ends. This is performed by connecting the serial output of the shift register to its serial input. The symbols **cil** and **cir** are used for the circular shift left and right operations, respectively.

Table 4.12 summarizes the shift designation, microoperations symbolic representation and description.

Table 4.12 Shift Microoperations and Symbolic Representation

Microoperations Symbolic Representation	Description
R<-shl R	Shift-Left Register R
R<-shr R	Shift-Right Register R
R<-cil R	Circular Shift-Left R
R<-cir R	Circular Shift-Right R
R<-ashl R	Arithmetic Shift-Left R
R<-ashr R	Arithmetic Shift-Right R

3. Arithmetic Shift

An arithmetic shift is a shift microoperation which is used to shift a signed binary number to the left or right. Arithmetic shifts must leave the sign bit unchanged because the sign of the number cannot change when it is multiplied or divided by 2.

NOTES

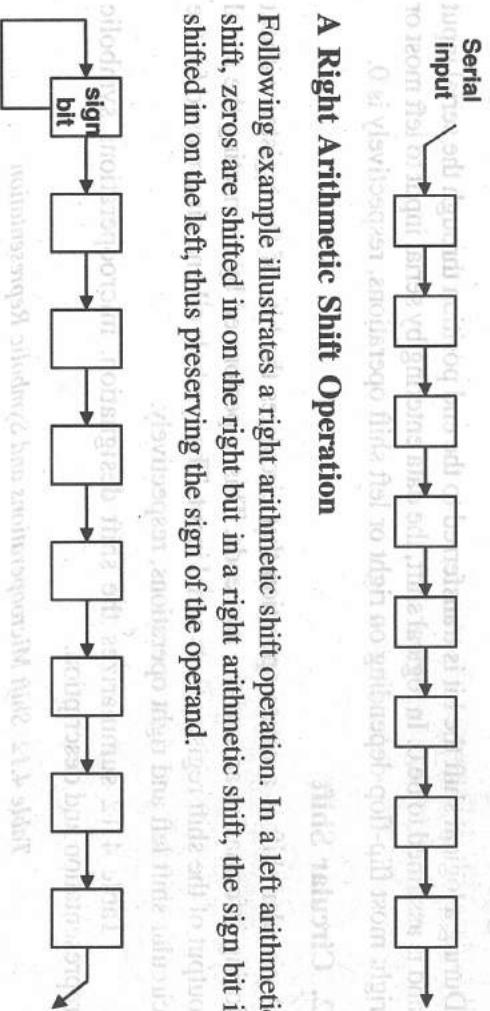
- Arithmetic shift is used for signed binary numbers (integer).
- Arithmetic left-shift is used to multiply a signed number by two.
- Arithmetic right-shift is used to divide a signed number by two.

NOTES

Information goes into shift microoperation through serial input. The main characteristics of shift microoperation are as follows:

A Right-Shift Operation

Following example illustrates a right-shift operation. In right-shift operation, binary numbers are used to shift all of the bits of its operand and every bit in the operand is moved a given number of bit positions and the vacant bit-positions are filled in. Hence, instead of being filled with all 0s, when shifting to the right, the leftmost bit, the sign bit in signed integer representations, is replicated to fill in all the empty positions as it is logical shift.

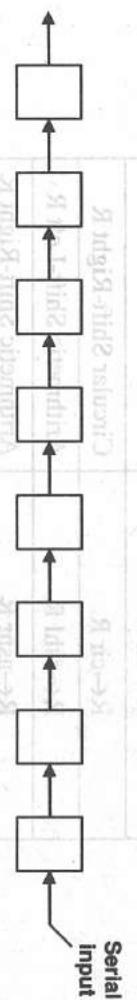


A Right Arithmetic Shift Operation

Following example illustrates a right arithmetic shift operation. In a left arithmetic shift, zeros are shifted in on the right but in a right arithmetic shift, the sign bit is shifted in on the left, thus preserving the sign of the operand.

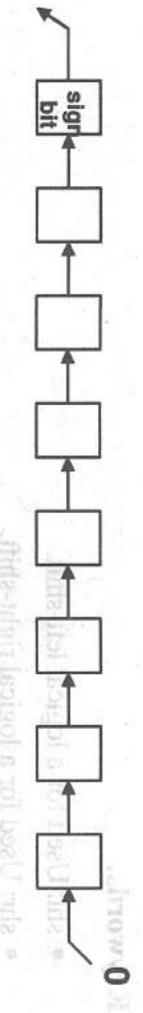
A Left-Shift Operation

Following example illustrates a left-shift operation. This operation is used to shift left logical of one position moves each bit to the left by one. The low-order bit (the right-most bit) is replaced by a zero bit and the high-order bit (the left-most bit) is discarded.



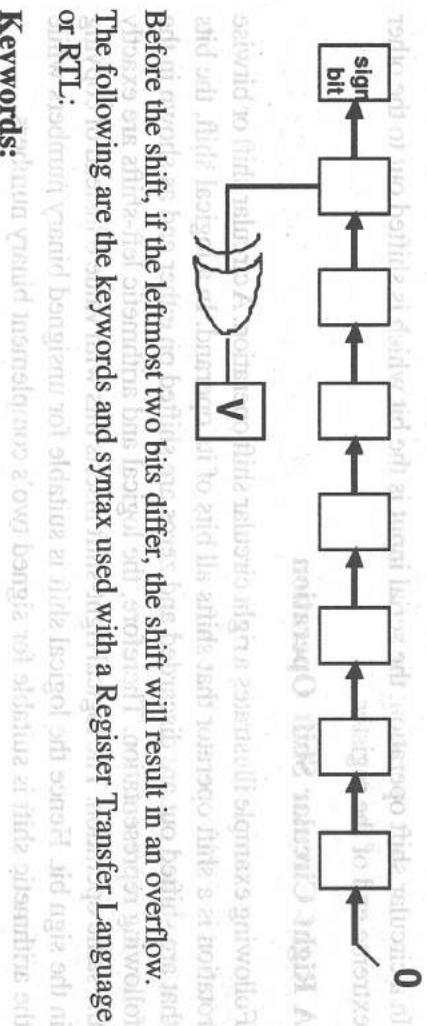
A Left Arithmetic Shift Operation

Following example illustrates a left arithmetic shift operation. The shift arithmetic right or 'sar' and shift logical right 'srl' instructions shift the bits of the destination operand to the right toward less significant bit locations. For each shift count, the least significant bit of the destination operand is shifted into the CF or Carry Flag and the most significant bit is either set or cleared depending on the instruction type.



NOTES

Following example illustrates a left arithmetic shift operation which is checked if any overflow occurs. A left arithmetic shift operation must be checked for the overflow. Before the shift, if the leftmost two bits differ, the shift will result in an overflow.



Before the shift, if the leftmost two bits differ, the shift will result in an overflow.

The following are the keywords and syntax used with a Register Transfer Language or RTL:

Keywords:

- **ashl:** Used for an arithmetic left-shift.
- **ashr:** Used for an arithmetic right-shift.

Syntax:

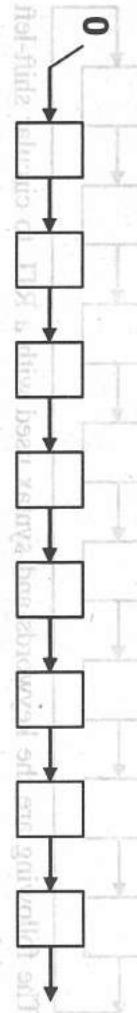
```
R2 ← ashl R2
R3 ← ashl R3
```

Logical Shift

In a logical shift, the serial input given to the right or left shift is zero (0).

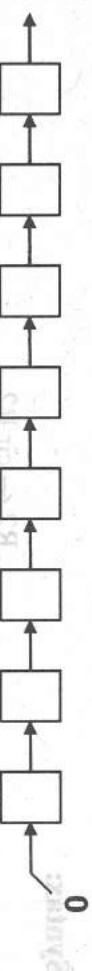
A Right Logical Shift Operation

Following example illustrates a right logical shift operation.



A Left Logical Shift Operation

Following example illustrates a left logical shift operation. Let us take an example of B-bit. A B-bit shift left logical operation performs a B-bit left shift and sets the lower B bits of the result to zeros.



The following are the keywords and syntax used with a RTL to shift left or right.

Keyword..

- shl: Used for a logical left-shift.
- shr: Used for a logical right-shift.

NOTES

Syntax:

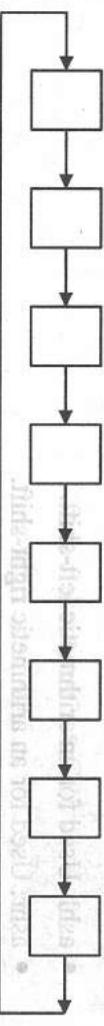
```
shl r1 by n into r2
shr r1 by n into r2
```

Circular Shift

In a circular shift operation, the serial input is the bit which is shifted out to the other extreme end of the register.

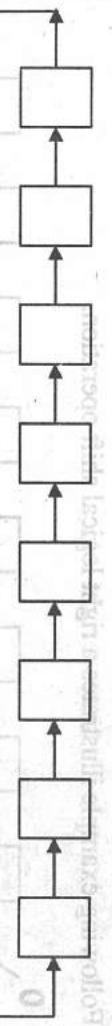
A Right Circular Shift Operation

Following example illustrates a right circular shift operation. A circular shift or bitwise rotation is a shift operator that shifts all bits of its operand. In a logical shift, the bits that are shifted out are discarded and zeros are shifted on either end as shown in the following representation. Therefore, the logical and arithmetic left-shifts are exactly the same operation. The logical right-shift inserts bits with value 0 instead of copying in the sign bit. Hence the logical shift is suitable for unsigned binary numbers while the arithmetic shift is suitable for signed two's complement binary numbers.



A Left Circular Shift Operation

Following example illustrates a left circular shift operation. The circular shift or bit rotation is a form of shift. In this operation, the bits are rotated as if the left and right ends of the register have been joined. The value that is shifted on the right position during a left-shift operation will be shifted out on the left and vice versa. This operation is helpful if it is necessary to retain all the existing bits. Therefore, circular shift is frequently used in digital cryptography.



The following are the keywords and syntax used with a RTL to circular shift-left or right.

Keywords:

- cil: Used for a circular left-shift.
- cir: Used for a circular right-shift.

Syntax:

```
cil r1 by n into r2
cir r1 by n into r2
```

Hardware Implementation of Shift Microoperations

Information is transferred to the registers connected in parallel and then shifted to right or left depending on the condition specified in the syntax. A clock pulse is required to load the data into the register. Shift is initiated using another pulse. The content of a register is first shifted onto a common bus which is connected to the combinational shifter for output operation and then the shifted number is loaded back into the register.

Only one clock pulse is needed for loading the shifted value into the register. Figure 4.18 illustrates hardware implementation of shift microoperations. As far as implementation of shift microoperation is concerned, it can be implemented by a left-right shift registers having parallel load capabilities.

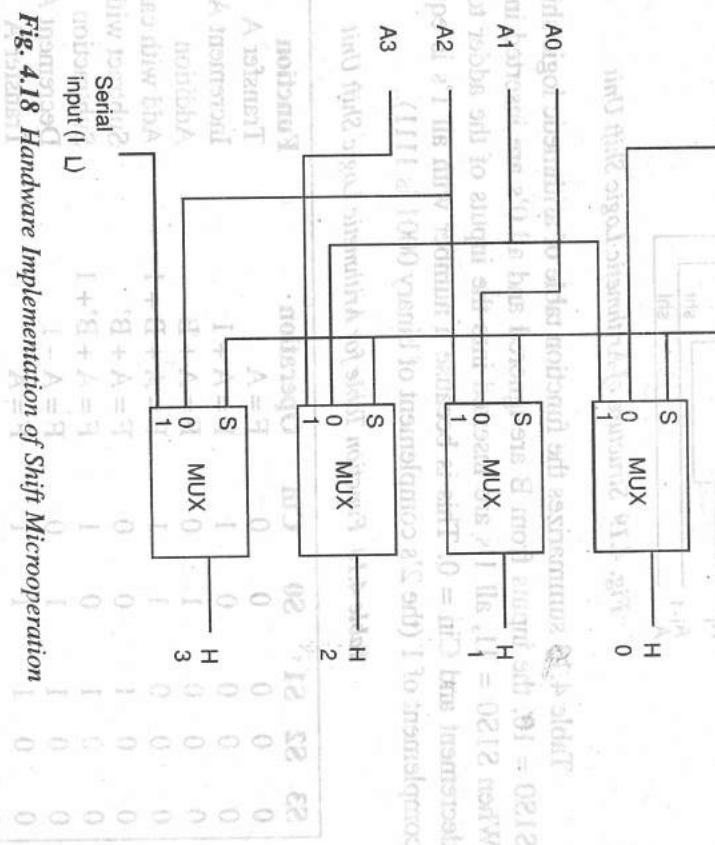


Fig. 4.18 Hardware Implementation of Shift Microoperation

Arithmetic Logic Shift Unit

Figure 4.19 illustrates the Arithmetic Logic Shift Unit or ALSU. It is a combination circuit that performs a number of arithmetic, logic and shift microoperations. In ALSU, an input multiplexer circuit comprising a plurality of bit position multiplexers, each bit position multiplexer having a set of data inputs for receiving the state of operand bit positions where each of a most significant group of the bit position multiplexers also has a data input for receiving a sign bit where each of a selected group of the bit position multiplexers also has a data input for receiving a carry bit and where each bit position multiplexer has an output for presenting a signal corresponding to a selected one of its data inputs responsive to one of a set of multiplexer control signals. If we combine all the shift registers then a simple structure of ALU is constructed. In effect, ALU is a combinational circuit whose inputs are contents of specific registers. The ALU performs the desired microoperation as determined by control signals on the input and places the results in an output or destination register. The whole operation

NOTES

of ALU can be performed in a separate unit but sometimes it can be made as a part of overall ALU. ALU has two data inputs; the i th bits of registers to be manipulated. The i -th or $i+1$ th bit is also fed for the case of shift microoperation of only one register.

NOTES

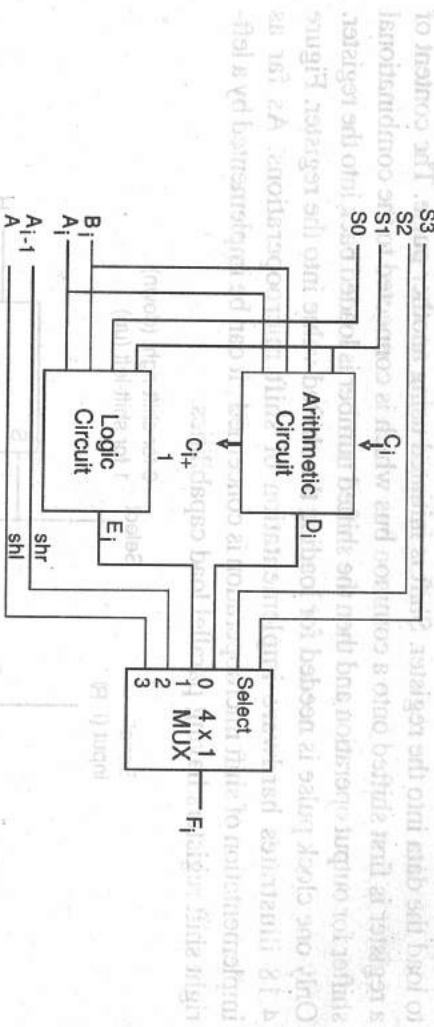


Fig. 4.19 Structure of Arithmetic Logic Shift Unit

Table 4.13 summarizes the function table of arithmetic logic shift unit. When $S1S0 = 10$, the inputs from B are ignored and all 0's are inserted into the inputs. When $S1S0 = 11$, all 1's are inserted into the inputs of the adder to produce the decrement and $Cin = 0$. This is because a number with all 1's is equal to the 2's complement of 1 (the 2's complement of binary 0001 is 1111).

Table 4.13 Function Table for Arithmetic Logic Shift Unit

S3	S2	S1	S0	Cin	Operation	Function
0	0	0	0	0	$F = A$	Transfer A
0	0	0	0	1	$F = A + 1$	Increment A
0	0	0	1	0	$F = A + B$	Addition
0	0	0	1	1	$F = A + B + 1$	Add with carry
0	0	1	0	0	$F = A + B'$	Subtract with borrow
0	0	1	0	1	$F = A + B' + 1$	Subtraction
0	0	1	0	0	$F = A - 1$	Decrement A
0	0	1	1	0	$F = A$	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	$F = A'$	Complement A
1	0	X	X	X	$F = shr A$	Shift-right A into F
1	1	X	X	X	$F = shl A$	Shift-left A into F

4.3.1 Data Manipulation Instructions

Data manipulation instructions perform operations on data and provide computational capabilities for the computer. Data manipulation is basically of the following three types:

- Arithmetic instructions.
- Logical and bit manipulation instructions.
- Shift instructions.

All these instructions involve fetch phase in which data is read in binary format from memory. The operands are brought to the processor register from where they go to the ALU where the desired operation is performed.

Let us discuss each of these three types of instructions in detail.

Arithmetic Instructions

Arithmetic instructions are used to perform operations on numerical data. There are four basic arithmetic operations: addition, subtraction, division and multiplication. Many computers have hardware that supports these systems. However, some processors just have addition and subtraction circuits. In such processors, multiplication and division are done through repeated additions and repeated subtractions, respectively using software subroutine. These four basic operations are sufficient for any scientific calculations. Some typical arithmetic operations are given in Table 4.14.

Table 4.14 Common Arithmetic Operations

S.No.	Name	Mnemonic	Action taken
1	Increment	INC	Add 1 to value stored in register or memory word.
2	Decrement	DEC	Subtract 1 from the value stored in register or memory word.
3	Add	ADD	Add content of two register or data in memory location.
4	Subtract	SUB	Subtract content of two registers or data in memory location.
5	Multiply	MUL	Multiply content of two registers or data in memory location.
6	Divide	DIV	Divide content of two registers or data in memory location; add content of two registers.
7	Add with carry	ADDC	Add with carry forwarded to it in content of two registers or data in memory location.
8	Subtract with borrow	SUBB	Subtract with carry forward; borrow from one register to another or from one memory location to another.
9	Negate (2's complement)	NEG	Find the negative of a number using 2's complement representation.
10	Absolute	ABS	Find absolute value of number.

Add, subtract, multiply and divide operations are executed differently for different types of data, for example, floating number addition involves finding the number with higher exponent value and changing the mantissa value of the other number so that exponents of both are same. Add the two mantissas and then normalize the exponent in case the integer type of data involves just the addition of two numbers.

Logical and Bit Manipulation Instructions

Logical and bit manipulation instructions are used to perform Boolean operations on non-numerical data. These operations are performed on the strings of bits stored in registers. Logical operations are especially useful for comparing two operands and making logical decision. Logical microoperations are useful in bit manipulation of binary data, i.e., these are used for manipulating individual bits or a portion of a word stored in a register. This is because the logical operations consider each bit of operand

NOTES

NOTES

separately and treat it as a Boolean variable. They can be used to change bit values, delete a group of bits or insert new bit values into the register. The three basic logical operations are: AND, NOT (complement) and OR. Other logical operations can be performed combining these basic operations. The complement microoperation is the same as 1's complement which makes all 0 1 and all 1 0. It is represented by putting bar on top of the symbol that denotes the register name. There are 16 different logical operations which can be performed with two binary variables.

The bits manipulation operations primarily involve three actions: selected bit can be cleared to 0, selected bit is set to 1, or selected bit is complemented. A variety of bit manipulation operations, such as selective set, selective complement, selective clear, (which may include masking operation or inserting bits, etc.), are possible by combining these three elementary operations. Table 4.15 summarizes the presents important fundamental logical operations and action taken in each of them.

Table 4.15 Common Logical Operations

S.No.	Name	Mnemonic	Action taken
1	Clear	CLR	Replaces the content of registers by 0.
2	Complement	COM	Produces 1's complement of data in register bit by inverting all bits of operand.
3	AND	AND	Performs logical AND operation bits stored on two registers.
4	OR	OR	Performs logical OR operation bits stored on two registers.
5	Exclusive -OR	XOR	Performs logical XOR operation bits stored on two registers.
6	Clear carry	CLRC	Sets carry bit to 0.
7	Complement carry	COMC	Complements the carry bit to 0.
8	Set carry	SETC	Sets the carry bit to 1.
9	Enable interrupt	EI	Flip-flops that control the interrupt is enabled
10	Disable interrupt	DI	Flip-flops that control the interrupt is disabled

Shift Instructions

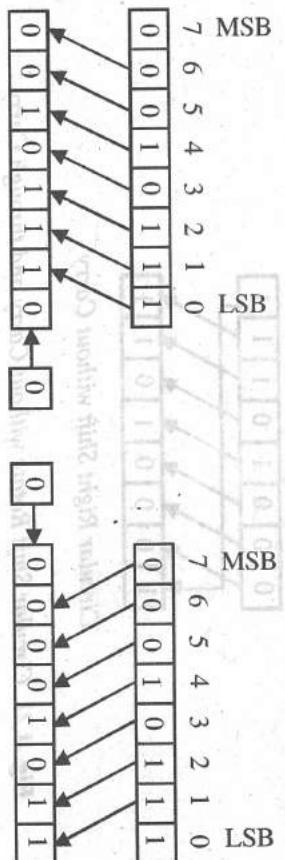
The shift operation is used to shift the content of an operand (data in register or memory location) to one or more bits to provide necessary variation. Shift microoperations are used for serial transfer of data. As we know that registers are collections of flip-flops, the contents of the register (flip-flops) can be shifted to both sides: left or right. There are three types of shift operations, namely logical, circular and arithmetic operations, which are used for manipulating the contents of registers. The input bit determines which type of shift operation is to be executed. Data that is shifted off the end of the register or memory location is either shifted into a flag

register, which can be used to set a condition flag or is dropped depending how the instruction is implemented.

The first flip-flop receives the serial input, and instantaneously the bits in the register are shifted, left or right.

- During the shift left operation, the serial input transfers a bit into the rightmost position and shifts each bit to adjacent left bit.
- During the shift right operation, the serial input transfers a bit into the leftmost position and shifts each bit to adjacent right bit.

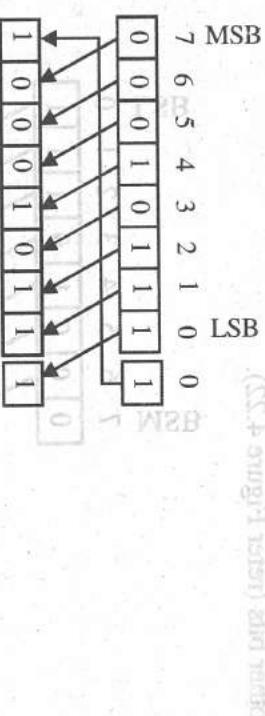
Logical Shift Operations: In logical shift, operation 0 is transferred as the serial input. It can be right or left, depending on whether 0 is entered through the least significant bit or through the most significant bit, respectively (refer Figure 4.20).



Logical Shift Right Logical Shift Left

Fig. 4.20 8-Bit Logical Shift Register

Circular Shift Operation (Rotate Operation) without Carry: In *circular shift* or *bit rotation*, the bits are ‘rotated’ as if the left and right ends of the register were joined, i.e., the register is just like a circular array. This operation just circulates the bits of the register. You can see in Figure 4.21 that the last element becomes first and all other bits are just shifted right. Thus, in this operation, all existing bits are retained after the operation; only their positions are changed. Although after this operation, the contents of a register will be different, the relative positions of 0’s and 1’s remain the same. As this operation retains all the existing bits, it is frequently used in digital cryptography. This is accomplished by connecting the serial output of the shift register to its serial input.



Right Rotate through Carry



NOTES

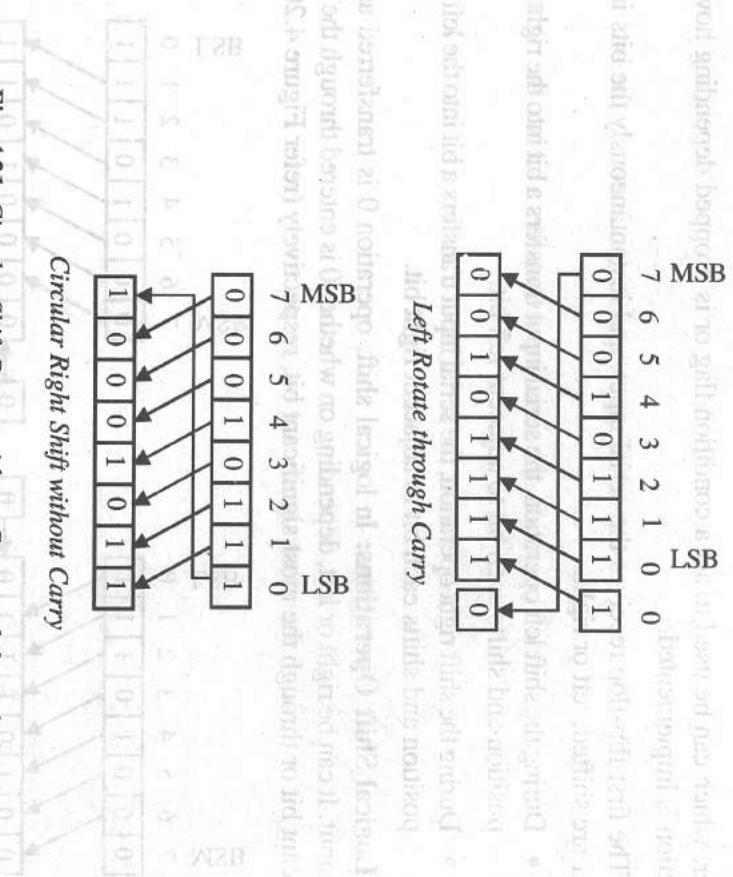


Fig. 4.21 Circular Shift Rotate without Carry and through Carry

Circular Shift Operations (Rotate Operation) through Carry: Rotate through carry is similar to the rotate without carry operations. The only difference is that in the former, the two ends of the register are considered to be separated by the carry flag. Thus, in this case, the bit that is shifted in is the old value of the carry flag, and the bit that is shifted out will become the new value of the carry flag.

Arithmetic Shift Operations: The arithmetic shift assumes that the data being shifted is an integer in nature. Hence, in the result, the sign bit is not shifted, maintaining the arithmetic sign of the shifted result. In an arithmetic shift, the bits that are shifted out of either end are discarded. In the case of a right arithmetic shift, the sign bit values are shifted to the right. However, in the case of a left arithmetic shift, the sign bit values are shifted to the left. shift to the right is called arithmetic shift because it maintains the sign bit while shifting out the other bits. shift to the left is called arithmetic shift because it inserts a 0 into the least significant bit (LSB) and shifts all other bits.

An arithmetic shift left is equivalent to multiplication of a signed binary number by 2. An arithmetic shift left inserts a 0 into Least Significant Bit (LSB) and shifts all other bits (refer Figure 4.22).

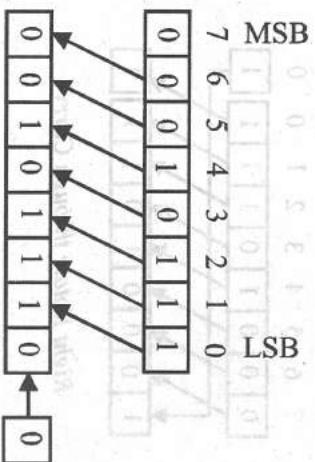


Fig. 4.22 Arithmetic Shift Left

NOTES

Similarly, an arithmetic shift right operation is equivalent to division of the number by 2. The arithmetic shift right leaves the sign bit unchanged and shifts the number, including the sign bit, to the right. The bit in LSB position is lost (refer Figure 4.23).

Fig. 4.23 Arithmetic Shift Right

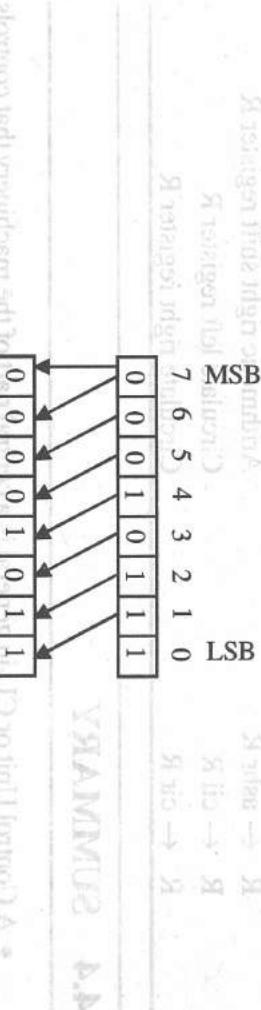


Fig. 4.23 Arithmetic Shift Right

When we use integer data, i.e., the arithmetic operations on signed numbers, it is possible that the magnitude of a result exceeds the number of bits assigned to represent the magnitude. In arithmetic left shift, the initial sign bit of R_{n-1} is lost and is replaced by bit at R_{n-2} . If R_{n-1} and R_{n-2} have different values, the sign reversal will occur. In such a case, the result changes the sign bit, making the result incorrect. We say it is an overflow condition. An overflow flip-flop, Vs, can be used to detect an arithmetic shift-left overflow. If an overflow occurs after an arithmetic shift left will be observed only if initially, before the shift, R_{n-1} is not equal to R_{n-2} .

$$Vs = R_{n-1} \text{ XOR } R_{n-2}$$

If $Vs = 0$, there is no overflow; but if $Vs = 1$, there is an overflow and a sign reversal after the shift. The Vs must be transferred into the overflow flip-flop with the same clock pulse that shifts the register.

It can be observed that the logical and arithmetic left-shifts are exactly the same operation. The only difference is that the logical right-shift inserts bits with value 0, while the arithmetic shift copies the sign bit. Hence, the logical shift is suitably used for unsigned binary numbers, while the arithmetic shift is suitable for the signed two's complement binary numbers. Table 4.16 summarizes the list of common shift operations.

Table 4.16 Common Shift Operations

S.No.	Name	Mnemonic	Action Taken
1	Logical shift right	SHR	Logical left right
2	Logical shift left	SHL	Logical left shift
3	Arithmetic shift right	SHRA	Arithmetic right shift
4	Arithmetic shift left	SHRL	Arithmetic left shift
5	Rotate right	ROR	Circular right shift
6	Rotate left	ROL	Circular left shift
7	Rotate right through carry	RORC	Circular right shift with carry
8	Rotate left through carry	ROLC	Circular left shift with carry

NOTES

Check Your Progress

8. Why a clock pulse is required in hardware implementation of shift microoperations?
9. Name the three actions which are involved by the bits manipulation operations.
10. Why shift operation is used?
11. Why overflow flip-flop is used?

Control Unit

Sub-Unit	Description
$R \leftarrow \text{shl } R$	Logical left shift register R
$R \leftarrow \text{shr } R$	Logical right shift register R
$R \leftarrow \text{ashl } R$	Arithmetic left shift register R
$R \leftarrow \text{ashr } R$	Arithmetic right shift register R
$R \leftarrow \text{cir } R$	Circulate left register R
$R \leftarrow \text{crr } R$	Circulate right register R

NOTES

- A Control Unit or CU in general is a central part of the machinery that controls its operation. It is specifically used in the area of computer design. The control unit coordinates the input and output devices of a computer system. It fetches the code of all of the instructions in the microprograms. It directs the operation of bytes of the other units by providing timing and control signals.
- The function of instruction register is to store actual instructions which are loaded from the memory.
- The function of instruction counter is to store memory address of the various instructions and is incremented automatically after every command.
- Typically, in most of the microprogrammed processors, an instruction is fetched from memory which is then interpreted by a microprogram stored in a single Control Memory or CM.

In some microprogrammed processors the microinstructions are not directly used by the decoders for generating control signals but they are used for accessing second control memory termed as nanoControl Memory or nCM.

- The number of bits a CPU uses to represent integer numbers is termed as 'register width', 'word size', 'bit width', 'data path width' or 'integer precision'. This number is considered as one of the most important characteristics of a CPU. Registers are generally measured by the number of bits they can hold, for example, an '8-bit register', a '16-bit register' or a '32-bit register'. A processor typically contains several types of registers and is classified on the basis of their content or instructions that operate on them.
- Control address register is a processor register which is used to change or control the general behaviour of a CPU or other digital device. Common tasks performed by control registers include interrupt control, switching the addressing mode, paging control, and coprocessor control. Control memory is a Random Access Memory or RAM consisting of addressable storage registers.
- The control memory address register specifies the address of the microinstruction. The control data register holds the microinstruction read from memory. The microinstruction contains a control word that specifies one or more microoperations for the data processor.
- Microinstructions are stored in control memory in groups with each group specifying a routine.

4.4 SUMMARY

- In a hardwired control unit, hardware (combinational and sequential circuits) generates the control signals using logic design. These hardware circuits include gates, flip-flops, decoders, multiplexers and other digital circuits.
- 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.
- The microprogram is essentially an interpreter, written in microcode and stored in firmware (ROM, PROM, or EEPROM) which is often referred to as the control memory or control store.

- The fetch cycle includes the fetching of instructions that leads to initiation of a series of microinstructions stored in control memory.
- A computer with a writable control memory is termed as dynamically microprogrammable since it is possible to change the content of the control memory for such computers.

- The control memory refers to a set of words in which each word represents a microinstruction. The computers that have writable microprogrammed control unit use RAM as two separate memories. One is used as the main memory and the other is used as the control memory.
- The execution of every machine instruction is done with the help of a sequence of microinstructions known as a routine. Each computer has its own microprogram routine that is stored in the control memory.
- Microinstructions are responsible for generating control signals from the control unit. For implementing a desired microoperation, these control signals are sent to the desired control lines.
- A control word refers to a set of control signals in which each bit represents a single control line. It can be programmed to perform various operations on the various components of the system.

- 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.
- Arithmetic microoperations perform various arithmetic operations on the data stored in registers. Arithmetic operations can be done only on numerics including numeric subfields, numeric arrays, numeric array elements, numeric table elements, numeric named constants, numeric figurative constants and numeric literals.
- Logic microoperations refer to the bitwise manipulation of the contents of a register. It uses NOT, AND, OR and eXclusive-OR (XOR). They are quite useful for masking bits.

- Shift microoperations are associated with serial transfer of data. These are specifically defined in conjunction with arithmetic, logic and other data

NOTES

(logical) processing operations. Depending on the case, the contents of a register can be shifted to the left or the right.

- Logical and bit manipulation instructions are used to perform Boolean operations on non-numerical data. These operations are especially useful for comparing two operands and making logical decision.

- The bit manipulation operations primarily involve three actions: selected bit can be cleared to 0, selected bit is set to 1 or selected bit is complemented.
- There are three types of shift operations, namely logical, circular and arithmetic operations which are used for manipulating the contents of registers. The input bit determines which type of shift operation to be executed.

4.5 KEY TERMS

- **Control unit:** A central part of the computer system that controls I/O operations
- **Microoperations:** Basic operations on which more complex instructions are built
- **Microcode:** A layer of hardware-level instructions and data structures involved in the implementation of higher level machine code instructions
- **Control buffer register:** A register that stores the binary word
- **Logic microoperation:** Binary operations for strings of bits stored in registers
- **Shift microoperation:** A microoperation used for serial transfer of data
- **Arithmetic shift:** A shift microoperation
- **Set carry:** A logical operation which sets carry bit to 1
- **Magnitude:** A term used for the size or length of a vector
- **Rotate left:** A type of shift operation used in circular left shift of

4.6 ANSWERS TO ‘CHECK YOUR PROGRESS’

1. The advantage of hardwired control is that it can be optimized to produce fast mode of operation.
2. The term micropogram was first coined by M. V. Wilkes in early 1950.
3. The execution of every machine instruction is done with the help of a sequence of microinstructions known as a routine.
4. A control word refers to a set of control signals in which each bit represents a single control line.
5. Control Buffer Register or CBR basically performs three functions:
 - Holding the control word retrieved from control memory,
 - Generating/propagating the control function values to the MCU and
 - Providing the information required for generating the next address.
6. Logic microoperations refer to the bitwise manipulation of the contents of a register. It uses NOT, AND, OR and eXclusive-OR (XOR).

7. A circular shift is a rotate operation that circulates the bits of the register without losing information around the two ends.
8. A clock pulse is required to load the data into the register.
9. The bits manipulation operations primarily involve three actions: selected bit can be cleared to 0, selected bit is set to 1 or selected bit is complemented.
10. The shift operation is used to shift the content of an operand (data in register or memory location) to one or more bits to provide necessary variation.
11. An overflow flip-flop, Vs, can be used to detect an arithmetic shift-left overflow.

NOTES

4.7 QUESTIONS AND EXERCISES

Short-Answer Questions

1. What is nanoprogramming?
2. What does hardware generate in a hardwired control unit?
3. What are microinstructions?
4. What is non-zero microcode?
5. Write the function of upper decoder.
6. Write the symbols which are used to denote OR and AND microoperations.
7. Name the symbols which are used for the shift left and right operations, respectively.
8. Name the four basic arithmetic operations.
9. Why are logical manipulations useful?
10. Name the application where circular shift operation (rotate operation) without carry is used.

Long Answer Questions

1. Explain the control unit and its functions with the help of examples and illustrations.
2. How hardwired implementation is done? Explain.
3. Discuss the microprogrammed control unit and its advantages over a hardwired unit with the help of illustrations.
4. Explain the execution process of microinstructions.
5. Describe the fundamental components of a microprogrammed unit with the help of illustrations.
6. Discuss the arithmetic operations giving suitable examples.
7. Explain the logic microoperations with the help of examples.
8. Explain the types of data manipulation instructions with the help of examples.
9. Discuss the circular shift operation (rotate operation) without carry with the help of illustrations.
10. How arithmetic shift operations are performed? Explain.

4.8 FURTHER READING

- Tanenbaum, Andrew S. 1999. *Structured Computer Organization*, 4th edition. New Jersey: Prentice-Hall Inc.
- Wilkinson. 1996. *Computer Architecture: Design and Performance*, 2nd edition. Hertfordshire: Prentice-Hall Inc.
- Mano, M. Morris. 1993. *Computer System Architecture*, 3rd edition. New Jersey: Prentice-Hall Inc.
- Stallings, William. 2006. *Computer Organization and Architecture*, 7th edition. New Jersey: Prentice-Hall Inc.
- Hanacher, V.C., Z.G. Vranesic and S.G. Zaky. 2002. *Computer Organization*, 5th edition. New York: McGraw-Hill International Edition.
- Conte, T. M. and C. E. Gimarc. 1995. *Fast Simulation of Computer Architecture*. Boston: Kluwer Academic Publishers.
- Gilmore, M. 1996. *Microprocessors: Principles and Applications*, 2nd edition. New York: McGraw-Hill.

NOTES

UNIT 5 PARALLEL ORGANIZATION

Structure

5.0 Introduction

5.1 Unit Objectives

5.2 Microprocessor Organization

5.2.1 Parallel Organization: Basics

5.2.2 Types of Parallel Processor Systems

5.2.3 Flynn's Classification of Computers

5.3 Symmetric Multiprocessors

5.3.1 Organization of Symmetric Multiprocessing

5.4 Clusters

5.4.1 Cluster Configurations

5.4.2 Cluster Computer Architecture

5.5 Summary

5.6 Key Terms

5.7 Answers to 'Check Your Progress'

5.8 Questions and Exercises

5.9 Further Reading

5.0 INTRODUCTION

In the previous unit, you learnt about the control unit. In this unit, you will learn about parallel organization. Microprocessor is a program controlled semiconductor device which fetch, decode and execute instructions. The basic units or blocks of a microprocessor are ALU, an array of registers and control unit. The function of microprocessor is to control all the activity of the system. It issues address, control signals, and fetches the instruction and data from memory. Then, it executes the instruction for further processing. Flynn's taxonomy is a classification of computer architectures proposed by Michael J. Flynn. The four classifications defined by Flynn are based upon the number of concurrent instruction and data streams available in the architecture are Single Instruction, Single Data stream or SISD, Single Instruction, Multiple Data streams or SIMD, Multiple Instruction, Single Data stream or MISD and Multiple Instruction, Multiple Data streams or MIMD. SISD is a sequential computer which does not support parallelism in either the instruction or data streams. Single control unit fetches single instruction stream from the memory. The CU or Control Unit then generates appropriate control signals to direct single processing element to operate on single data stream. SIMD supports multiple data streams against a single instruction stream to perform operations which are parallelized. In MISD, multiple instructions operate on a single data stream. In MIMD, multiple processors simultaneously execute different instructions on different data.

You will also learn about symmetric multiprocessors. In symmetric multiprocessor, the processor bus is used as communication path between the CPU and the main bus. The processor bus is the set of wires used to carry information to and from the processor. Symmetric multiprocessing organization involves a multiprocessor computer hardware architecture where two or more identical processors

NOTES

5.1 UNIT OBJECTIVES

- Microprocessor Organization
- Parallel Organization: Basics
- Types of Parallel Processor Systems
- Flynn's Classification of Computers
- Organization of Symmetric Multiprocessing
- Cluster Configurations
- Cluster Computer Architecture
- Summary
- Key Terms
- Answers to 'Check Your Progress'
- Questions and Exercises
- Further Reading

are connected to a single shared main memory and controlled by a single operating system. Finally, you will learn about clusters. A computer cluster is a group of linked computers which work together closely and hence can be viewed as a single computer. The components of a cluster are commonly connected to each other through local area networks. Clusters are deployed to improve performance and availability over that of a single computer.

NOTES

5.1 UNIT OBJECTIVES

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

- Understand the organization of microprocessor in a computer system
- Explain the basics of parallel organization
- Classify the parallel processor systems
- Describe Flynn's classification of computers
- Explain the significance of symmetric multiprocessors
- Discuss computer cluster and its configuration
- Illustrate cluster computer architecture

5.2 MICROPROCESSOR ORGANIZATION

The microprocessor is a Central Processing Unit (CPU) that is the 'brain' of the computer. It processes the information as per a program and provides output in the form of digital signals. The microprocessor is fabricated on a single chip by using the Metal Oxide Semiconductor (MOS) technology.

The important features of microprocessor are as follows:

- **Cost:** The low cost of microprocessor is considered as the most important characteristic of a microcomputer. It is significant that the cost is not proportional to the complexity rather the cost per function goes on decreasing with the increasing complexity of a chip. With a high volume of production, the cost per unit microprocessor chip. The volume of production is very high because of the prevalent use of microprocessors, so the microprocessor chips are available at comparatively low prices.
- **Size:** The second important feature of a microprocessor is its smaller size. As a consequence of enhancement in fabrication technology, VLSI, electronic circuitry has become so dense that a minute silicon chip can contain several thousands of transistors constituting the microprocessor.
- **Power Consumption:** Low power consumption is another characteristic. Microprocessors are usually manufactured by Metal-Oxide semiconductor technology, which has the characteristic feature of low power consumption. The word micro can be accredited to the low cost, small size and low power consumption of a microprocessor.
- **Versatility:** A microprocessor-based system can be configured for various applications to simplify or alter the software program with the basic hardware which makes it flexible.

- **Reliability:** Another important property of microprocessors is extreme reliability. The failure rate of an IC (Integrated Circuit) is quite uniform at the package level, regardless of its complexity.

The components of microprocessor are register section, one or more Arithmetic and Logic Unit or ALU and a control unit (refer Figure 5.1).

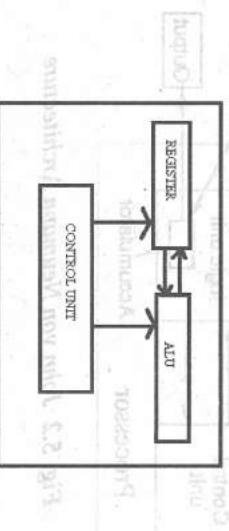


Fig. 5.1 Components of a Microprocessor

Now, let us discuss each of these components in detail.

Register Section

The register section (also known as register set) is related to the main memory of the computer system. During the processing of instructions, it stores the data.

ALU bus and control bus are used to transfer data between the units.

The microprocessor contains the ALU and the control unit for a microcomputer. It is connected to memory and I/O by buses which carry information between the units.

Control Unit

Two main operations of control unit are performed in microprocessor:

- **Instruction Sequencing:** In this operation, the methods by which instructions are selected for execution or the manner in which control of the processor is transferred from one instruction to another.
- **Instruction Interpretation:** In this operation, the method used for activating the control signals that cause the data processing unit to execute the instruction.

Microprocessor Architecture

Following are the basic architectures supported by microprocessor system organization:

John von Neumann Architecture

The design of John von Neumann architecture is simpler than the modern Harvard architecture (discussed below) which is also a stored-program system but has separate dedicated sets of address and data buses for memory and fetching instructions. A stored-program digital computer holds programmed instructions and data in read-write RAM. Stored-program computers were advancement over the program-controlled computers, such as the Colossus and the Electronic Numerical Integrator And Computer or ENIAC. These computers were programmed by setting switches and inserting patch leads to route data and to control signals between various functional units. Figure 5.2 illustrates John von Neumann architecture.

NOTES

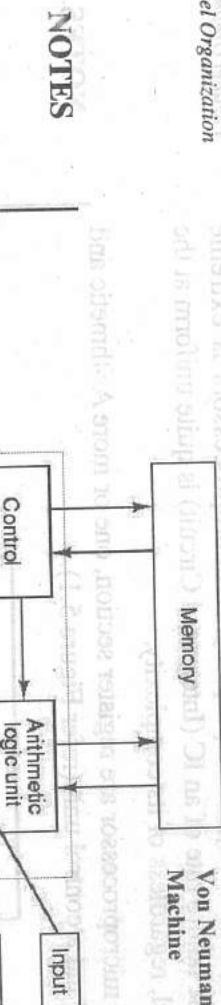


Fig. 5.2 John von Neumann Architecture

Harvard Architecture

The Harvard architecture is computer architecture with physically separate storage and signal pathways for instructions and data. The term 'Harvard architecture' originated from the Harvard Mark I relay-based computer which stored instructions on punched tape (24 bits wide) and data in electro-mechanical counters. Figure 5.3 illustrates the Harvard architecture. In the Harvard architecture computer, the CPU fetched the next instruction and loaded or stored data simultaneously. This is, in contrast, to a Von Neumann architecture computer, in which both instructions and data are stored in the same memory system and (without the complexity of a cache) must be accessed in turn. The physical separation of instruction and data memory is used to be the distinguishing feature of modern Harvard architecture computers.

HARVARD ARCHITECTURE

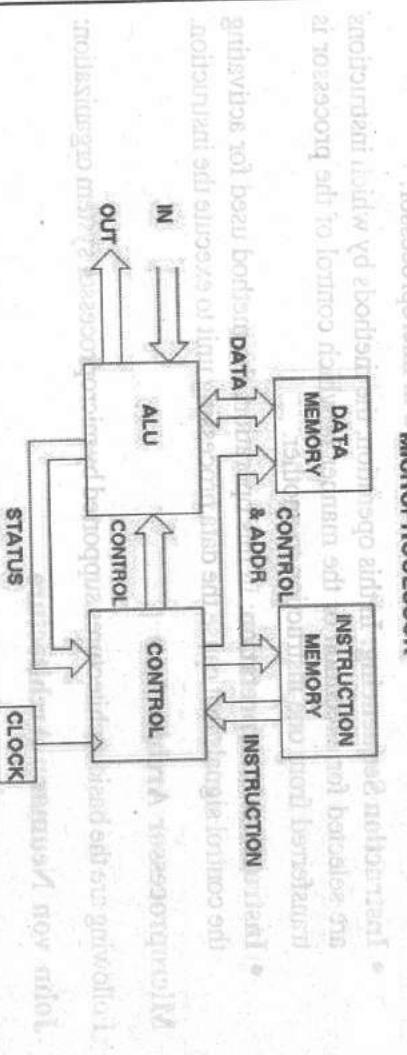


Fig. 5.3 Harvard Architecture

Classification of Processors Based on Register Section

On the basis of register section, the processors can be classified as follows:

Accumulator-Based Microprocessors

One of the most often used parts of a microprocessor is the accumulator. The accumulator is the storage space which often has its content altered. In microprocessors,

one of the operands holds a special register called ‘accumulator’. The arithmetic and logical operations performed by using this accumulator alter the value that it contains. Usually, the result of an operation is also placed in the accumulator. For example, you can add the contents of the accumulator to the contents of a memory location. Figure 5.4 illustrates this operation.

NOTES

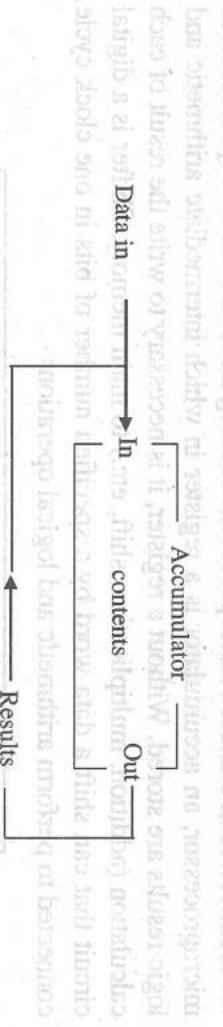


Fig. 5.4 Accumulator Operation

The microprocessor can take the contents of the accumulator as the data coming in, perform some operation and place the result back in the accumulator. Occasionally, there can be a condition when no data is coming in but some operation is still being performed on the contents of the accumulator only. For example, the microprocessor might find the 1's complement of the contents of the accumulator and place the result in the accumulator in place of the original number. Some microprocessors have only one accumulator while others have more than one.

Such one-operand instructions are predominant in several organizations. Intel 8085 and Motorola 6809 are the examples of accumulator-based microprocessors.

The General-Purpose Register-Based Microprocessors

These processors have a set of registers, which contain data, memory addresses and the results of an arithmetic or logic operations for an indefinite time. The number and size of these registers vary from processor to processor.

In microprocessors, general-purpose registers are temporary storage locations. They store addresses or data and are capable of manipulating data by shift or rotate operations. General-purpose registers are similar to the accumulator. However, they differ from the accumulator in operations that involve two pieces of data that are usually not performed in them with the results going back into the register itself, as in the case of the accumulator. The microprocessor will often alter the contents of a register. Examples of such microprocessors are Intel 8086/80386, Motorola 68000/68020. Figure 5.5 shows the operations of a general-purpose register.

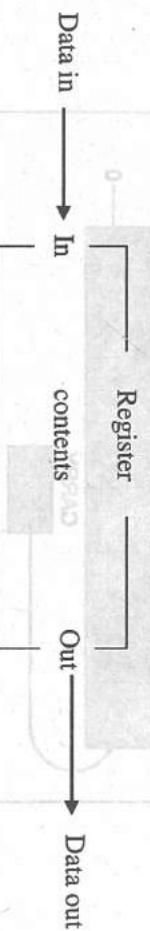


Fig. 5.5 General-purpose Register Operations

One might wonder why a microprocessor needs general-purpose registers when it has RAM to temporarily store information. The answer is speed. Data in registers can be accessed and moved much more quickly than data in RAM.

Implementation of Microprocessor Organization

The processor is considered as 'the brain' of the computer. Its execution speed depends on its frequency (measured in MHz) but two processors from different manufacturers may have almost similar performance but different frequencies. Figure 5.6 illustrates the architecture of standard microprocessor of Zilog Z80, an 8-bit microprocessor. In microprocessor, an accumulator is a register in which intermediate arithmetic and logic results are stored. Without a register, it is necessary to write the result of each calculation (addition, multiplication, shift, etc.) to main memory. Shifter is a digital circuit that can shift a data word by a specified number of bits in one clock cycle connected to perform arithmetic and logical operations.

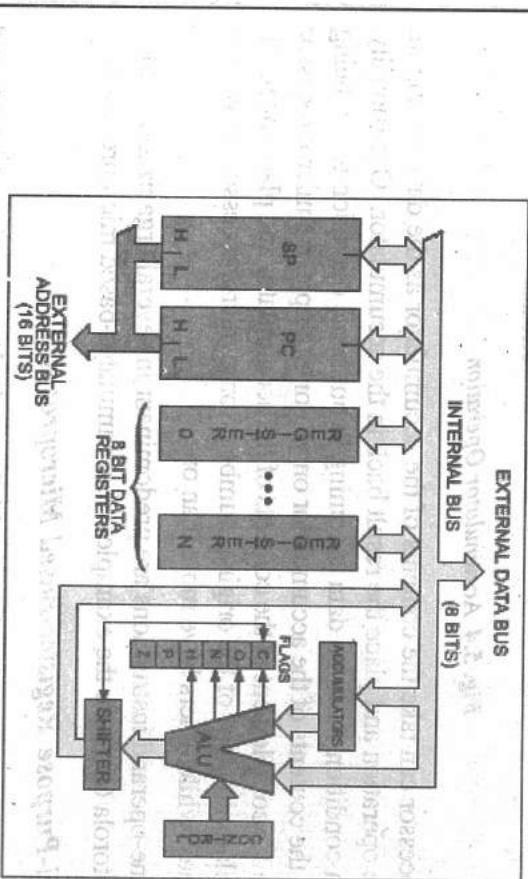


Fig. 5.6 Standard Microprocessor Architecture

In Figure 5.6, the control option on the right represents the control unit which synchronizes the entire system. The ALU performs arithmetic and logic operations. A special register is used to control one of the inputs of the ALU. It is known as an accumulator. The accumulator may be referenced as input and output (source and destination) within the same instruction. The ALU must also provide shift and rotate facilities. A shift operation consists of moving the contents of a byte by one or more positions to the left or to the right. This feature is illustrated in Figure 5.7 where each bit has been moved to the left by one position.

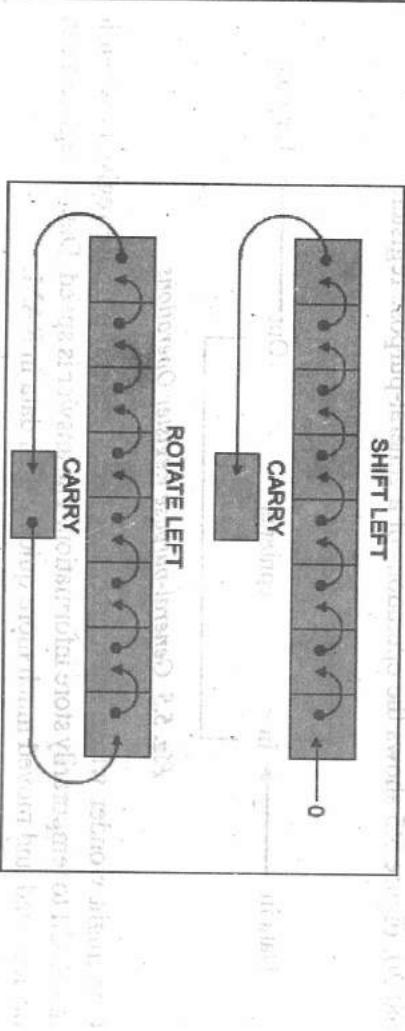


Fig. 5.7 Shift and Rotate Operation

Note: Some 'Shift and Rotate' instructions do not include the Carry flag. Instructions

Parallel Organization

The shifter is implemented on the ALU output. The role of shifter is to store exceptional conditions within the microprocessor. The contents of the flags registers are tested by specialized instructions and read on the internal data bus. A conditional instruction will cause the execution of a new program depending on the value of one of these bits.

Most of the instructions executed by the processor will modify some or all of the flags. Following registers are used in microprocessor organization:

- **General-Purpose Registers:** General-purpose registers must be provided in the ALU to manipulate data at high speed. Because of restrictions on the number of bits which is reasonable to provide within an instruction, the number of 8-bit (directly addressable) registers is usually limited to fewer than eight. Each of all of these registers is a set of eight flip-flops, connected to the bidirectional internal data bus. These eight bits can be transferred simultaneously to or from the data bus. The implementation of these registers in MOS flip-flops provides the fastest level of memory. Internal registers are usually labeled from 0 to n. They may not contain any data used by the program. These general-purpose registers will normally be used to store 8-bit data. When microprocessor is used to manipulate eight two of these registers at a time, then it is called 'register pairs'. This arrangement facilitates the storage of 16-bit data.

• **Address Registers:** Address registers are 16-bit registers intended for the storage of addresses. They are also often called data counters or pointers. They are double registers, i.e., two eight-bit registers. Their essential characteristic is to be connected to the address bus. The address registers create the address of the bus. The address bus appears on the left and the bottom part of the illustration in Figure 5.8. The only way to load the contents of these 16-bit registers is via the data bus. Two transfers will be necessary along the data bus in order to transfer 16 bits. In order to differentiate between the lower half and the higher half of each register, they are usually labelled as L (Low) or H (High), denoting bits 0 through 7 and 8 through 15, respectively (refer Figure 5.8). This label is used whenever it is necessary to differentiate the halves of these registers. At least two address registers are present within most microprocessors. 'MUX' stands for multiplexer.

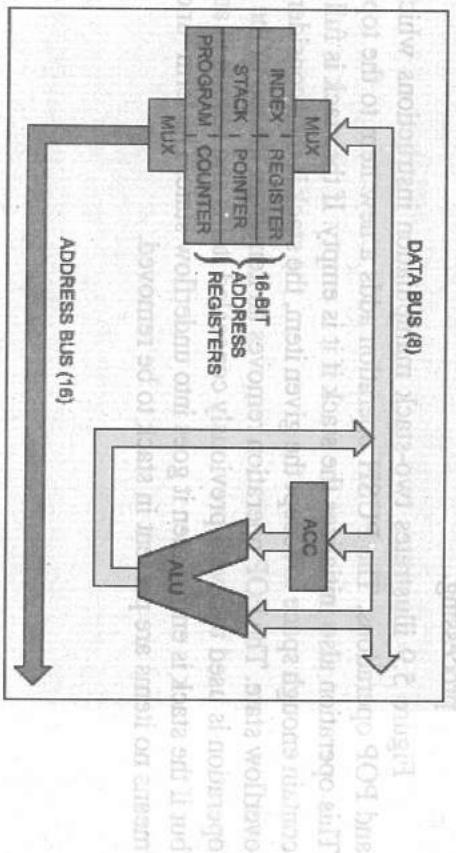


Fig. 5.8 16-Bit Address Registers Create the Address Bus

NOTES

- **Special Purpose Registers:** It carries out a specific control or data handling task. They hold program state and usually includes the following:

NOTES

- o **Program Counter:** The program counter must be present in any processor. It contains the address of the next instruction to be executed. Execution of a program is normally sequential. In order to access the next instruction, it is necessary to bring (queue) it from the memory into the microprocessor. The contents of the PC will be deposited on the address bus and transmitted towards the memory. The memory will then read the contents specified by this address and send back the corresponding word to the microprocessor unit.
 - o **Stack Pointer:** In general-purpose microprocessors, the stack is implemented in 'software', i.e., the memory. In order to keep track of the top of this stack within the memory, a 16-bit register is dedicated to the Stack Pointer or SP. The SP contains the address of the top of the stack within the memory.
 - o **Index Register:** Indexing is a memory-addressing facility used in microprocessors. In fact, it is a facility for accessing blocks of data in the memory with a single instruction. An index register will typically contain a displacement which will be automatically added to a base or it might contain a base which would be added to a displacement. In essence, indexing is used to access any word within a block of data.
 - **Stack:** A stack is formally called a Last-In, First-Out or LIFO structure. A stack is a set memory locations allocated to this data structure. The essential characteristic of this structure is that it is set in a chronological structure. This first element introduced into the stack is always at the bottom of the stack. The element most recently deposited in the stack is on top of the stack. The availability of a stack is required to implement three programming facilities within the computer system: subroutines, interrupts and temporary data storage. A stack may be implemented in the following ways:
 - o A fixed number of registers may be provided within the microprocessor itself. This is a 'hardware stack.' It has the advantage of high speed.
 - o Most general-purpose microprocessors choose the software stack for data processing.
- Figure 5.9 illustrates two-stack manipulation instructions which use PUSH and POP operations. The PUSH operation adds a new item to the top of the stack. This operation also initializes the stack if it is empty. If the stack is full and does not contain enough space to accept the given item, the stack is then considered to be in an overflow state. The POP operation removes an item from the top of the stack. A POP operation is used to reveal previously concealed items and results in an empty stack but if the stack is empty then it goes into underflow state. The term 'underflow state' means no items are present in stack to be removed.

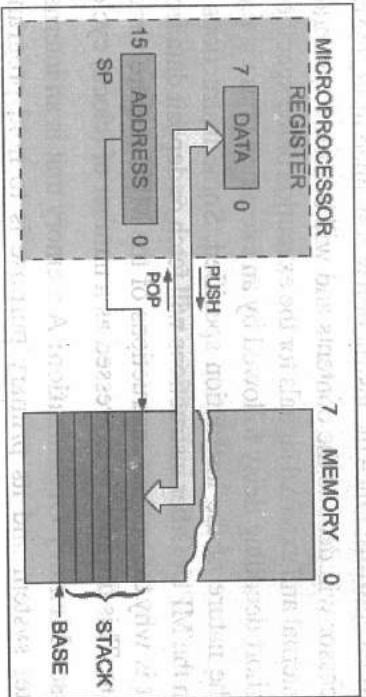


Fig. 5.9 Two Stack-Manipulation Instructions

Instruction Execution Cycle

The microprocessor unit appears on the left, and the memory unit appears on the right (refer Figure 5.10). The memory chip may be a Read Only Memory (ROM) or a Random Access Memory (RAM) or any other chip which contains memory. The memory is used to store instructions and data. Here, one instruction is fetched from the memory for the program counter. It is assumed that the program counter has valid contents. It now holds a 16-bit address which is the address of the next instruction to fetch in the memory. Every microprocessor instruction proceeds through following cycles:

Fetch

In the fetch cycle, the contents of the program counter are stored on the address bus and gated to the memory (on the address bus). Simultaneously, a read signal maybe issued on the control bus of the system, if required. The memory will receive the address. This address is used to specify one location within the memory. Upon receiving the read signal, the memory will decode the address it has received through internal decoders and will select the location specified by the address. In Figure 5.10, this instruction will be stored on the data bus on top of the Micro Processing Unit or MPU.

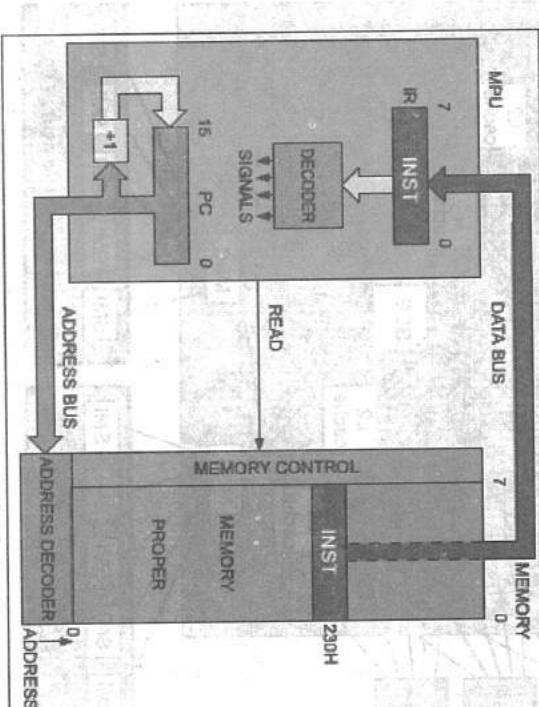


Fig. 5.10 Automatic Sequencing

NOTES

Decoding and Execution: Once the instruction is contained in IR, the control unit of the microprocessor will decode the contents and will be able to generate the correct sequence of internal and external signals for the execution of the specified instruction. Therefore, a short decoding delay followed by an execution phase is required which depends on the nature of the instruction specified. Some instructions will execute entirely within the MPU. Other instructions will fetch or deposit data from or into the memory. That is why, the various instructions of the MPU require various length of time to execute. This duration is expressed as a number of (clock) cycles.

Microprocessor Memory Organization: A memory unit is an internal part of any microcomputer system and its primary purpose is to hold programs and data. Microcomputer memory system can be logically divided into three groups. These groups are known as processor memory, primary or main memory and secondary memory. Processor memory refers to the microprocessor registers. These registers are used to hold temporary results when a computation process is in progress. Also there is no speed disparity between these register and the microprocessor because they are assembled using the same technology. Primary or main memory is the storage area in which all programs are executed. The microprocessor can directly access only those items that are stored in primary memory. Secondary memory refers to the storage medium comprising slow device, such as magnetic tapes and disks. These devices are used to hold large data files and huge programs, such as compilers and database management systems which are not needed by the processor frequently. Secondary memory stores programs and data in excess of the main memory. There are two types of optical disks. These are the Compact Disk Read Only Memory or CD-ROM and the Write-Once, Read-Many or WORM. The CD-ROM is inexpensive compared to the WORM drive. Hard disk memory is also frequently used with microcomputer systems. The hard disk also known as the fixed disk and is not removable like the floppy disk.

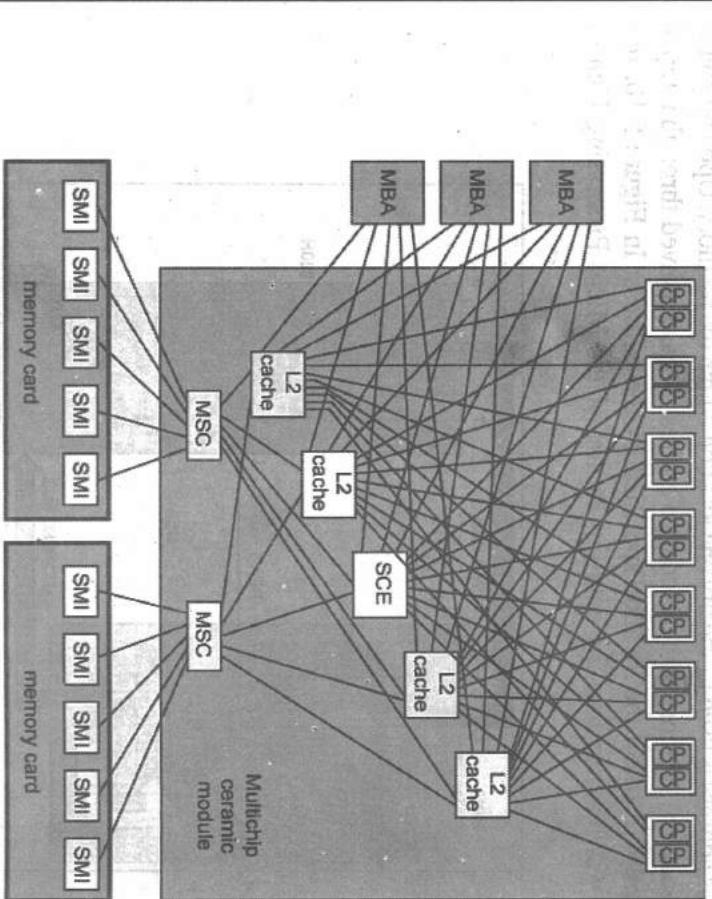


Fig. 5.11 IBM z990 Multiprocessor Structure

Figure 5.11 illustrates IBM z990 multiprocessor structure which uses SMI or Synchronous Memory Interface, SCE or System Control Element, MSC or Main Store Control, MBA or Memory Bus Adapter and CP or Central Processor. The role of system control element is to control the activity of the system. In IBM z990 multiprocessor, memory card is used for storing digital information. The advantage of exclusive caches is that they store more data. In multiprocessor, the L2 cache is connected to the front side bus. Through it, it connects to the chipset's north bridge and RAM. A memory bus adapter uses those data which are to be transmitted to a computer or other device. The function of an adapter is to convert a specific type of memory card into another format or size that is required to read the data.

NOTES

Parallel processing is the computing architecture that allows simultaneous use of multiple processors to execute the various parts of the same program. 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 FORmula TRANslation or 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 (VHSICHDL).

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 (refer Figure 5.12). 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.

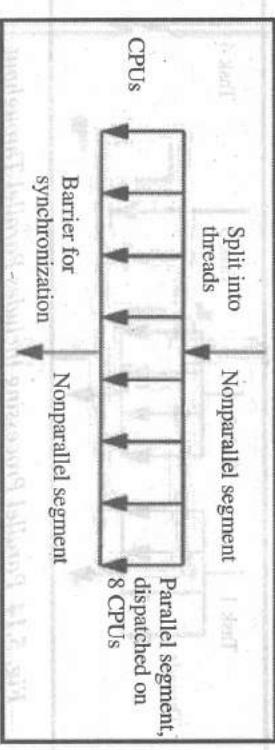


Fig. 5.12 Architecture under Parallel Processing

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 because parallelism depends on the application and the coding in which it is installed.

NOTES

Figure 5.13 illustrates a graphic representation of parallel application's speedup. In parallel computing, the term 'speedup' refers to how much a parallel algorithm is faster than a corresponding sequential algorithm.

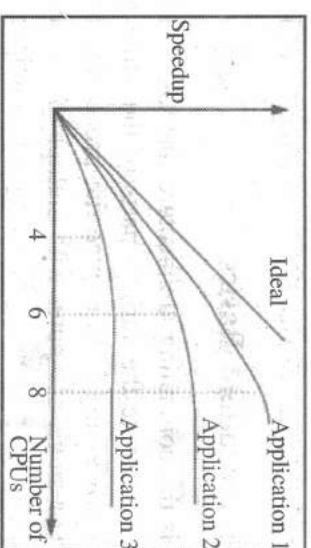


Fig. 5.13 Parallel Application's Speedup

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 that refers to the performance for multiple parallel machines at the same time on a multiprocessor (refer Figure 5.14).

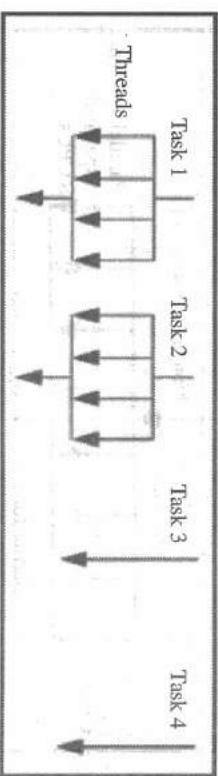


Fig. 5.14 Parallel Processing includes Parallel Throughput

NOTES

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.

QUESTION 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)

Symmetric Multiprocessor or 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. Figure 5.15 illustrates the structure of symmetric multiprocessor in which various CPUs are used to perform I/O operations.

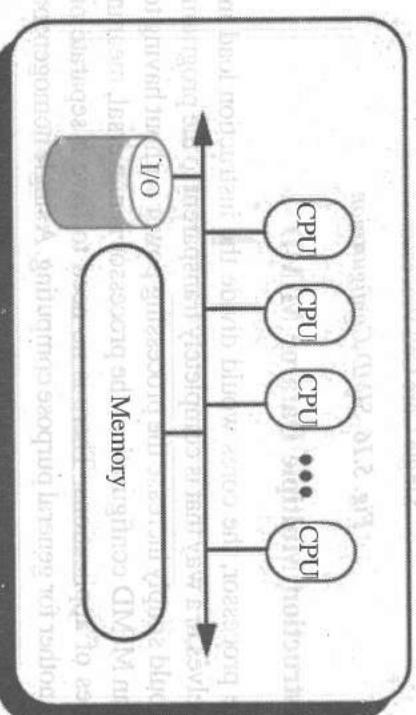


Fig. 5.15 Symmetric Multiprocessor

Non-Uniform Memory Access or 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 Massively Parallel Processing or MPP.

Uniform Memory Access or 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.

NOTES

Single Instruction Multiple Data or 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 5.16). 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.

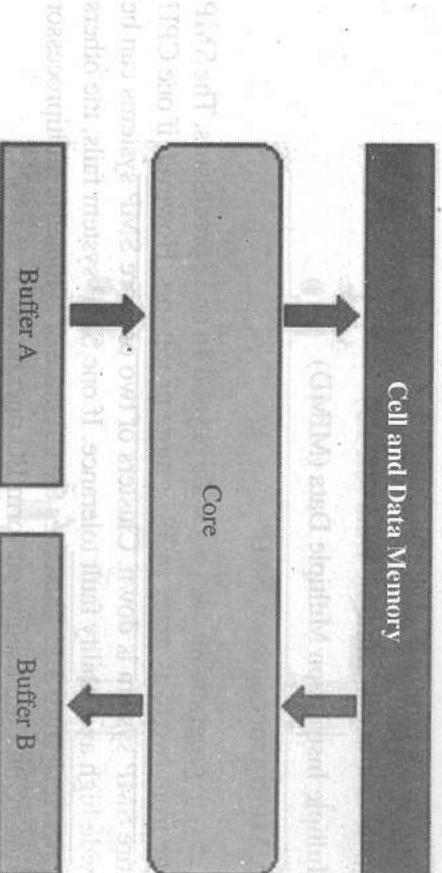


Fig. 5.16 SIMD Configuration

Multiple Instruction Multiple Data or 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.

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 5.17.

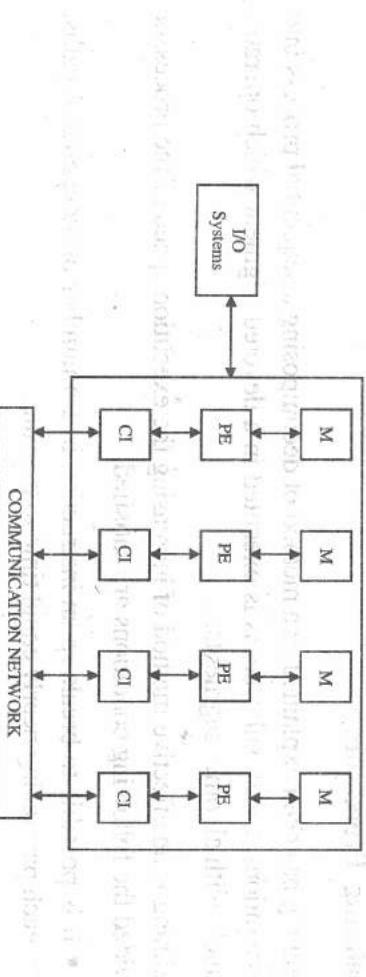


Fig. 5.17 Parallel Computer

In Figure 5.17, PE represents Processing Elements which are used to perform transformations on data items, CI or Communication Interface which is an electronic circuit, designed to a specific standard, enables one machine to communicate with another machine, I/O system represents a communication system between information processing systems and M represents local memory for PE.

Communication Network

Following are the advantages of communication network:

- It is a fixed interconnection network.
- It can be a programmable interconnection network where switches are used for interconnection which can be programmed using bit patterns to change the existing connections.
- A single bus or a set of buses are used in the communication network.
- A memory shared by all the PEs which is used to communicate among the PEs.

5.2.2 Types of Parallel Processor Systems

Parallel computing means doing several tasks simultaneously, thus improving the performance of the computer system. Parallel processing is a form of computing where many instructions are carried out simultaneously on the principle that larger problems can be divided into smaller ones, which are solved concurrently in parallel. The different forms of parallel computing are as follows:

- Bit-level parallelism
 - Instruction-level parallelism
 - Data parallelism
 - Task parallelism
- If a computer system provides the technique for simultaneously processing of different sets of data, i.e., if multiple instructions are executed simultaneously, then the computer system is said to be using parallel processing technique. The parallel processing technique increases the computational speed of the processing system. Thus, parallel processing technique enhances the performances and throughput of a computer. The various techniques by which parallel processing can be achieved are as follows.

NOTES

Pipelining Processors

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.

NOTES

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 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 5.18. R_1, R_2, R_3, R_4 and R_5 are registers that receive new information with every clock pulse. Multiplier and adder are combinational circuits used in Figure 5.18. The suboperations done in every segment of the pipeline are as follows:

- $R_1 \leftarrow A_i$ Input A_i and B_i
- $R_2 \leftarrow B_i$
- $R_3 \leftarrow R_1 * R_2$ Multiply and input C_i
- $R_4 \leftarrow C_i$
- $R_5 \leftarrow R_3 + R_4$ Add C_i to product

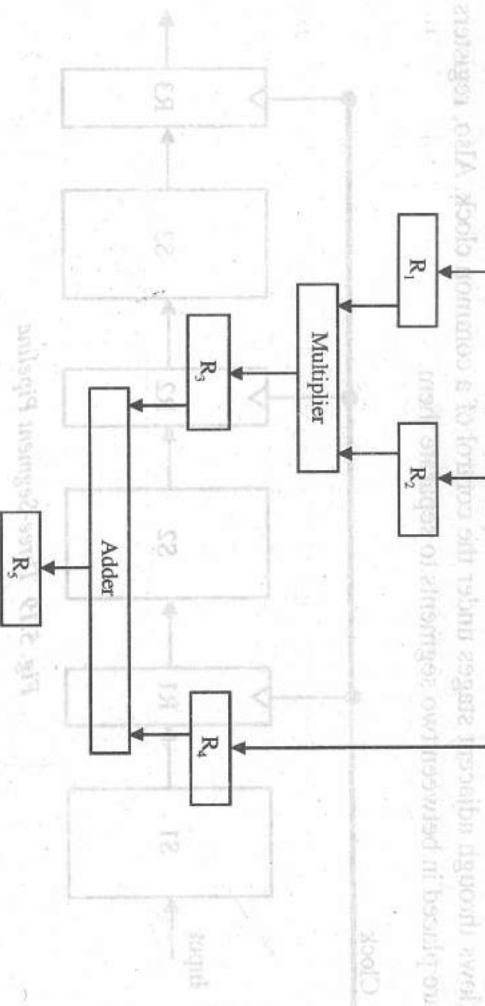


Fig. 5.18 Pipeline Processing of instruction

Table 5.1 Content of Registers in a Pipeline

Clock Pulse Number	Segment 1		Segment 2		Segment 3	
	R_1	R_2	R_3	R_4	R_5	
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_2	$A_1 * B_1 + C_1$	
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$	
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$	
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$	
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$	
8	A_8	B_8	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$	
9	—	—	—	—	$A_7 * B_7 + C_7$	

The five registers are loaded with new data at every clock pulse, as summarized in Table 5.1. With the first clock pulse, the value of A_1 and B_1 transfers into register R_1 and R_2 , respectively. The product of R_1 and R_2 will transfer into R_3 with the second clock pulse. At the same clock pulse, the value of C_1 will transfer into register R_4 and the value of A_2 and B_2 will transfer into register R_1 and R_2 . The third clock pulse operates on all three segments simultaneously. It places A_3 and B_3 into R_1 and R_2 , sends the product of R_1 and R_2 into R_3 , transfers C_2 into R_4 and places the sum of R_3 and R_4 into R_5 . It takes three clock pulses to fill the pipe and retrieve the first output from R_5 . 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 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 5.19. 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. R_i is the register that will hold the intermediate

NOTES

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.

NOTES

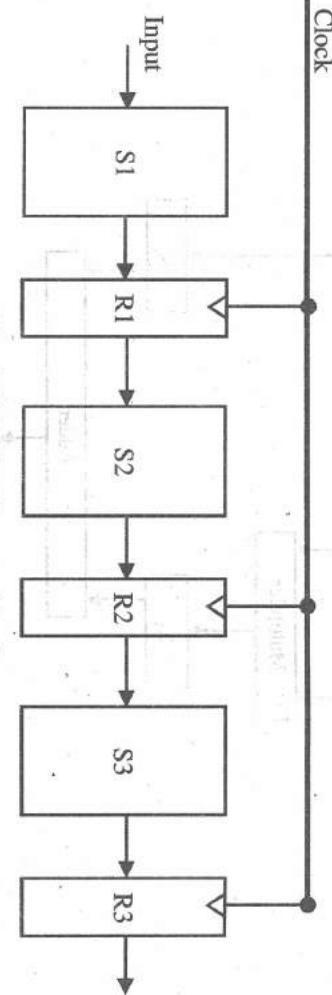


Fig. 5.19 Three-Segment Pipeline

Space-Time Diagram for Pipeline

The behaviour of a pipeline can be illustrated with a space-time diagram (refer Figure 5.20). This is a diagram that shows the segment utilization as a function of time. The time in clock cycles, is given along the horizontal axis and the vertical axis gives the segment number.

Segment	1	2	3	4	5	6	7	8	9	Clock cycles
1	T_1	T_2	T_3	T_4	T_5	T_6				
2		T_1	T_2	T_3	T_4	T_5	T_6			
3			T_1	T_2	T_3	T_4	T_5	T_6		
4				T_1	T_2	T_3	T_4	T_5	T_6	

Fig. 5.20 Space-Time Diagram

The diagram shown in Figure 5.20 represents six tasks T_1 through T_6 executed in four segments. Initially, task T_1 is handled by segment 1. After the first clock, segment 2 is busy with T_1 , while segment 1 is busy with task T_2 . Continuing in this manner, the first task T_1 is completed after the fourth clock cycle. From then on, 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_p is used to execute n tasks.

The first task T_1 needs the same time as kt_p 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 $(n - 1)t_p$.

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_n to complete each task. The total time required for n tasks is nt_n .

The *speedup* of a pipeline processing over an equivalent non-pipeline processing is defined by the ratio as follows:

$$S = nt_n / (k + (n - 1)t_p)$$

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 as follows:

$$S = t_n / t_p$$

Array Processors

An array processor is a processor that is designed for performing calculations on large-sized array of data. There are two types of array processor:

- Attached Array Processor
- SIMD Array Processor

Attached Array Processor

An attached array processor is a peripheral device that is attached to a computer so that the performance of the 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 ALU, which may contain one or more pipelined floating-point adders and multipliers.

Figure 5.21 shows the interconnection of an attached array processor with a computer. The array processor is interfaced with the computer through 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 computer the functionality of the vector manipulations at a fraction of the cost of supercomputer.

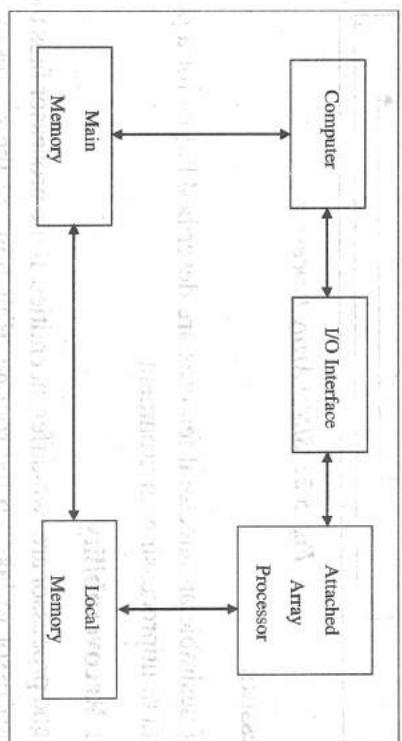


Fig. 5.21 Attached Array Processor

NOTES

An SIMD array processor has 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 5.22. The processing elements may include ALU, floating-point arithmetic unit and registers. The main memory of the CPU is used for the storage of program. The operation of the processing element is controlled by the master control unit, whose main function is to decode the instruction and how the instruction is to be executed. Data operands are transferred to local memories. Each processing element 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 th data element in a local memory M_i . It then gives the add instruction $c_i = a_i + b_i$ to the processing element causing the addition to take place simultaneously. The result c_i is stored in the local memory. Thus, the whole process is performed in one cycle.

ILLINOIS Integrator And Calculator or ILLIAC IV computer is the best-known SIMD array processor developed at the University of Illinois and manufactured by the Burroughs Corporation.

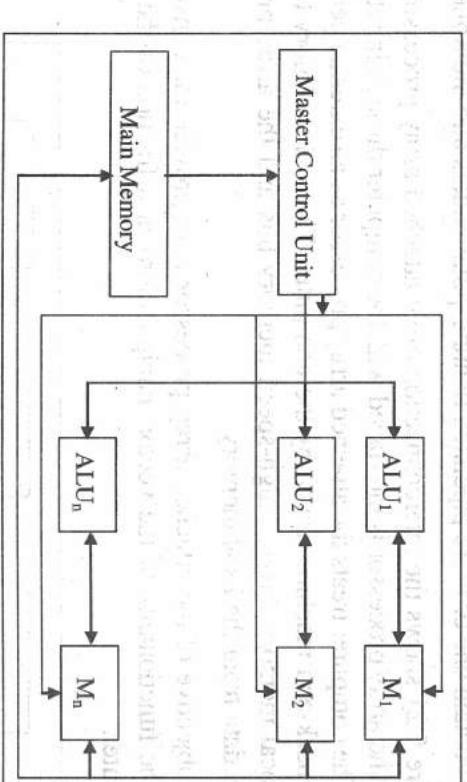


Fig. 5.22 SIMD Array Processor

Multiprocessors

A number of desirable architectural features are described below for a processor to be effective in multiprocessing environment.

- **Processor Recoverability**

The process and processor are two different entities. If the processor fails there should be another processor to take up the routine. Reliability of the processor should be present.

• Efficient Context Switching

A general-purpose register is a large register file that can be used for multi-programmed processors. For effective utilization it is necessary for the processor to support more than one addressing domain and hence to provide a domain change or context switching operation.

Notes

• Large Virtual and Physical Address Space

A processor intended to be used in the construction of a general-purpose medium to large-scale multiprocessor must support a large physical address space. In addition, a large virtual space is also needed.

• Effective Synchronization Primitives

The processor design must provide some implementation of invisible actions which serve as the basis for synchronization primitives.

• Interprocessor Communication Mechanism

The set of processor used in the multiprocessor must have an efficient means of inter-processor mechanism.

• Instruction Set

The instruction set of the processor should have adequate facilities for implementing high level languages that permit effective concurrency at the procedural level and for efficiently manipulating the data structures.

Processor Symmetry

In a multiprocessing system, all CPUs may be equal, or some may be reserved for special purposes. A combination of hardware and operating-system software design considerations determine the symmetry (or lack thereof) in a given system. For example, hardware or software considerations may require that only one CPU respond to all hardware interrupts, whereas all other work in the system may be distributed equally among CPUs; or execution of kernel-mode code may be restricted to only one processor (either a specific processor, or only one processor at a time) whereas the user-mode code may be executed in any combination of processors. Multiprocessing systems are often easier to design if such restrictions are imposed, but they tend to be less efficient than systems in which all CPUs are utilized equally.

Systems that treat all CPUs equally are called symmetric multiprocessing systems. In systems where all CPUs are not equal, system resources may be divided in a number of ways, including ASymmetric MultiProcessing (ASMP), NUMA multiprocessing and clustered multiprocessing.

Shared Memory

Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space (refer Figure 5.23). Multiple processors can operate independently but share the same memory resources. Changes in a memory location effected by one processor are visible to all other processors. Shared memory machines can be divided into two main classes based upon memory access times: *UMA* and *NUMA*.

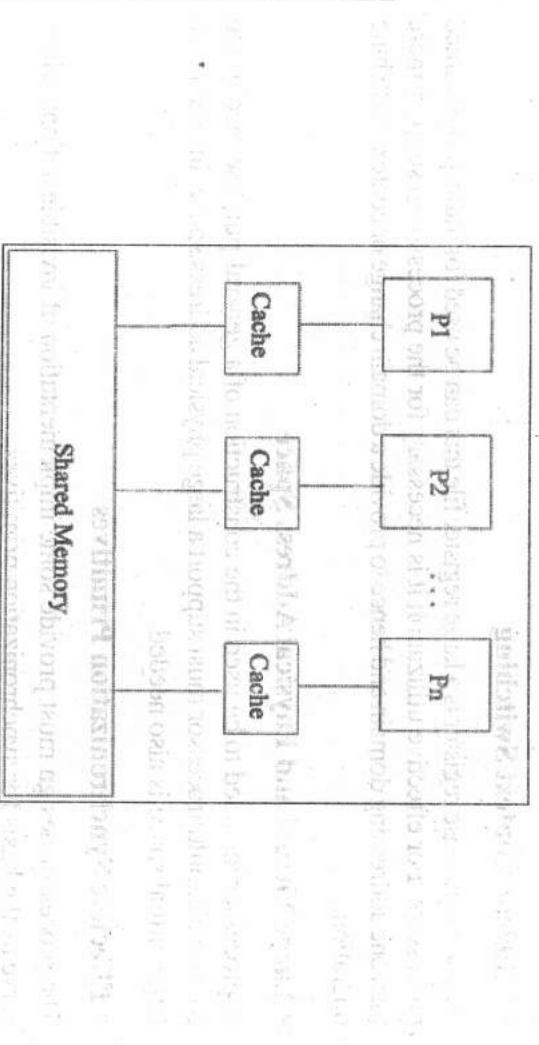
NOTES

Fig. 5.23 Shared Memory Machine

Uniform Memory Access or UMA: Most commonly represented today by symmetric multiprocessor machines, they include identical processors each having equal access and access times to memory. Sometimes, they are called CC-UMA - Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.

Non-Uniform Memory Access or NUMA: They are made by physically linking two or more SMPs. One SMP can directly access the memory of another SMP, not all processors have equal access time to all memories. Memory access across link is slower. If cache coherency is maintained, then may also be called CC-NUMA - Cache Coherent NUMA.

Advantages of Shared Memory

Following are the advantages of shared memory:

- Global address space provides a user friendly programming perspective to memory.
- Data sharing between tasks is both fast and uniform due to the proximity of memory to the CPUs.

Disadvantages of Shared Memory

Following are the disadvantages of shared memory:

- Primary disadvantage is the lack of scalability between the memory and the CPUs. Adding more CPUs can geometrically increase traffic on the shared memory-CPU path and for cache coherent systems, geometrically increase traffic associated with cache/memory management.
- Programmer responsibility for synchronization constructs that insure "correct" access of global memory.
- It becomes increasingly difficult and expensive to design and produce shared memory machines with an ever increasing numbers of processors.

Distributed Memory

Like shared memory systems, distributed memory systems vary widely but share a common characteristic (refer Figure 5.24). Distributed memory systems require a communication network to connect to the inter-processor memory. Processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors. Since each processor has its own local memory, it operates independently. The changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of cache coherency does not apply. When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when the data is communicated. Synchronization between tasks is likewise the programmer's responsibility. The network 'fabric' used for data transfer varies widely, though it can be as simple as Ethernet (refer Figure 5.24).

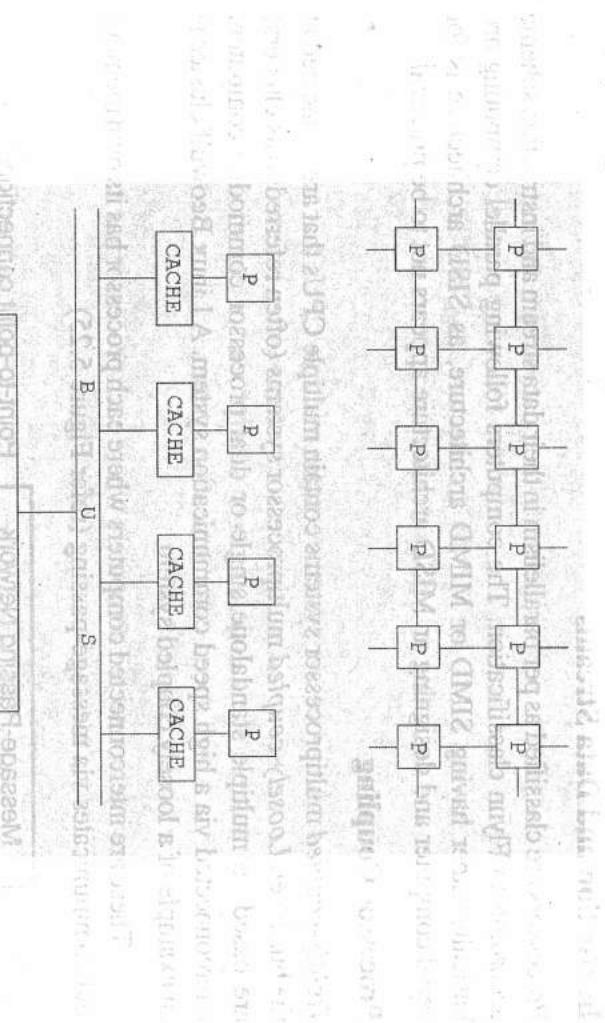


Fig. 5.24 Distributed Memory Machine

Advantages of Distributed Memory

Following are the advantages of distributed memory:

- Memory is scalable with a number of processors. Increase the number of processors and the size of memory increases proportionately.

NOTES

- Each processor can rapidly access its own memory without interference and without the overheads incurred with trying to maintain a cache coherency.
- It is very cost-effective as it can use commodity, off-the-shelf processors and networking.

NOTES

Disadvantages of Distributed Memory

Following are the disadvantages of distributed memory:

- The programmer is responsible for many of the details associated with data communication between the processors.
- It may be difficult to map the existing data structures, based on global memory, to this memory organization.
- It supports non-uniform memory access operation times.

Instruction and Data Streams

Processors are classified as per parallelism in their data stream and instruction scheme according to Flynn classification. The computers following parallel computing are basically either having SIMD or MIMD architecture, as SISD architecture is for serial computer and designing an MISD architecture appears not to be practical.

Processor Coupling

Tightly-coupled multiprocessor systems contain multiple CPUs that are connected at the bus level. *Loosely-coupled* multiprocessor systems (often referred to as clusters) are based on multiple standalone single or dual processor commodity computers interconnected via a high speed communication system. A Linux Beowulf cluster is an example of a loosely coupled system.

These are interconnected computers where each processor has its own memory and communicates via message-passing (refer Figure 5.25)

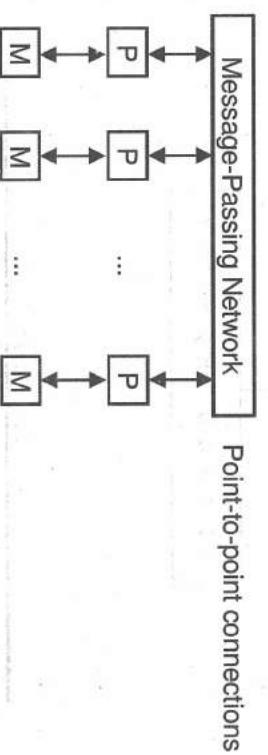


Fig. 5.25 Interconnected Computers

Example of systems:

- **Tree Structure:** Teradata, DADO.
- **Mesh-Connected:** Rediflow, Series 2010, J-Machine.
- **Hypercube:** Cosmic Cube, iPSC, NCUBE, FPS T Series, Mark III.

Limitations of loosely-coupled systems are as follows:

- Communication overhead.
- Hard to program.

5.2.3 Flynn's Classification of Computers

Parallel Organization

The classification based on the multiplicity of instruction streams and data streams in a computer system is known as Flynn's classification.

Parallel processing can be classified in a variety of ways. It can be classified on the basis of internal organization of processors, the interconnection structure between processors or the flow of information through the system.

M.J. Flynn studied how do instructions and data flow in a system and classified parallel computers into the following four categories:

- 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 sequence 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 or SISD

A computer with a single processor is called 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 may be achieved 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, processes data and the results are stored back in the main memory. A SISD computer organization is illustrated in Figure 5.26.

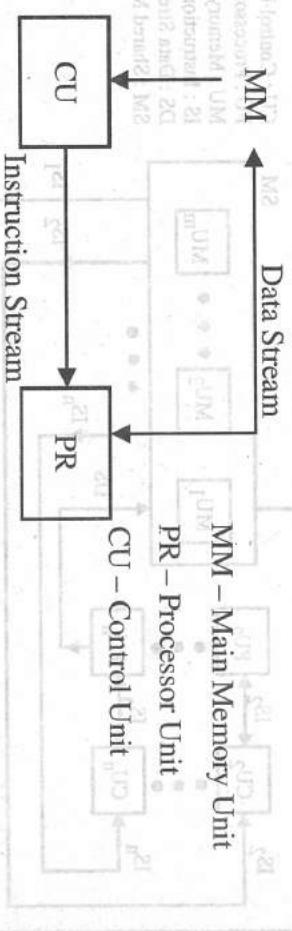


Fig. 5.26 SISD Organization

Single Instruction Stream, Multiple Data Stream or SIMD

It represents an organization of 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 the data. SIMD computers are

NOTES

used to solve many problems in science, which require identical operations to be applied to different data set synchronously. Such computers are known as array processors. An SIMD computer organization is shown in Figure 5.27.

NOTES

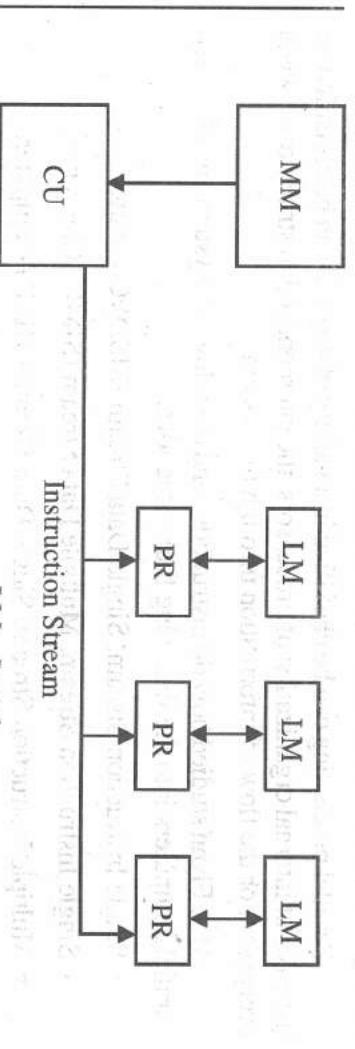


Fig. 5.27 SIMD Organization

Multiple Instruction Stream, Single Data Stream or MISD

It refers to a computer in which several instructions manipulate the same data stream concurrently. In this type of processor, 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. A MISD computer organization is shown in Figure 5.28.

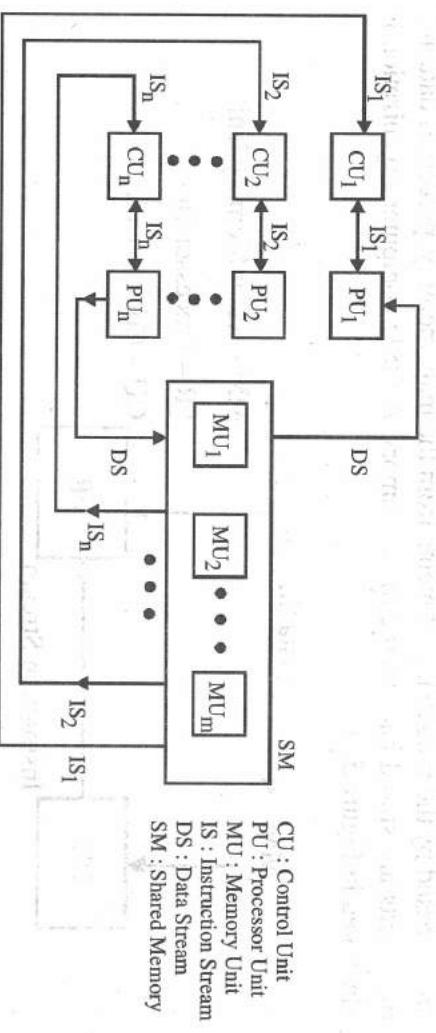


Fig. 5.28 MISD Organization

Multiple Instruction Stream, Multiple Data Stream or MIMD

MIMD computers are the general-purpose parallel computers. Its organization refers to a computer system capable of processing several programs at the same time. MIMD

systems include all multiprocessing systems. An MIMD computer organization is shown in Figure 5.29.

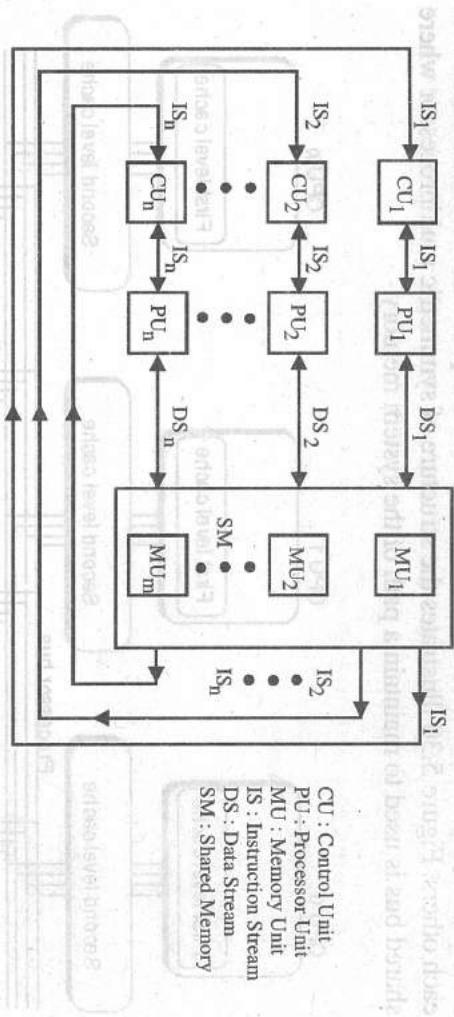


Fig. 5.29 MIMD Organization

5.3 SYMMETRIC MULTIPROCESSORS

Multiprocessor architectures are used frequently in computing areas. As the need for increasing the speed, multiprocessor computers are optimized to produce more computing power. A part of the multiprocessor systems are Symmetric Multiprocessor (SMP) systems. SMP systems are tightly-coupled programs using operating systems such as Windows 2000, Linux, Solaris, QNX. There are two main paths used for the increased processor capacity in a parallel architecture. One way is to let the operating system schedule separate programs to separate processors in any arbitrary way. The other way is to design multithreaded applications for parallel computation. Such programs support multiple processors and can through the use of parallel algorithms use the multiple processors to produce results in parallel since dividing a problem to n independent partial problems has proven to be architecture dependent if a speedup is to be achieved. Speedup is a measurement of how much time a program takes to complete a task after 'change'. The term 'change' refers to adding $n-1$ processors to the same architecture having one processor. As an example, if a task takes T_s to execute on one processor, and the portion of the task that can be parallelized is P , then the time for parallel execution on N processors will be calculated as per parallel execution time:

$$T_p = \left(1 - P\right) + \frac{P}{N} + V(N) T_s$$

In the above expression, V is the overhead created for the parallel execution. The speedup can be calculated as the quotient between the sequential execution time and the parallel execution time. Following formula is used to calculate speedup calculation:

The overhead $V(N)$ in above expression is partially created through extra code in defined algorithm. It also contains an architecture dependent portion. This portion

NOTES

NOTES

includes communications overhead. In a distributed system, this mechanism is applied for network delays. In an SMP system, the process of communication delays is often generated by the need for coherence traffic, i.e., the processors communicate with each others. Figure 5.30 illustrates the structure of symmetric multiprocessor where shared bus is used to maintain a path of the system memory.

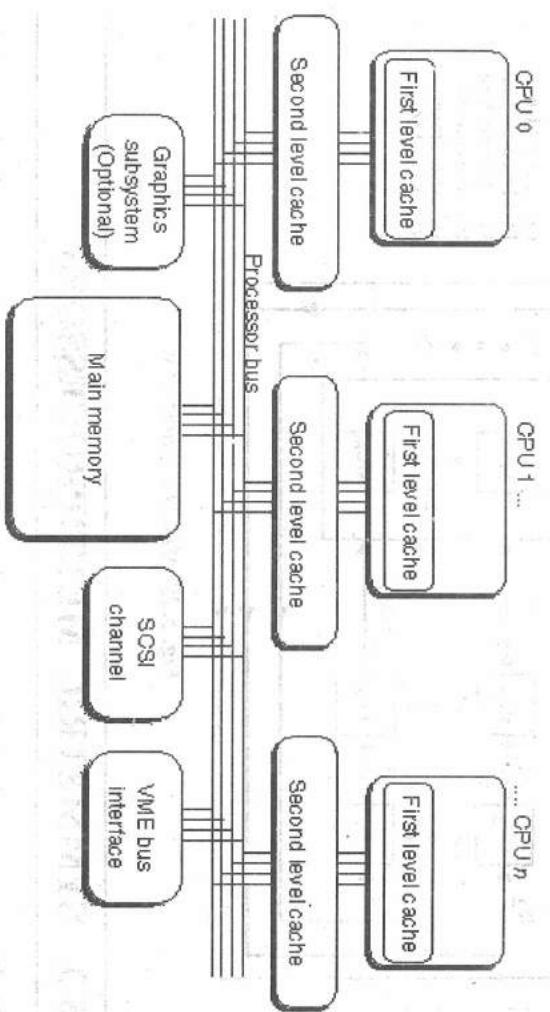


Fig. 5.30 Processor Bus used in Symmetric Multiprocessor

Figure 5.30 illustrates the structure of symmetric multiprocessor. In this structure, CPU 0, CPU 1, ..., CPU n , are used to maintain data that is reusable between transactions at a process or cluster level. In symmetric multiprocessor, the processor bus is used as communication path between the CPU and the main bus. The processor bus is the set of wires used to carry information to and from the processor (refer Figure 5.30). The memory bus is considered as the main data path at the system level on the PC. All transfers of data to and from the processor go through this bus. It is also used for communication between the CPU and the processor support chipset. The processor support chipset includes chips, such as an external memory cache and the bus controller chip found on some microcomputers. Processor buses can have a maximum data transfer rate of the motherboard clock. These CPUs are connected to the same memory module through a bus or crossbar switch. Therefore, SMP involves a multiprocessor computer architecture where two or more identical processors can connect to a single shared main memory. Most common multiprocessor systems today use SMP architecture. SMP systems allow any processor to work on any task with proper operating system support. Small Computer System Interface or SCSI is a set of standards for physically connecting and transferring data between computers and peripheral devices. The SCSI standards define commands, protocols, channels, and electrical and optical interfaces. VMEbus is a computer bus standard developed for the Motorola 68000 lines of CPUs. In computer architecture, a bus is a subsystem that transfers data between components inside a computer or between computers. The prime feature of SMP systems is to transmit data frequently between processors.

for balancing the workload. When designing operating systems for symmetric multiprocessors, a set of performance issues must be considered. Cache coherence and lock granularity are very important factors in symmetric multiprocessors. SMP systems lack not only from coherence and lock-granularity but also from more high cache miss latency. The latency depends on more complicated coherence and bus protocols to the main memory. Optimization techniques are used to hide these latencies. For example, non-blocking caches and split-transaction buses are used in all new systems to hide latencies. One important characteristic of SMP system is the use of primary CPU. During system operation, the primary CPU performs various responsibilities for system timekeeping, writing messages to the console terminal and accessing any other Input/Output or I/O devices that are not accessible to all members. An SMP configuration may include some devices that are not accessible from all SMP members. The console terminal is accessible only from the primary processor.

Booting the Symmetric Multiprocessor System

The process of booting the SMP system is started with CPU and full access to the console subsystem. Therefore, the particular terminal is known as the 'BOOT CPU'. The BOOT CPU controls the bootstrap sequence and boots the other available CPUs. The booted primary and all currently booted secondary processors are called members of the active set. These processors actively participate in system operations and respond to inter-processor interrupts to control the system events. In an SMP system, each processor services its own software interrupt requests which are sent in the following way:

- When a current kernel thread is preempted by a higher priority computable resident thread, the Interrupt Priority Level or IPL reschedules interrupt service routine to take the current thread out of execution and switches to the higher priority kernel thread.
- When a device driver completes an I/O request, an IPL uses I/O post processing. At this stage, completed requests are queued to a CPU-specific post processing queue and are serviced on that CPU whether others are queued to a system queue and serviced on the primary CPU.
- When the current kernel thread has used its quantum of CPU time, the software timer interrupt service routine running on that CPU performs quantum-end processing.
- Each processor services its own set of fork queues. A fork process is executed on the same CPU from which it has been requested. Many fork processes are requested from device interrupt service routines which currently execute only on the primary CPU more fork processes execute on the primary processors.

5.3.1 Organization of Symmetric Multiprocessing

Symmetric multiprocessing organization involves a multiprocessor computer hardware architecture where two or more identical processors are connected to a single shared main memory and controlled by a single operating system. Most common multiprocessor systems today use symmetric multiprocessing architecture. In the case

NOTES

NOTES

of multi-core processors, the symmetric multiprocessing architecture applies to the cores treating them as separate processors. Processors may be interconnected using buses, crossbar switches or on-chip mesh networks. The bottleneck in the scalability of SMP uses buses or crossbar switches is the bandwidth and power consumption of interconnecting among the various processors, the memory and the disk arrays. A computer system that uses symmetric multiprocessing is called a symmetric multiprocessor or symmetric multiprocessor system. SMP systems allow any processor to work on any task no matter where the data for that task are located in memory, provided that each task in the system is not in execution on two or more processors at the same time with proper operating system support. SMP systems can easily transmit data between processors to balance the workload efficiently. L2 cache is part of a computer's processor that helps improve computer performance. In chip level multiprocessing, multiple CPUs are connected via a shared bus to a shared memory (level 2 cache). Each processor also has its own fast memory (a level 1 cache). The tightly-coupled feature of the CMP allows less physical distances between processors and memory and therefore leads to minimal memory access latency and higher performance. This type of architecture works well in multi-threaded applications where threads can be distributed across the processors to operate in parallel. This is known as thread-level parallelism. An I/O adapter is used with the I/O subsystem to communicate with a scanner port for a processor which is assembled at another location (refer Figure 5.31).

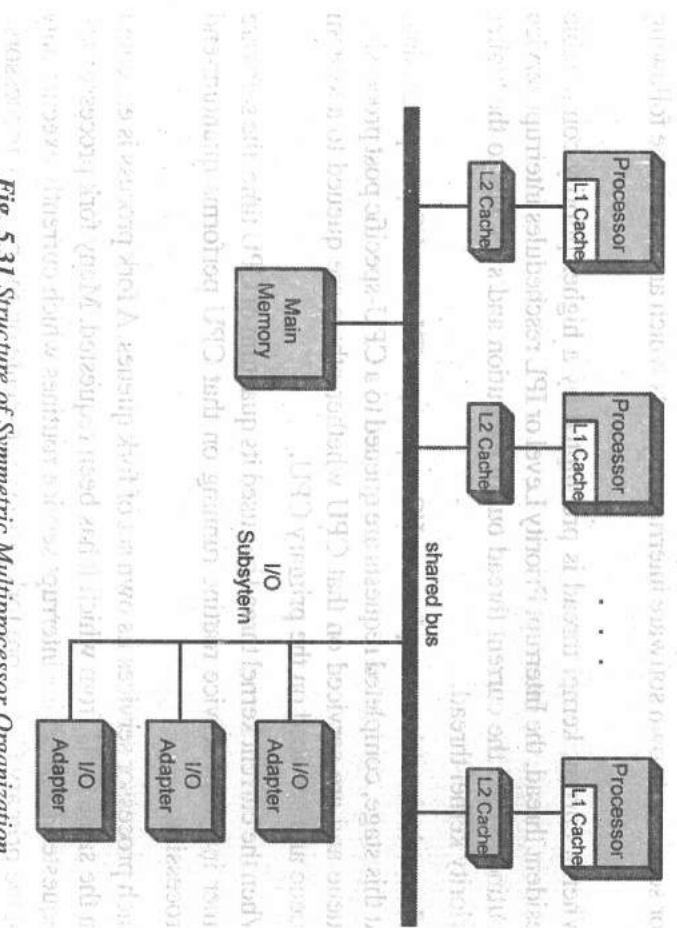


Fig. 5.31 Structure of Symmetric Multiprocessor Organization

Figure 5.31 illustrates the structure of symmetric multiprocessor organization. The advantage of a memory-mapped I/O subsystem is that the CPU can use any instruction that accesses memory to transfer data between the CPU and a memory-mapped I/O device. The MOV instruction is the one most commonly used to send and receive data from a memory-mapped I/O device. For example, if you have an I/O port that is

read/write, you can use the ADD instruction to read the port, add data to the value read, and then write data back to the port. A port is a buffered I/O device or vice-versa. Bus networks are used to connect multiple clients but problems occur when two clients want to transmit at the same time on the same bus. Thus, systems which use bus network architectures normally have some scheme of collision handling or collision avoidance for communication on the bus. This scheme supports Carrier Sense Multiple Access or CSMA and a bus master which control accessing technique to the shared bus resource. CSMA is a Media Access Control or MAC protocol in which a node checks the absence of other traffic before transmitting data packets on a shared transmission medium, such as an electrical bus.

NOTES

5.4 CLUSTERS

A computer cluster is a group of linked computers which work together closely and hence can be viewed as a single computer. The components of a cluster are commonly connected to each other through fast local area networks. Clusters are used to improve performance and availability over that of a single computer while being much more cost-effective than single computers in terms of speed or availability. A computer cluster consists of a set of loosely connected computers that work together as a single system. Computer clusters have a wide range of applicability and deployment ranging from small business clusters with nodes to the fastest supercomputers, such as the K computer.

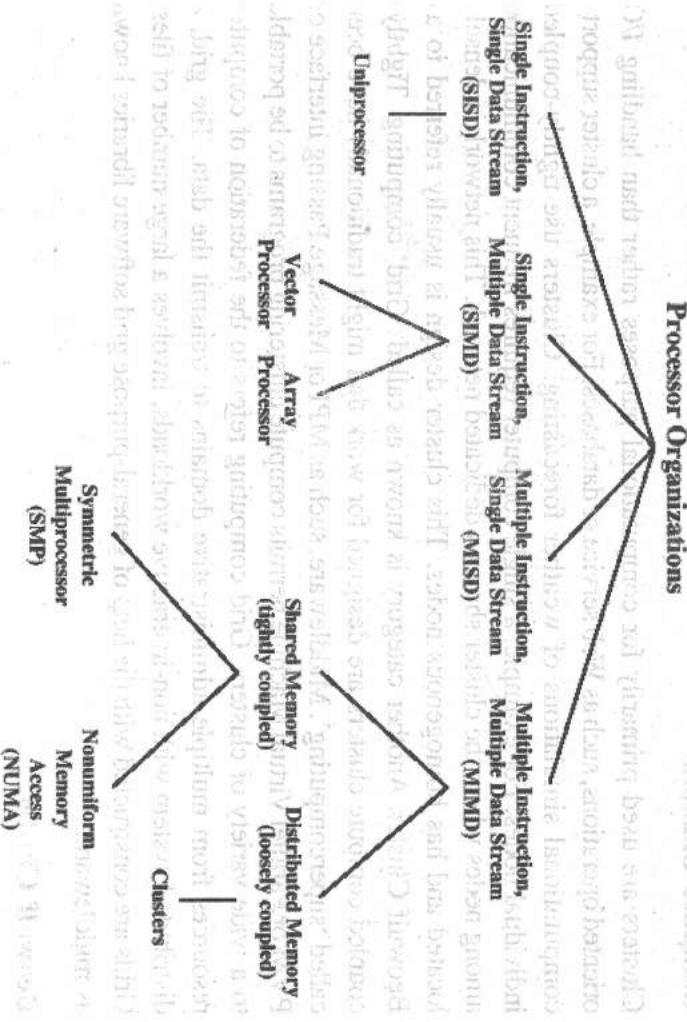


Fig. 5.32 Cluster in Hierarchy of Parallel Organization

- Check Your Progress**
8. What is VMEbus?
 9. Write the function of BOOT CPU in symmetric multiprocessor system.
 10. Why is an I/O adapter used in symmetric multiprocessor organization?

Figure 5.32 illustrates the hierarchy of parallel organization which includes clusters. A cluster is a type of parallel or distributed processing system which consists of a collection of interconnected computers cooperatively working together as a single, integrated computing resource. Following are the types of clusters which are used in parallel system implementation:

- **High-Availability Clusters:** High-availability clusters, also known as failover cluster, are implemented primarily for the purpose of improving the availability of services that the cluster provides. Mechanism of these clusters is used to operate with the help of redundant nodes which are then used to provide service when system components fail. The most common size for high-availability cluster is two nodes which is the minimum requirement to provide redundancy. High-availability cluster implementations use redundancy of cluster components which eliminate single points of failure. The Linux-HA project is known as commonly used free software high-availability package for the Linux operating system.

• **Load-Balancing Clusters:** Load-balancing cluster is used when multiple computers are linked together to share computational workload or function as a single virtual computer. Logically, from the user side load-balancing clusters are multiple machines but work as a single virtual machine. Requests sent from the user are managed and distributed by the standalone computers to form a cluster. This technique results in balanced computational work among different machines improving the performance of the cluster systems.

Compute Clusters

Clusters are used primarily for computational purposes rather than handling I/O oriented operations, such as Web service or databases. For example, a cluster supports computational simulations of weather forecasting. Clusters use tightly-coupled individual nodes. For example, a single computer requires frequent communication among nodes where the cluster shares a dedicated network. This network is densely located and has homogenous nodes. This cluster design is usually referred to as Beowulf Cluster. Another category is known as called 'Grid' computing. Tightly-coupled compute clusters are designed for work that might traditionally have been called 'supercomputing'. Middleware, such as MPI or Message Passing Interface or PVM or Parallel Virtual Machine permits compute clustering programs to be portable to a wide variety of clusters. Grid computing refers to the federation of computer resources from multiple administrative domains to transmit the data. The grid, a distributed system with non-interactive workloads, involves a large number of files. Grids are constructed with the help of general-purpose grid software libraries known as middleware.

Beowulf Cluster

Beowulf cluster, commodity-grade computers, is networked into a small Local Area Network or LAN with libraries and programs installed. This cluster allows processing

to be shared among them. The result is a high performance parallel computing cluster from inexpensive personal computer hardware. The name ‘Beowulf’ is referred to a specific computer built by Thomas Sterling and Donald Becker at National Aeronautics and Space Administration or NASA.

Figure 5.33 illustrates schematic view of a Beowulf cluster consisting of 8 identical high end Personal Computers or PCs connected by a high speed network. The PCs are implemented with Advanced Micro Devices or AMD Athlon processor. Athlon is a series of x86-compatible microprocessors designed and manufactured by AMD. Like AMD K5 and K6, the Athlon buffers internal microinstructions at runtime resulting from parallel x86 instruction decoding. The CPU used in AMD is an out-of-order design. The Athlon utilizes Double Data Rate (DDR) technology which means that at 100 MHz the Athlon front side bus transfers at a rate similar to a 200 MHz single data rate bus.

NOTES

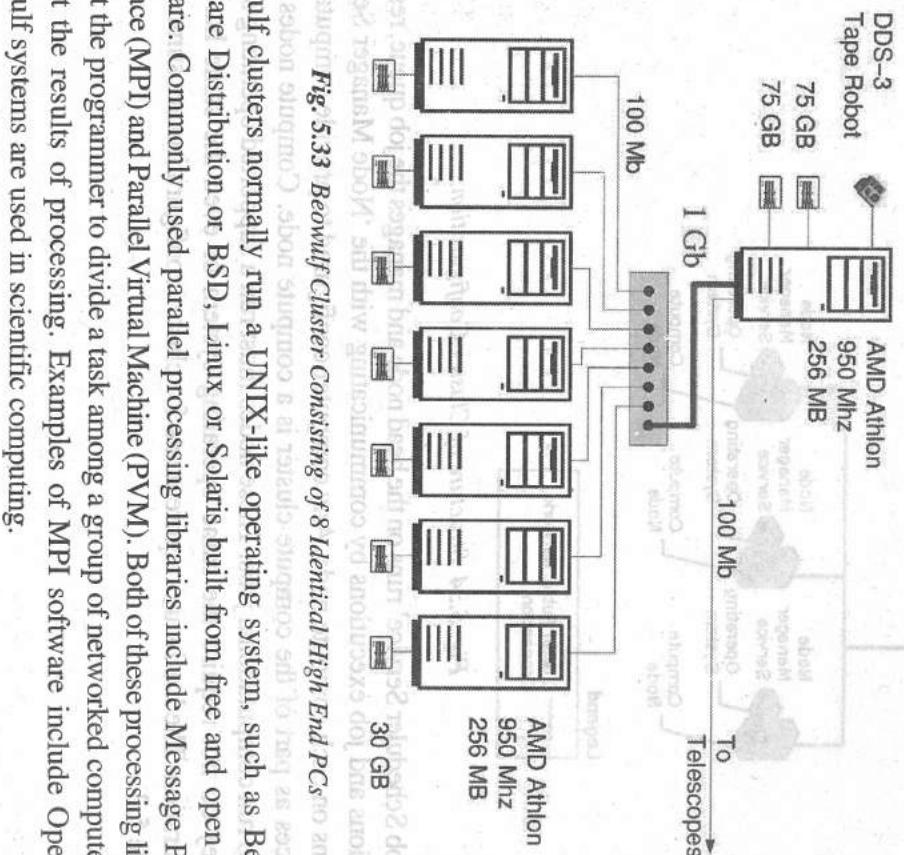


Fig. 5.33 Beowulf Cluster Consisting of 8 Identical High End PCs

Beowulf clusters normally run a UNIX-like operating system, such as Berkeley Software Distribution or BSD, Linux or Solaris, built from free and open source software. Commonly used parallel processing libraries include Message Passing Interface (MPI) and Parallel Virtual Machine (PVM). Both of these processing libraries permit the programmer to divide a task among a group of networked computers and collect the results of processing. Examples of MPI software include OpenMPI. Beowulf systems are used in scientific computing.

5.4.1 Cluster Configurations

Cluster configurations include a shared disk subsystem connected to all servers in the cluster. The shared disk subsystem can be connected via high speed fibre channel cards, cables, switches and use shared Small Computer System Interface or SCSI for configuration. SCSI, a set of standards, is used for connecting and transferring data between computers and peripheral devices. If a server fails, then another designated server in the cluster automatically mounts the shared disk volumes. This approach

gives network users continuous access to the volumes on the shared disk subsystem. Resources include data (volumes), applications, server licenses and services. Although fiber channel is the recommended configuration, you can configure your cluster to use shared or SCSI. Figure 5.34 illustrates the structure of cluster configuration which uses a setup among the elements that make up a compute cluster.

NOTES

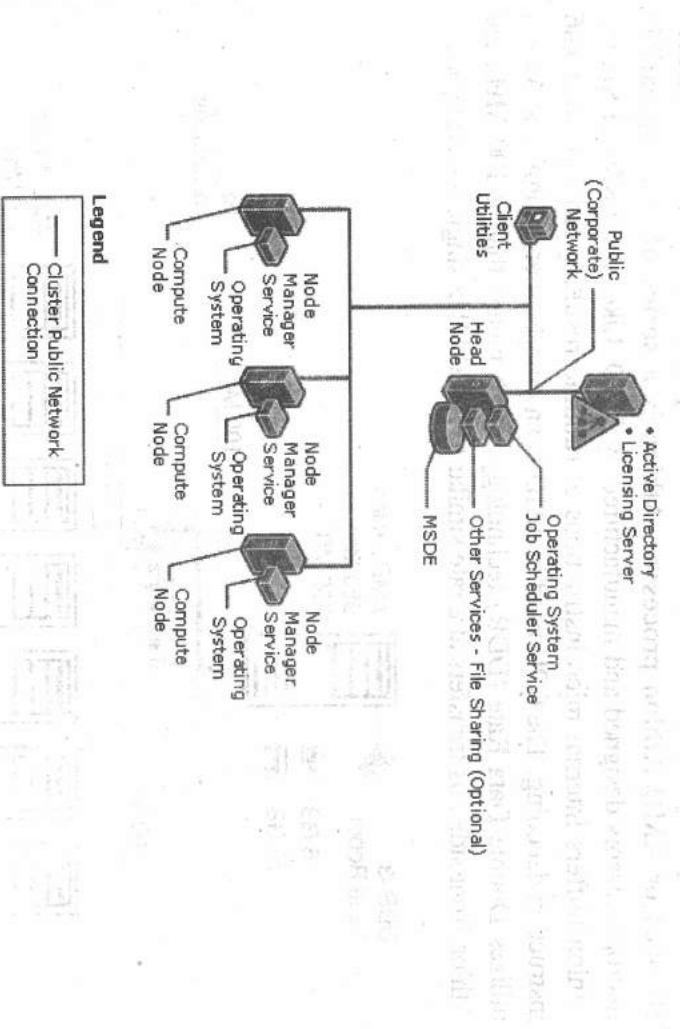


Fig. 5.34 Structure of Cluster Configuration

The 'Job Scheduler Service' runs on the head node and manages the job queue, resource allocations and job executions by communicating with the 'Node Manager Service' that runs on each compute node. Any computer configured to provide computational resources as part of the compute cluster is a compute node. Compute nodes allow users to run computational jobs. These nodes must run a supported operating system but they do not require the same operating system or even the same hardware configuration. The hardware requirements for cluster configuration are summarized in Table 5.2.

Table 5.2 Hardware Requirements for Cluster Configuration

Parallel Organization

Required Hardware	Requirements	NOTES
CPU	X64-bit computer with Intel Pentium or Xeon family processors with Extended Memory 64 Technology (EM64T) architecture along with AMD Opteron or Athlon family processors are required in cluster configuration.	
RAM	512 (MB) minimum RAM space is required.	
Multiprocessor support	Windows Compute Cluster Server 2003 and Standard x64 Edition up to eight processors per server are required.	
Disk space for setup	4 GB disk space is required for configuration setup.	
Disk volumes	For head node, two volumes are required if Remote Installation Services or RIS is used, because RIS data cannot reside on the system volume. For compute nodes, a single system volume for Redundant Array of Independent Disks (RAID) is required in cluster configuration.	
Network adapter	Each node requires at least one network adapter. Additional network adapters are used to set up a private network for the cluster or to set up a high speed network for Microsoft Message Passing Interface or MS MPI.	
Preboot eXecution Environment (PXE) support	If the Automated Addition method is used of adding compute nodes supporting PXE in the boot sequence is added. This is configured in the BIOS.	
Basic Input/Output System (BIOS) and network adapter		

In cluster configuration, Automated Addition method installs an operating system and computes Cluster Pack software on the node to add the other node to the cluster.

5.4.2 Cluster Computer Architecture

The components of a cluster are connected to each other through fast local area networks and each node runs its own instance on an operating system. Computer clusters are emerged as a result of convergence of a number of computing trends which includes the availability of low cost microprocessors, high speed networks and software for high performance distributed computing. The cluster service refers to the collection of components on each node that perform cluster-specific activity and resource refers to the hardware and software components within the cluster that are managed by the cluster service. The prime resource of cluster computing refers to the Dynamically Linked Libraries (DLLs). Cluster resources include physical hardware devices, such as disk drives and network cards, and logical items, such as Internet Protocol (IP) addresses, applications and application databases. Each node in the cluster will have its own local resources. A resource group is a collection of resources managed by the cluster service as a single and logical unit. Application resources and cluster entities can be easily managed by grouping logically related resources into a resource group. When a cluster service operation is performed on a resource group, the operation affects all individual resources contained within the group. Therefore, a resource group is created to contain all the elements needed by a specific application server and client for successful use of the application. Figure 5.35 illustrates the cluster computer architecture. Sequential and parallel applications use cluster middleware.

Wellness

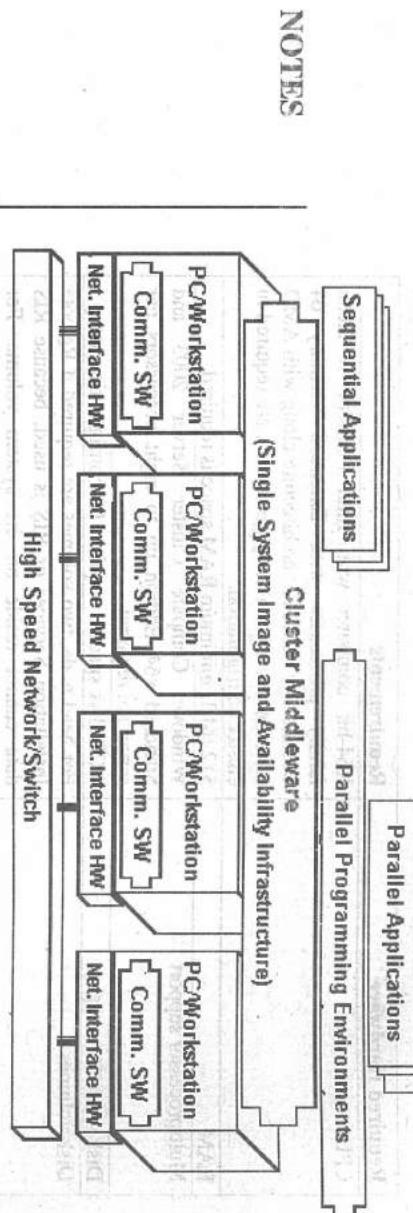


Fig. 5.35 Cluster Computer Architecture

Server Clusters

Server clusters are based on a 'shared-nothing' model of cluster architecture. This model refers to the process which states how servers in a cluster manage and use local and common cluster devices and resources. In the shared-nothing cluster, each server manages its local devices. Devices common to the cluster, such as a common disk array and connection media are selectively owned and managed by a single server at any given time.

5.5 SUMMARY

- One of the most often used parts of a microprocessor is the accumulator. In microprocessors, one of the operands holds a special register called 'accumulator'. The arithmetic and logical operations performed by using this accumulator alter the value that it contains.
- General-purpose registers are temporary storage locations. They store addresses or data and are capable of manipulating data by shift or rotate operations. General-purpose registers are similar to the accumulator.
- 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.
- In Non-Unified Memory Access or 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 a Massively Parallel Processing or MPP.
- Parallel processing is a form of computing where many instructions are carried out simultaneously on the principle that larger problems can be divided into smaller ones, which are solved concurrently in parallel.
- Pipelining is a method of decomposing a sequential process into suboperations. Every subprocess is executed in a devoted segment, which operates parallelly with all other segments.

Check Your Progress

11. Why are high-availability clusters implemented?
12. When load-balancing cluster is used?
13. What does grid computing refers?
14. What is Athlon?

- An attached array processor is a peripheral device that is attached to a computer so that the performance of the 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 slightly for solving complex scientific problems.
- An SIMD array processor has 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.

NOTES

- Loosely coupled multiprocessor systems, often referred to as clusters, based on multiple standalone single or dual processor commodity computers are interconnected via a high speed communication system.
- MIMD computers are the general-purpose parallel computers. Its organization refers to a computer system capable of processing several programs at the same time. MIMD systems include all multiprocessing systems.
- There are two main paths used for the increased processor capacity in a parallel architecture. One way is to let the operating system schedule separate programs to separate processors in any arbitrary way. The other way is to design multithreaded applications for parallel computation.
- A cluster is a type of parallel or distributed processing system which consists of a collection of interconnected computers cooperatively working together as a single, integrated computing resource.
- High-availability clusters, also known as failover cluster, are implemented primarily for the purpose of improving the availability of services that the cluster provides. They operate with the help of redundant nodes which are then used to provide service when system components fail.

5.6 KEY TERMS

- **Microprocessor:** A program controlled semiconductor device which fetches, decode and executes instructions
- **General-purpose registers:** Temporary storage location
- **Non-uniform memory access:** A computer memory design used in multiprocessing
- **Pipelining:** An effective method of increasing the execution speed of the processor
- **Array processor:** A CPU that implements an instruction set containing of tasks instructions
- **Kernel thread:** Unit of kernel scheduling
- **Symmetric multiprocessing:** The processing of programs by multiple processors that share a common operating system and memory
- **Port:** A buffered I/O

- Cluster: A group of linked computers
- Beowulf cluster: Commodity-grade computers
- Dynamic link library: A module that contains functions and data
- Grid computing: The federation of computer resources from multiple administrative domains to transmit the data

NOTES

5.7 ANSWERS TO ‘CHECK YOUR PROGRESS’

1. Examples of accumulator-based microprocessors are Intel 8085 and Motorola 6809.
2. 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.
3. An array processor is a processor that is designed for performing calculations on large-sized array of data. There are two types of array processor:
 - Attached Array Processor
 - SIMD Array Processor
4. The operation of the processing element is controlled by the master control unit, whose main function is to decode the instruction and how the instruction is to be executed.
5. A Linux Beowulf cluster is an example of a loosely coupled system.
6. A computer with a single processor is called Single Instruction Stream, Single Data Stream (SISD) computer.
7. The type of processor which may be generalized using a two-dimensional arrangement of processing elements. Such a structure is known as systolic processor.
8. VMEbus is a computer bus standard developed for the Motorola 68000 lines of CPUs.
9. The function of BOOT CPU is to control the bootstrap sequence and boot the other available CPUs.
10. An I/O adapter is used with the I/O link to communicate with a scanner port for a processor at another location in symmetric multiprocessor organization.
11. High-availability clusters, also known as failover cluster, are implemented primarily for the purpose of improving the availability of services that the cluster provides.
12. Load-balancing cluster is used when multiple computers are linked together to share computational workload or function as a single virtual computer.
13. Grid computing refers to the federation of computer resources from multiple administrative domains to transmit the data.
14. Athlon is a series of x86-compatible microprocessors designed and manufactured by AMD.

5.8 QUESTIONS AND EXERCISES

Parallel Organization

Short-Answer Questions

NOTES

1. What is a microprocessor?
2. Name the two types of registers.
3. What are the applications of parallel processing?
4. Write the examples of general-purpose register-based microprocessors.
5. What is the objective of attached array processor?
6. What is ILLIAC IV computer?
7. Write the limitation of loosely coupled systems.
8. Name the categories of Flynn's classification of computers.
9. What is a computer cluster?
10. Name the two types of clusters.
11. What is resource group?

Long-Answer Questions

1. Discuss the accumulator-based microprocessors with the help of illustration.
2. Explain the basics of parallel organization.
3. Discuss the parallel processing using parallel throughput.
4. What is pipelining processing? Explain with the help of examples.
5. Describe the Flynn's classification of computers with the help of examples.
6. Explain the symmetric multiprocessors with the help of examples and illustrations.
7. How the symmetric multiprocessor system is booted? Explain with the help of an example.
8. Discuss the structure of symmetric multiprocessor organization with the help of schematic diagram.
9. Describe the hierarchy of parallel organization with the help of illustration.
10. Explain Beowulf cluster with the help of suitable diagram.
11. Describe the hardware requirements for cluster configuration.

5.9 FURTHER READING

-
- Tanenbaum, Andrew S. 1999. *Structured Computer Organization*, 4th edition. New Jersey: Prentice-Hall Inc.
- Wilkinson. 1996. *Computer Architecture: Design and Performance*, 2nd edition. Hertfordshire: Prentice-Hall.
- Mano, M. Morris. 1993. *Computer System Architecture*, 3th edition. New Jersey: Prentice-Hall Inc.

Stallings, William. 2006. *Computer Organization and Architecture*, 7th edition. New Jersey: Prentice-Hall Inc.

Hamacher, V.C., Z.G. Vranesic and S.G. Zaky. 2002. *Computer Organization*, 5th edition. New York: McGraw-Hill International Edition.

Conte, T.M. and C.E. Gimarc. 1995. *Fast Simulation of Computer Architecture*. Boston: Kluwer Academic Publishers.

Gilmore, M. 1996. *Microprocessors: Principles and Applications*, 2nd edition. New York: McGraw-Hill.

NOTES

1. In general, parallel processing is a technique of dividing a task among multiple processors so that they can work on different parts of the task simultaneously. This is in contrast to sequential processing, where one processor performs the task sequentially.
2. A multiprocessor is a computer system that contains two or more processing units, each with its own memory, that can operate simultaneously.
3. In a multiprocessor system, each processor has its own local memory, which is used by that processor to store data and instructions.
4. In a multiprocessor system, each processor has its own local memory, which is used by that processor to store data and instructions.
5. In a multiprocessor system, each processor has its own local memory, which is used by that processor to store data and instructions.
6. In a multiprocessor system, each processor has its own local memory, which is used by that processor to store data and instructions.
7. In a multiprocessor system, each processor has its own local memory, which is used by that processor to store data and instructions.
8. In a multiprocessor system, each processor has its own local memory, which is used by that processor to store data and instructions.
9. In a multiprocessor system, each processor has its own local memory, which is used by that processor to store data and instructions.
10. In a multiprocessor system, each processor has its own local memory, which is used by that processor to store data and instructions.
11. In a multiprocessor system, each processor has its own local memory, which is used by that processor to store data and instructions.
-
- ## 2.0 RUMBLE MEDIAL
-
1. In a multiprocessor system, each processor has its own local memory, which is used by that processor to store data and instructions.
2. In a multiprocessor system, each processor has its own local memory, which is used by that processor to store data and instructions.
3. In a multiprocessor system, each processor has its own local memory, which is used by that processor to store data and instructions.
4. In a multiprocessor system, each processor has its own local memory, which is used by that processor to store data and instructions.
5. In a multiprocessor system, each processor has its own local memory, which is used by that processor to store data and instructions.
6. In a multiprocessor system, each processor has its own local memory, which is used by that processor to store data and instructions.
7. In a multiprocessor system, each processor has its own local memory, which is used by that processor to store data and instructions.
8. In a multiprocessor system, each processor has its own local memory, which is used by that processor to store data and instructions.
9. In a multiprocessor system, each processor has its own local memory, which is used by that processor to store data and instructions.
10. In a multiprocessor system, each processor has its own local memory, which is used by that processor to store data and instructions.
11. In a multiprocessor system, each processor has its own local memory, which is used by that processor to store data and instructions.
-

NOTES

NOTES

308

*Self-Instructional
Material*

Volume