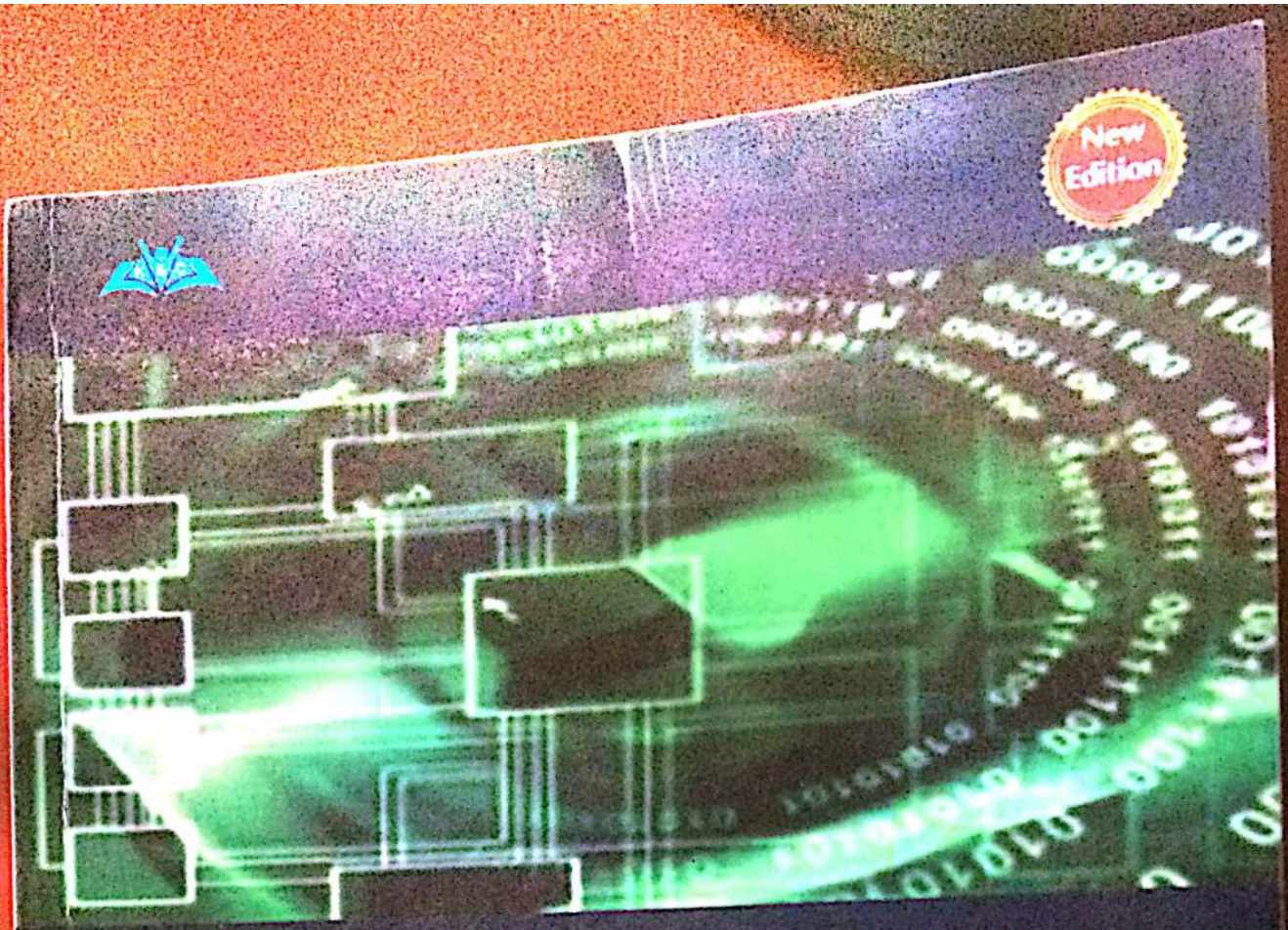


COMPUTER ARCHITECTURE & ORGANIZATION

B.Sc. CSIT/BIM

Er. Anil Kumar Sah

Computer Architecture & Organization

BSc.CSIT/BIM

Er. Anil Kumar Sah
Senior Lecturer
St. Xavier's College

KEC Publication and Distribution Pvt. Ltd.
Kathmandu, Nepal
Phone: 01-4168301, 01-4241777

Computer Architecture & Organization

Computer Architecture & Organization

BSc.CSIT/BIM Third Semester

Edition : First 2075

Copyright: Authors

Publisher & Distributor by: KEC Publication and Distribution Pvt. Ltd.

Kathmandu

☎01-4168301, 01-4241777

email: kecpublication14@gmail.com

Price Rs: 450.00

ISBN No.: 978 - 9937 - 724 - 11 - 1

Layout and design : Bikash Dhakal

Printed in Kathmandu

Preface

This book has been s
under Tribhuwan U
BE and BIT.

In this book, the co
Organization is conc
a computer system a
the various function
need of the user.

The objective of first
organization, design
advanced computer a

We highly expect tha
encourage our efforts
book, for which we w

Preface

This book has been specially designed as a text book for the students of BSc. CSIT and BIM programs under Tribhuwan University. This book is also expected to serve the need of the students of BCA, BE and BIT.

In this book, the concept of computer architecture and organization is explained. Computer Organization is concerned with the way the hardware components are connected together to form a computer system and computer architecture is concerned with the structure and behaviors of the various functional models of the computer and how they interact to provide the processing need of the user.

The objective of first five chapters of this book is to illustrate the basic understanding of computer organization, design and programming of basic computer. The last six chapters emphasize on advanced computer architecture and organization.

We highly expect that all our good wishers, contributors, students, teachers and colleagues will encourage our efforts. We hope for constant suggestion and provide us feedbacks to improve this book, for which we will be opened and highly thankful to all of them.

Author
Kartik 2075

BSc CSIT Syllabus

Full Marks: 60 + 20 + 20
Pass Marks: 24 + 8 + 8
Credit Hrs: 3

Course Title: Computer Architecture
Course No: CSC208
Nature of the Course: Theory + Lab
Semester: III

Course Description

This course includes concepts of instruction set architecture, organization or micro-architecture, and system architecture. The instruction set architecture includes programmer's abstraction of computer. The micro-architecture consist internal representation of computers at register and functional unit level. The system architecture includes organization of computers at the cache and bus level.

Course Objectives:

- Discuss representation of data and algorithms used to perform operations on data
- Demonstrate different operations in terms of Micro-operations
- Explain architecture of basic computer and micro-programmed control unit
- Understand and memory and I/O organization of a typical computer system
- Demonstrate benefits of pipelined systems

Course Contents: Unit 1: Data Representation (4 Hrs.)

1.1. Data Representation: Binary Representation, BCD, Alphanumeric Representation, Complements, Fixed Point representation, Representing Negative Numbers, Floating Point Representation, Arithmetic with Complements, Overflow, Detecting Overflow

1.2. Other Binary Codes: Gray Code, self Complementing Code, Weighted Code, Excess-3 Code, EBCDIC

1.3. Error Detection Codes: Parity Bit, Odd Parity, Even parity, Parity Generator & Checker

Unit 2: Register Transfer and Microoperations (5 Hrs.)

2.1. Microoperation, Register Transfer Language, Register Transfer, Control Function

2.2. Arithmetic Microoperations: Binary Adder, Binary Adder-subtractor, Binary Incrementer, Arithmetic Circuit

2.3. Logic Microoperations, Hardware Implementation, Applications of Logic Microoperations.

2.4. Shift Microoperations: Logical Shift, Circular shift, Arithmetic Shift, Hardware Implementation of Shifter.

Unit 3: Ba
3.1. Instr
3.2. Reg
3.3. Instr
Sign
3.4. Instr
Inpu
3.5. Des
Unit 4: Mi
4.1. Con
4.2. Add
Form
4.3. Desi
Unit 5: Ce
5.1. Maj
5.2. Instr
Subr
5.3. RISC
Unit 6: Pip
6.1. Para
6.2. Pipe
and
6.3. Instr
Cont
6.4. Vect
Unit 7: Co
7.1. Add
Com
7.2. Mult
Divi
Unit 8: Inp
8.1. Inpu
Map

Unit 3: Basic Computer Organization and Design (8 Hrs.)

3.1. Instruction Code, Operation Code, Stored Program Concept

3.2. Registers and memory of Basic Computer, Common Bus System for Basic Computer.

3.3. Instruction Format, Instruction Set Completeness, Control Unit of Basic Computer, Control Timing Signals

3.4. Instruction Cycle of Basic computer, Determining Type of Instruction, Memory Reference Instructions, Input-Output Instructions, Program Interrupt & Interrupt Cycle.

3.5. Description and Flowchart of Basic Computer

Unit 4: Microprogrammed Control (4 Hrs.)

4.1. Control Word, Microprogram, Control Memory, Control Address Register, Sequencer

4.2. Address Sequencing, Conditional Branch, Mapping of Instructions, Subroutines, Microinstruction Format, Symbolic Microinstructions

4.3. Design of Control Unit

Unit 5: Central Processing Unit (4 Hrs.)

5.1. Major Components of CPU, CPU Organization

5.2. Instruction Formats, Addressing Modes, Data Transfer and manipulation, Program Control, Subroutine Call and Return, Types of Interrupt

5.3. RISC vs CISC, Pros and Cons of RISC and CISC, Overlapped Register Windows

Unit 6: Pipelining (6 Hrs.)

6.1. Parallel Processing, Multiple Functional Units, Flynn's Classification

6.2. Pipelining: Concept and Demonstration with Example, Speedup Equation, Floating Point addition and Subtraction with Pipelining

6.3. Instruction Level Pipelining: Instruction Cycle, Three & Four-Segment Instruction Pipeline, Pipeline Conflicts and Solutions

6.4. Vector Processing, Applications, Vector Operations, Matrix Multiplication

Unit 7: Computer Arithmetic (6 Hrs.)

7.1. Addition and Subtraction with Signed Magnitude Data, Addition and Subtraction with Signed 2's Complement Data

7.2. Multiplication of Signed Magnitude Data, Booth Multiplication, Division of Signed magnitude Data, Divide Overflow

Unit 8: Input Output Organization (4 Hrs.)

8.1. Input-Output Interface: I/O Bus and Interface Modules, I/O vs. Memory Bus, Isolated vs. Memory-Mapped I/O

8.2. Asynchronous Data Transfer: Strobe, Handshaking

8.3. Modes of Transfer: Programmed I/O, Interrupt-Initiated I/O, Direct memory Access

8.4. Priority Interrupt: Polling, Daisy-Chaining, Parallel Priority Interrupt

8.5. Direct Memory Access, Input-Output Processor, DMA vs. IOP

UNIT 9: MEMORY ORGANIZATION (4 HRS.)

9.1 Memory Hierarchy, Main Memory, RAM and ROM Chips, Memory address Map, Memory Connection to CPU, Auxiliary Memory (magnetic Disk, Magnetic Tape)

9.1 Associative Memory: Hardware Organization, Match Logic, Read Operation, Write Operation

9.1 Cache Memory: Locality of Reference, Hit & Miss Ratio, Mapping, Write Policies

LABORATORY WORKS:

The laboratory work includes implementing and simulating the algorithms, studied in the course, by using high level languages like C or VHDL. The laboratory works should include at least following concepts;

- Simulate features like overflow, data representation by using VHDL
- Simulate design of different units by using VHDL
- Simulate pipelining by using VHDL
- Implement algorithms for computer arithmetic using high level language like C or C++

IT 217 : Com
Lecture Hou
COURSE OBJECT
This mo
used in
compute
COURSE DESCRI
Digital
designe
architec
Course Detai
1. DATA REPRES
1.1 Data Ty
1.2 Complen
1.3 Fixed Po
1.4 Floating
1.5 End of C
1.5.1 C
i
b
2. REGISTER TRAN
2.1 Register
2.2 Bus and
2.3 Arithmeti
2.4 Arithmeti
3. BASIC COMPUT
3.1 Instruction
3.2 Computer
3.3 Computer
3.4 Timing an
3.5 Instruction
3.6 Input Out
3.7 End of Ch
3.7.1 C
4. PROGRAMMING
4.1 Introductio
4.2 Machine L

BIM SYLLABUS

IT 217 : Computer Organization **Credit Hours: 3**

Lecture Hours: 48

COURSE OBJECTIVE:

This module aims to provide knowledge to understand the architecture of hardware components used in organization and design of digital computers and also design procedures of digital computers. Laboratory work is essential in this module.

COURSE DESCRIPTION:

Digital components used in the organization and design of digital computers, detailed steps that a designer must go through in order to design an elementary basic computer, the organization and architecture of CPU, I/O and Memory, Pipelining and Vector Processing, Concept of Multiprocessing.

Course Details

1. DATA REPRESENTATION — LH 3

1.1 Data Types

1.2 Complements

1.3 Fixed Point Representation

1.4 Floating Point Representation

1.5 End of Chapter 1 Assignment:

1.5.1 Computer Program: Write program to visualize the representation of complement numbers, integers, floating point numbers and character data, overflow detection while adding binary data.

2. REGISTER TRANSFER AND MICROOPERATIONS — LH 3

2.1 Register and Register Transfer Language

2.2 Bus and Memory Transfers

2.3 Arithmetic, Logic and Shift Micro-operations

2.4 Arithmetic Logic Shift Unit

3. BASIC COMPUTER ORGANIZATION AND DESIGN — LH 7

3.1 Instruction Codes

3.2 Computer Registers

3.3 Computer Instructions

3.4 Timing and Control

3.5 Instruction Cycle

3.6 Input Output and Interrupt

3.7 End of Chapter 2 and 3 Assignment:

3.7.1 Circuit Design: Design of Basic Computer

4. PROGRAMMING THE BASIC COMPUTER — LH 5

4.1 Introduction

4.2 Machine Language

4.3 Assembly Language
4.4 The Assembler
4.5 Program Loops
4.6 Programming Arithmetic and Logic Operations
4.7 Subroutines
4.8 Input Output Programming
4.9 End of Chapter 3 and 4 Assignment:
4.9.1 Computer Program: Develop assembler which can execute the program written for Basic Computer

5. Microprogrammed Control — LH3
5.1 Control Memory
5.2 Address Sequencing
5.3 Microprogram Example
5.3.1 Computer Configuration
5.3.2 Microinstruction Format

6. Central Processing Unit — LH4
6.1 CPU Organizations
6.2 Instruction Formats
6.3 Addressing Modes
6.4 RISC and CISC Characteristics
6.5 End of Chapter 5 and 6 Assignment:
6.5.1 Computer Program: Write program to illustrate the use of different addressing modes.

7. Pipeline and Vector Processing — LH6
7.1 Parallel Processing, Flynn's Classification of Computers
7.2 Pipelining
7.3 Arithmetic Pipeline
7.4 Instruction Pipeline
7.5 Pipeline Hazards and Their Solution
7.6 Array and Vector Processing
7.7 End of Chapter 7 Assignment:
7.7.1 Case Study: Available array and vector processor; their application domain
7.7.2 Computer Program: write program which simulates instruction pipeline and arithmetic pipeline. Program generates space time diagram considering pipeline hazards.

8. Computer Arithmetic — LH4
8.1 Addition and Subtraction of Signed Magnitude Data
8.2 Addition and Subtraction of Signed 2's Complement Data
8.3 Multiplication of Signed Magnitude Data
8.4 Multiplication of Signed 2's Complement Data
8.5 End of Chapter 8 Assignment:

8.5.1 Computer Program: Implement all algorithms learned in this chapter.

9. Input and Output Organization — LH 5

9.1 Peripheral Devices

9.2 I/O Interface

9.3 Asynchronous Data Transfer

9.4 Modes of Transfer

9.5 Priority Interrupt

9.6 Direct Memory Access

9.7 I/O Processor

9.8 End of chapter 9 Assignment:

9.8.1 Case Study: Device Independence I/O Software; USB (universal serial bus)

10. Memory Organization — LH 4

10.1 Memory Hierarchy

10.2 Main Memory

10.3 Associative Memory

10.4 Cache Memory

10.5 Virtual Memory

10.6 Memory Management Hardware

10.7 End of Chapter 10 Assignment:

10.7.1 Computer Program: write program to visualize the advantage of using memory hierarchy.

11. Multiprocessor — LH 4

11.1 Characteristics of Multiprocessor

11.2 Interconnection Structures

11.3 Inter Processor Arbitration

11.4 Inter Processor Communication and Synchronization

11.5 Cache Coherence

11.6 End of chapter 11 Assignment:

11.6.1 Case Study: ARM multicore processors and Intel multicore processors

11.6.2 Computer Program: Simulate cache coherence problem and simulate solution too.

Table of Content

CHAPTER 1

Data Representation

1.1 Introduction 2
1.2 Binary Arithmetic Operations 10
1.3 Compliments 11
1.4 Fixed Point Representation 14
1.5 Floating Point Representation 16
1.6 Other Binary Codes 18
1.7 Error Detection Code 21
1.8 Computer Architecture and Organization 23

CHAPTER 2

Register Transfer and Microoperations

2.1 Introduction 30
2.2 Register Transfer Language 30
2.3 Register Transfer 30
2.4 Bus and Memory Transfers 32
2.5 Arithmetic Microoperations 33
2.6 Logic Microoperations 37
2.7 Shift Microoperations 39
2.8 Arithmetic Logic Shift Unit 40

CHAPTER 3

Basic Computer Organization and Design

3.1 Instruction Code 44
3.2 Computer Registers 46
3.3 Computer Instructions Format with its Types 49
3.4 timing and Control Unit 51
3.5 Instruction Cycle 53
3.6 Memory reference instructions 56
3.7 Input-output configuration of basic computer 59
3.8 Complete Computer Description 63
3.9 Design of Basic Computer 65
3.10 Adder and Logic Circuit 69

CHAPTER 4

Programming the Basic Computer

4.1 Introduction 72
4.2 Machine Language 73
4.3 Assembly Language 75
4.4 The Assembler and the Compiler Programs 77
4.5 Program Loops 80
4.6 Programming Arithmetic and Logic Operations 82
4.7 Subroutines 87
4.8 Input-Output Programming 90

CHAPTER 5

Microprogrammed Control

5.1 Central Memory 106
5.2 Address Sequencing 108
5.3 Computer Hardware Configuration 111
5.4 Design of Control Unit 114

CHAPTER 6

Central Processing Unit

6.1 Introduction 120
6.2 General Register Organization 120
6.3 Stack organization 122
6.4 Instruction formats 125
6.5 ADDRESSING MODES 127
6.6 Data Transfer Instructions 129
6.7 RISC and CISC 136

CHAPTER 7

Pipelining and Vector Processing

7.1 Parallel Processing 142
7.2 Pipelining 143
7.3 Arithmetic pipeline 146
7.4 Instruction Pipeline 147
7.5 RISC Pipeline 149
7.6 Vector Processing 151
7.7 Array Processors 153

CHAPTER 8

Computer Arithmetic

8.1 Introduction 156
8.2 Addition and Subtraction 156
8.3 Multiplication Algorithm 159
8.4 Division Algorithms 165

CHAPTER 9

Input/Output Organization

9.1 Introduction to Peripheral Devices 174
9.2 Input-Output Interface 174
9.3 Modes of Transfer 177
9.4 Input Output Processor (I/O Channel) 179
9.5 Serial and Parallel Communication 181

CHAPTER 10

Memory Organization

10.1 Memory Hierarchy 188
10.2 Main Memory 189
10.3 Auxiliary Memory 191
10.4 Associative Memory: Content Addressable Memory (CAM) 192
10.5 Cache Memory 194
10.6 Virtual Memory 197
10.7 Memory Management Hardware 202

CHAPTER 11

Multiprocessor

11.1 Introduction 208
11.2 Interconnection Structures 210
11.3 Inter processor Arbitration(IA) 214
11.4 Interprocessor Communication and Synchronization (ICS) 218
11.5 Cache Coherence 219

Bibliography 223
TU Exam Question 224
Chapterwise Exercises 225

Chapter 1

Data Representation

Objective

This chapter shows how the various data types found in digital computers are represented in binary form in computer registers. Emphasis is on the representation of numbers employed in Arithmetic operations, and on the binary coding of symbols used in data processing.

OUTLINE

1.1 Introduction
1.2 Complements
1.3 Fixed-Point Representation
1.4 Floating-Point Representation
1.5 Other Binary Codes
1.6 Error Detection Codes
1.7 Introduction to computer architecture and organization

1.1 INTRODUCTION

Computers are classified according to **functionality, physical size** and **purpose**. Functionality, Computers could be **analog, digital or hybrid**. Digital computers process data that is in discrete form whereas analog computers process data that is continuous in nature. Hybrid computers on the other hand can process data that is both discrete and continuous. In digital computers, the user input is first converted and transmitted as electrical pulses that can be represented by two unique states ON and OFF. The ON state may be represented by a "1" and the off state by a "0".The sequence of ON'S and OFF'S forms the electrical signals that the computer can understand. A digital signal rises suddenly to a peak voltage of +1 for some time then suddenly drops -1 level on the other hand an analog signal rises to +1 and then drops to -1 in a continuous version.

Although the two graphs look different in their appearance, notice that they repeat themselves at equal time intervals. Electrical signals or waveforms of this nature are said to be periodic. Generally, a periodic wave representing a signal can be described using the following parameters

- Amplitude(A)
- Frequency(f)
- periodic time(T)
  - **Amplitude (A):** this is the maximum displacement that the waveform of an electric signal can attain.
  - **Frequency (f):** is the number of cycles made by a signal in one second. It is measured in hertz.1hert is equivalent to 1 cycle/second.
  - **Periodic time (T):** the time taken by a signal to complete one cycle is called periodic time. Periodic time is given by the formula **T=1/f,** where **f** is the frequency of the wave.
  - When a digital signal is to be sent over analog telephone lines e.g. e-mail, it has to be converted to analog signal. This is done by connecting a device called a **modem** to the digital computer. This process of converting a digital signal to an analog signal is known as **modulation**. On the receiving end, the incoming analog signal is converted back to digital form in a process known as **demodulation**.

Concepts of Data Representation in Digital Computers

Data and instructions cannot be entered and processed directly into computers using human language. Any type of data be it numbers, letters, special symbols, sound or pictures must first be converted into machine-readable form i.e. binary form. Due to this reason, it is important to understand how a computer together with its peripheral devices handles data in its electronic circuits, on magnetic media and in optical devices.

Data representation in digital circuits

Electronic components, such as microprocessor, are made up of millions of electronic circuits. The availability of high voltage(on) in these circuits is interpreted as '1' while a low voltage (off) is interpreted as '0'.This concept can be compared to switching on and off an electric circuit. When the switch is closed the high voltage in the circuit causes the bulb to light ('1' state).on the

other hand when the switch is open, the bulb goes off ('0' state). This forms a basis for describing data representation in digital computers using the binary number system.

Data representation on magnetic media

The laser beam reflected from the land is interpreted, as 1.The laser entering the pot is not reflected. This is interpreted as 0.The reflected pattern of light from the rotating disk falls on a receiving photoelectric detector that transforms the patterns into digital form. The presence of a magnetic field in one direction on magnetic media is interpreted as 1; while the field in the opposite direction is interpreted as "0".Magnetic technology is mostly used on storage devices that are coated with special magnetic materials such as iron oxide. Data is written on the media by arranging the magnetic dipoles of some iron oxide particles to face in the same direction and some others in the opposite direction

Data representation on optical media

In optical devices, the presence of light is interpreted as '1' while its absence is interpreted as '0'.Optical devices use this technology to read or store data. Take example of a CD-ROM, if the shiny surface is placed under a powerful microscope, the surface is observed to have very tiny holes called **pits**. The areas that do not have pits are called **land**.

Reason for Use of Binary System in Computers

It has proved difficult to develop devices that can understand natural language directly due to the complexity of natural languages. However, it is easier to construct electric circuits based on the binary or ON and OFF logic. All forms of data can be represented in binary system format. Other reasons for the use of binary are that digital devices are more reliable, small and use less energy as compared to analog devices.

Bits, bytes, nibble and word

- The terms bits, bytes, nibble and word are used widely in reference to computer memory and data size.
- **Bits:** can be defined as either a binary, which can be 0, or 1.It is the basic unit of data or information in digital computers.
- **Byte:** a group of bits (8 bits) used to represent a character. A byte is considered as the basic unit of measuring memory size in computer.
- **A nibble**: is half a byte, which is usually a grouping of 4 bytes.
- **Word:** two or more bits make a word. The term **word length** is used as the measure of the number of bits in each word. For example, a word can have a length of 16 bits, 32 bits, 64 bits etc.

Types of Data Representation

Computers not only process numbers, letters and special symbols but also complex types of data such as sound and pictures. However, these complex types of data take a lot of memory and processor time when coded in binary form. This limitation necessitates the need to develop better ways of handling long streams of binary digits. Higher number systems are used in computing to reduce these streams of binary digits into manageable form. This helps to improve the processing speed and optimize memory usage.

Number systems and their representation

- A **number system** is a set of symbols used to represent values derived from a common base or radix.
- As far as computers are concerned, number systems can be classified into two major categories:
- decimal number system
- binary number system
- octal number system
- hexadecimal number system

Decimal number system

The term decimal is derived from a Latin prefix deci, which means ten. Decimal number system has ten digits ranging from 0-9. Because this system has ten digits; it is also called a base ten number system or denary number system. A decimal number should always be written with a subscript 10 e.g. X_{10}. But since this is the most widely used number system in the world, the subscript is usually understood and ignored in written work. However ,when many number systems are considered together, the subscript must always be put so as to differentiate the number systems.

The magnitude of a number can be considered using these parameters.

- Absolute value
- Place value or positional value
- Base value
  - **The absolute** value is the magnitude of a digit in a number. for example the digit 5 in 7458 has an absolute value of 5 according to its value in the number line.
  - **The place value** of a digit in a number refers to the position of the digit in that number i.e. whether; tens, hundreds, thousands etc.
  - **The total value** of a number is the sum of the place value of each digit making the number.
  - **The base value** of a number also k known as the **radix**, depends on the type of the number systems that is being used .The value of any number depends on the radix. for example the number 100_{10} is not equivalent to 100_2.

Binary Number System

It uses two digits namely, 1 and 0 to represent numbers. unlike in decimal numbers where the place value goes up in factors of ten, in binary system, the place values increase by the factor of 2.binary numbers are written as X_2.consider a binary number such as 1011_2.The right most digit has a place value of 1×2^0 while the left most has a place value of 1×2^3.

Octal Number System

Consists of eight digits ranging from 0-7.the place value of octal numbers goes up in factors of eight from right to left.

Hexadecimal Number System

This is a base 16 number system that consists of sixteen digits ranging from 0-9 and letters A-F where A is equivalent to 10,B to 11 up to F which is equivalent to 15 in base ten system. The place value of hexadecimal numbers goes up in factors of sixteen.

- A hexadecimal number can be denoted using 16 as a subscript or capital letter H to the right of the number .For example, 94B can be written as 94B16 or 94BH.

Further conversion of numbers from one number system to another

- To convert numbers from one system to another. The following conversions will be considered.
- Converting between binary and decimal numbers.
- Converting octal numbers to decimal and binary form.
- Converting hexadecimal numbers to decimal and binary form.

Converting binary numbers to decimal numbers

- To convert a binary number to a decimal number, we proceed as follows:
- First, write the place values starting from the right hand side.
- Write each digit under its place value.
- Multiply each digit by its corresponding place value.
- Add up the products. The answer will be the decimal number in base ten.

Example

Convert 101101_2 to base 10(or decimal) number

Place value	2^5	2^4	2^3	2^2	2^1	2^0
Binary digits	1	0	1	1	0	1

Multiply each digit by its place value

N_{10} $=(1*2^5)+(0*2^4)+(1*2^3)+(1*2^2)+(0*2^1)+(1*2^0)$

N_{10} $=32+0+8+4+0+1$

$=45_{10}$

$32*1$ $=32$

$16*0$ $=0$

$8*1$ $=8$

$4*1$ $=4$

$2*0$ $=0$

$1*1$ $=1$

$=45_{10}$

NB: *remember to indicate the base subscript since it is the value that distinguishes the different systems.*

- The binary equivalent of the fractional part is extracted from the products by reading the respective integral digits from the top downwards as shown by the arrow next page.
- Combine the two parts together to set the binary equivalent.

Convert 0.375_{10} into binary form

Read this digits

$0.375 \times 2 = 0.750$

$0.750 \times 2 = 1.500$

$0.500 \times 2 = 1.000$ (fraction becomes zero)

Therefore $0.375_{10} = 0.011_2$

NB: *When converting a real number from binary to decimal, work out the integral part and the fractional parts separately then combine them.*

Example

Convert 11.011_2 to a decimal number.

Solution

Convert the integral and the fractional parts separately then add them up.

$2 \times 1 = 2.000$

$1 \times 1 = +1.000$

3.000_{10}

Weight	2^1	2^0 .	2^{-1}	2^{-2}	2^{-3}
Binary digit	1	1 .	0	1	1
Values in base 10	2	1 .	0	0.25	0.125

$0.50 \times 0 = 0.000$

$0.25 \times 1 = 0.250$

$0.125 \times 1 = +0.125$

0.375_{10}

$3.000_{10} + 0.375_{10} = 3.375_{10}$

Thus $11.011_2 = 3.375_{10}$

Converting a decimal fraction to binary

Divide the integral part continuously by 2.For the fractional part, proceed as follows:

Multiply the fractional part by 2 and note down the product

- Take the fractional part of the immediate product and multiply it by 2 again.
- Continue this process until the fractional part of the subsequent product is 0 or starts to repeat itself.

The following examples illustrate how to convert hexadecimal number to a decimal number

Example

Convert octal number 321_8 to its binary equivalent

Solution

Working from left to the right, each octal number is represented using three digits and then combined we get the final binary equivalent. Therefore:

$3 = 011_2$

$2 = 010_2$

$1 = 001_2$

Combining the three from left to right

3 2 1

011 010 001

$321_8 = 011010001_2$

Converting binary numbers to hexadecimal numbers

- To convert binary numbers to their binary equivalents, simply group the digits of the binary number into groups of four from right to left e.g. 11010001.The next step is to write the hexadecimal equivalent of each group e.g.

  1101- D

  0001- 1

The equivalent of 11010001 is D1H or $D1_{16}$

Converting hexadecimal numbers to decimal number

- To convert hexadecimal number to base 10 equivalent we proceed as follows:
  - First, write the place values starting from the right hand side.
  - If a digit is a letter such as 'A' write its decimal equivalent
- Multiply each hexadecimal digit with its corresponding place value and then add the products

The binary equivalent of the fractional part is extracted from the products by reading the respective integral digits from the top downwards as shown by the arrow next page. Combine the two parts together to set the binary equivalent.

Convert 0.375_{10} into binary form

Read this digits

$0.375 \times 2 = 0.750$

$0.750 \times 2 = 1.500$

$0.500 \times 2 = 1.000$ (fraction becomes zero)

Therefore $0.375_{10} = 0.011_2$

Converting octal numbers to decimal numbers

- To convert a base 8 number to its decimal equivalent we use the same method as we did with binary numbers. However, it is important to note that the maximum absolute value of a octal digit is 7.For example 982 Is not a valid octal number because digit 9 is not an octal digit, but 736_8 is valid because all the digits are in the range 0-7.Example shows how to convert an octal number to a decimal number.

Example

Convert 512_8 to its base 10 equivalent

Solution

Place value 8^2 8^1 8^0

64 8 1

Octal digit 5 1 2

Write each number under its place value as shown below

Multiply each number by its place value.

$N_{10} = (5 \times 8^2)+(1 \times 8^1)+(2 \times 8^0)$

$=(5 \times 64)+8+2$

$=320+8+2$

$N_{10} = 330_{10}$

$64 \times 5=320$

$8 \times 1= 8$

$1 \times 2=+ 2$

330

Therefore$512_8 = 330_{10}$

Converting octal numbers to binary numbers

To convert an octal number to binary, each digit is represented by three binary digits because the maximum octal digit i.e. 7 can be represented with a maximum of seven digits. See table:

Octal digit	Binary equivalents
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Example

Convert the hexadecimal number 111_{16} to its binary equivalent.

Solution

Place each number under its place value.

$16^2\ 16^1\ 16^0$

1 1 1

$256 \times 1= 256$

$16 \times 1 = 16$

$1 \times 1= + 1$

273

Therefore $111_{16} = 273_{10}$

Example

Convert octal number 321_8 to its binary equivalent

Solution

Working from left to the right, each octal number is represented using three digits and then combined we get the final binary equivalent. Therefore:

$3=011_2$

$2=010_2$

$1=001_2$

Combining the three from left to right

3 2 1

011 010 001

$321_8=011010001_2$

Converting binary numbers to hexadecimal numbers

To convert binary numbers to their binary equivalents, simply group the digits of the binary number into groups of four from right to left e.g. 11010001.The next step is to write the hexadecimal equivalent of each group e.g.

1101- D

0001- 1

The equivalent of 11010001 is D1H or $D1_{16}$

Converting hexadecimal numbers to decimal number

To convert hexadecimal number to base 10 equivalent we proceed as follows:

First, write the place values starting from the right hand side.

- If a digit is a letter such as 'A' write its decimal equivalent
  - Multiply each hexadecimal digit with its corresponding place value and then add the products
  - The following examples illustrate how to convert hexadecimal number to a decimal number

Example

Convert the hexadecimal number 111_{16} to its binary equivalent

Solution

Place each number under its place value.

$16^2 16^1 16^0$

1 1 1

256 x1= 256

16 x 1 = 16

1 x 1= + 1

273

Therefore $111_{16}=273_{10}$

Converting Hexadecimal Numbers to Binary Numbers

- Since F is equivalent to a binary number$1111_2$the hexadecimal number are therefore represented using4 digits as shown in the table below:

Hexadecimal digit	Decimal equivalent	Binary equivalent
00	00	0000
01	01	0001
02	02	0010
03	03	0011
04	04	0100
05	05	0101
06	06	0110
07	07	0111
08	08	1000
09	09	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

The simplest method of converting a hexadecimal number to binary is to express each hexadecimal digit as a four bit binary digit number and then arranging the group according to their corresponding positions as shown in example

Example

Convert 321_{16}

Hexadecimal digit 3 2 1

Binary equivalent 0011 0010 0001

Combining the three sets of bits, we get 001100100001_2

$321_{16} = 001100100001_2$

Example 2

Convert $5E6_{16}$ into binary

Hexadecimal digit

5 E 6

Binary equivalent 0101 1110 0110

$5E616 = 010111100110_2$

1.2 Binary Arithmetic Operations

In mathematics, the four basic arithmetic operations applied on numbers are addition, subtraction, multiplications and division. In computers, the same operations are performed inside the central processing unit by the arithmetic and logic unit (ALU). However, the arithmetic and logic unit cannot perform binary subtractions directly. It performs binary subtractions using a process known as multiplication and division, the arithmetic and logic unit uses a method called *shifting* before adding the bits.

Representation of Signed Binary Numbers

- In computer technology, there are three common ways of representing a signed binary number.
  - Prefixing an extra sign bit to a binary number.
  - Using ones compliment.
  - Using twos compliment.

Prefixing an Extra Sign Bit to a Binary Number

In decimal numbers, a signed number has a prefix "+" for a positive number e.g. $+27_{10}$ and "-" for a negative number e.g. -27. However, in binary, a negative number may be represented by prefixing a digit 1 to the number while a positive number may be represented by prefixing a digit 0. For example, the 7-bit binary equivalent of 127 is 1111111_2. To indicate that it is positive, we add an extra bit (0) to the left of the number i.e. $(0)1111111_2$.

To indicate that it is negative number we add an extra bit (1) i.e. $(1)1111111_2$.

The problem of using this method is that the zero can be represented in two ways i.e.$(0)0000000_2$ and $(1)0000000_2$.

1.3 COMPLIMENTS

The term compliment refers to *a part* which together with another makes up a *whole*. For example in geometry two complimentary angle (90^0). The idea of compliment is used to address the problem of signed numbers i.e. positive and negative.

One's Compliment

In decimal numbers (0 to 9), we talk of nine's compliment. For example the nines compliment

Of 9 is 0, that of 5 is 4 while 3 is 6. However, in binary numbers, the ones compliment is the bitwise NOT applied to the number. Bitwise NOT is a unary operator (operation on only one operand) that performs logical negation on each bit. For example the bitwise NOT of 11002 is 00112e. 0's are negated to 1's while 1's are negated to 0's.

Two's compliment

Twos compliment, equivalent to tens compliment in decimal numbers, is the most popular way of representing negative numbers in computer systems. The advantages of using this method are:

- There are no two ways of representing a zero as in the case with other two methods.
- Effective addition and subtraction can be done even with numbers that are represented with a sign bit without a need for circuitries to examine the sign of an operand.

The twos compliment of a number is obtained by getting the ones compliment then adding a 1. For example, to get the twos compliment of a decimal number 45_{10},

First convert it to its binary equivalent then find its ones compliment. Add a 1 to ones compliment i.e.

$45_{10} = 00101101_2$

Bitwise NOT (00101101) $= 11010010$

Two's compliment $= 11010010_2 + 1_2$

$= 11010011_2$

Binary addition

The five possible additions in binary are

$0 + 0 = 0$

$0 + 1_2 = 1_2$

$1_2 + 0 = 1_2$

$1_2 + 1_2 = 10_2$ (read as 0, carry 1)

$1_2 + 1_2 + 1_2 = 11_2$ (read as 1, carry 1)

Example 1

Find the sum of $111_2 + 011_2$

Solution

Arrange the bits vertically. 111

Working from the right to the left, we proceed as follows: + 011

Step 1 $1_2 + 1_2 = 10_2$, (write down 0 and carry 1) 1010_2

Step 2 $1_2 + 1_2 + 1_2 = 11_2$, (add and carry over digit to 1 + 1 in order to get 1 + 1 +1. From the sum, write down digit one the carry Forward)

Step 3 $1_2 + 1_2 + 0_2 = 10_2$, (add the carry over digit to 1 + 0 in order to get 1 + 1 + 0.since this is the last step, write down 10)

Therefore $111_2 + 011_2 = 1010_2$

This can be summarized in the table

1st number	1	1	1	
2nd number	0	1	1	
Carry digit -		1	1	
sum	1	0	1	0

Example 2

Add the following binary number

10110_2

1011_2

$+ 111_2$

Solution

Add the first two numbers and then add the sum to the third number a follows:

Step 1	**Step 2**
10110_2	100001_2
$+ 1011_2$	$+ 111_2$
100001_2	101000_2

Binary Subtraction

Direct subtraction

The four possible subtractions in binary are:

1. $0 - 0 = 0$
2. $1_2 - 0 = 1_2$
3. $1_2 - 1_2 = 0$
2. $10_2 - 1_2 = 1_2$ (borrow 1 from the next most significant digit to make 0 become 10_2, Hence, $10_2 - 1_2 = 1_2$)

Subtraction using one's compliment

The main purpose of using ones compliment in computers is to perform binary subtraction. For example to get the difference in 5 – 3, using the ones compliment, we proceed as follows:

1. Rewrite the problem as 5 + (-3) to show that he computer binary subtraction by adding the binary equivalent of 5 to ones compliment of 3.
2. Convert the absolute value of 3 into 8-bits equivalent i.e. 00000011_2.
3. Take the ones compliment of 00000011_2e. 11111100_2 which is the binary representation of -3_{10}.
4. Add the binary equivalent of 5 to ones compliment of 3 i.e.

   00000101
   + 11111000
   (1)00000001

Subtraction Using Two's Compliments

Like in ones compliment, the twos compliment of a number is obtained by negating a positive number to is negative counterpart. For example to get the difference in 5-3, using twos compliment, we proceed as follow:

1. Rewrite the problem as 5 + (-3)
2. Convert the absolute value of 3 into 8-bit binary equivalent i.e. 00000011.
3. Take the ones compliment of 00000011 i.e. 11111100.
4. add a 1 to the ones compliment i.e. 11111100 to get 11111101
5. add he binary equivalent of 5 to the twos compliment of 3 i.e.

   00000101
   + 11111001

(1)00000010 Ignoring the overflow bit, the resulting number is 00000010, which is directly read as a binary equivalent of +2.

Example

Using twos compliment

$31_{10} - 17_{10}$ in binary form.

Solution

17_{10} in binary 00010001
1's compliment 11101110
2's compliment 11101111
$31_{10} = 00011111_2$
$00011111 + 11101111 = (1)00001110_2$

1.4 Fixed Point Representation

Positive integers, including zero, can be represented as unsigned numbers. However, to represent negative integers, we need a notation for negative values. In ordinary arithmetic, a negative number is indicated by a minus sign and a positive number by a plus sign. Because of hardworking limitations, computers must represent everything with 1's and 0's, including the sign of a number. As a consequence, it is customary to represent the sign with a bit placed in the leftmost position of the number. The convention is to make the sign bit equal to 0 for positive and to 1 for negative.

In addition to the sign, a number may have a binary (or decimal) point. The position of the binary point is needed to represent fractions, integers, or mixed integer-fraction numbers. The representation of the binary point in the register is complicated by the fact that it is characterized by a position in the register. There are two ways of specifying he position of the binary point in a register: by giving it a fixed position or by employing a floating point representation. The fixed point method assumes that the binary point is always fixed on one position. The two positions most widely used are (1) a binary point in the extreme left of the register to make the stored number a fraction, and (2) a binary point in the extreme right case, the binary point is not actually present, but its presence is assumed from the fact that the number stored in the register is treated as a fraction or as an integer. The floating-point representation uses a second register to store a number that designates the position of the decimal point in the first register. Floating-point representation is discussed further in the next section.

Binary Integer Representation

	12	–12
Signed-magnitude	0 00001100	1 0001100
Signed-1's complement	0 00001100	1 1110011
Signed-2's complement	0 00001100	1 1110100

Computer Arithmetic

- Signed-magnitude awkward
- Signed-1's complement has two representations for zero: – 0 and + 0
- Therefore, signed-2's complement is used

Arithmetic Addition

- Simply add the two numbers and discard any leftmost carry out bit
- Negative numbers are in 2's complement
  - Including the result!

Example

+7	0000 0111	-7	1111 1001
+11	0000 1011	+11	0000 1011
+18	0001 0010	+4	0000 0100

+7	0000 0111	-7	1111 1001
-11	1111 0101	-11	1111 0101
-4	1111 1100	-18	1110 1110

Arithmetic Subtraction

- minuend - subtrahend = result
- minuend + subtrahend's r's complement = result
- Simply add the two numbers and discard any leftmost carry out bit
- Negative numbers are in 2's complement
  - Including the result!

Example

- (-5) - (-14) = 9
- (1111 1011) - (1111 0010) = 0000 1001
- (1111 1011) + (0000 1110) = 0000 1001
- $\boxed{1}$0000 1001 = 0000 1001

Overflow Examples

+79	0 100 1111	-79	1 011 0001
+81	0 101 0001	-81	1 010 1111
+160	1 010 0000	-160	0 110 0000

Carry-in = 1
Carry-out = 0

Carry-in = 1
Carry-out = 0

Decimal Fixed-Point Representation

The representation of decimal numbers in register is a function of the binary code used to represent a decimal digit. A 4-bit decimal code requires four flip-flops for each decimal digit. The representation of 4385 in BCD requires 16 flip-flops, four flip-flops for each digit. The number will be represented in a register with 16 flip-flops as follows:

0100 0011 1000 0101

The procedures developed for the signed-2's complement system apply also to the signed-10's complement system for decimal numbers. Addition is done by adding all digits, including the sign digit, and discarding the end carry. Obviously, this assumes that all negative numbers are in 10's complement form. Consider the addition (+375) + (-240) = +135 done in the signed 10's complement system.

	0 375	(0000 0011 0111 0101)$_{BCD}$
+	9 760	(1001 0111 0110 0000)$_{BCD}$
	0 135	(0000 0001 0011 0101)$_{BCD}$

The subtraction of decimal numbers either unsigned or in the signed-10's complement system is the same as in the binary cases. Take the 10's complement of the subtrahend and add it to the minuend. Many computers have special hardware to perform arithmetic calculations directly with decimal numbers in BCD. The user of the computer can specify by programmed instructions that the arithmetic operations be performed with decimal numbers directly without having to convert them to binary.

1.5 Floating Point Representation

Floating point is useful for representing a number in a wide range: very small to very large. It is widely used in the scientific world. Consider, the following FP representation of a number

Sign bit	Exponent E	significand F (also called mantissa)
+/-	x x x x	y y y y y y y y y y y y

In decimal it means (+/-) 1. yyyyyyyyyyyy × 10^{xxxx}

In binary, it means (+/-) 1. yyyyyyyyyyyy × 2^{xxxx}

(The 1 is implied)

IEEE 754 single-precision (32 bits)

s	xxxxxxxx	yyyyyyyyyyyyyyyyyyyyyyy
1	8	23 bits

Single precision

Largest = 1. 1 1 1 ... x 2 +127 ≈ 2 x 10 +38

Smallest = 1.000 ... x 2 –128 ≈ 1 x 10 -38

These can be positive and negative, depending on s. (But there are exceptions too)

IEEE 754 double precision (64 bits)

S	exponent	significand
1	11 bits	52 bits

Largest = 1. 1 1 1... x 2 +1023

Smallest = 1.000... X 2 –1024

Overflow and underflow in FP

An overflow occurs when the number if too large to fit in the frame. An underflow occurs when the number is too small to fit in the given frame.

How do we represent zero?

IEEE standards committee solved this by making zero a special case: if every bit is zero (the sign bit being irrelevant), then the number is considered zero.

Then how do we represent 1.0?

It should have been 1.0 x 20 (same as 0)! The way out of this is that the interpretation of the exponent bits is not straightforward. The exponent of a single-precision float is "shift- 127" encoded (biased representation), meaning that the actual exponent is (xxxxxxx minus 127). So thankfully, we can get an exponent of zero by storing 127.

Exponent = 11111111 (i.e. 255) means 255-127 = 128

Exponent = 01111111 (i.e. 127) means 127-127 = 0

Exponent = 00000001 (i.e. 1) means 1-127 = -126

The consequence of shift-127

Exponent = 00000000 (reserved for 0) can no more be used to represent the smallest number. We forego something at the low extreme of the spectrum of representable magnitudes, which should be 2-127

It seems wise, to give up the smallest exponent instead of giving up the ability to represent 1 or zero!

More special cases

Zero is not the only "special case" float. There are also representations for positive and negative infinity, and for a not-a-number (NaN) value, for results that do not make sense (for example, non-real numbers, or the result of an operation like infinity times zero). How do these work? A number is infinite if every bit of the exponent is 1 (yes, we lose another one), and is NaN if every bit of the exponent is 1 plus any mantissa bits are 1. The sign bit still distinguishes +/-inf and +/-NaN. Here are a few sample floating point representations:

Exponent	Mantissa	Object
0	0	Zero
0	Nonzero	Denormalized number*
1-254	Anything	+/- FP number
255	0	+ / - infinity
255	Nonzero	NaN

* Any non-zero number that is smaller than the smallest normal number is a denormalized number. The production of a denormal is sometimes called gradual underflow because it allows a calculation to lose precision slowly when the result is small.

Floating Point Addition

Example using decimal

A = 9.999 x 10 1, B = 1.610 x 10 –1, A+B =?

Step 1. Align the smaller exponent with the larger one.

B = 0.0161 x 101 = 0.016 x 101 (round off)

Step 2. Add significands

9.999 + 0.016 = 10.015, so A+B = 10.015 x 101

Step 3. Normalize

A+B = 1.0015 x 102

Step 4. Round off

A+B = 1.002 x 102

Now, try to add 0.5 and -0.4375 in binary.

Floating Point Multiplication

Example using decimal

A = 1.110 x 1010, B = 9.200 x 10-5 A x B =?

Step 1. Exponent of A x B = 10 + (-5) = 5

Step 2. Multiply significands 1.110x 9.200 = 10.212000

Step 3. Normalize the product 10.212 x 105 = 1.0212 x 106

Step 4. Round off

A x B = 1.021 x 106

Step 5. Decide the sign of A x B (+ x + = +) So, A x B = + 1.021 x 106

1.6 Other Binary Codes

In computing, a single character such as a letter, a number or a symbol is represented by a group of bits. The number of bits per character depends on the *coding* scheme used. The most common coding schemes are: Binary Coded Decimal, Gray Code, Self Complementing Code, Weighted Code, Excess-3 Code, Extended Binary Coded Decimal Interchange Code (EBCDIC) and American Standard Code for Information Interchange (ASCII).

Binary Coded Decimal

Binary Coded Decimal is a 4-bit code used to represent numeric data only. For example, a number like 9 can be represented using Binary Coded Decimal as 1001_2. It is mostly used in simple electronic devices like calculators and microwaves. This is because it makes it easier to process and display individual numbers on their Liquid Crystal Display (LCD) screens.

A standard Binary Coded Decimal, an enhanced format of Binary Coded Decimal, is a 6-bit representation scheme which can represent non-numeric characters. This allows 64 characters to be represented. For letter A can be represented as 110001_2 using standard Binary Coded Decimal.

Gray Code:

It was named after Frank Gray and was used as solution guide for tower of Hanoi problem. It is used in labeling the axes of K-maps and allows system designer to perform error checking.

Conversion from BCD to Gray

- MSB of the gray code and the binary code is the same.
- Going from left to right, add each adjacent pair of binary code bits to get the next gray code bit discarding the carry obtained.

Input				Output			
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	1
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1

Table1.1 *Truth table for BCD to Gray code conversion*

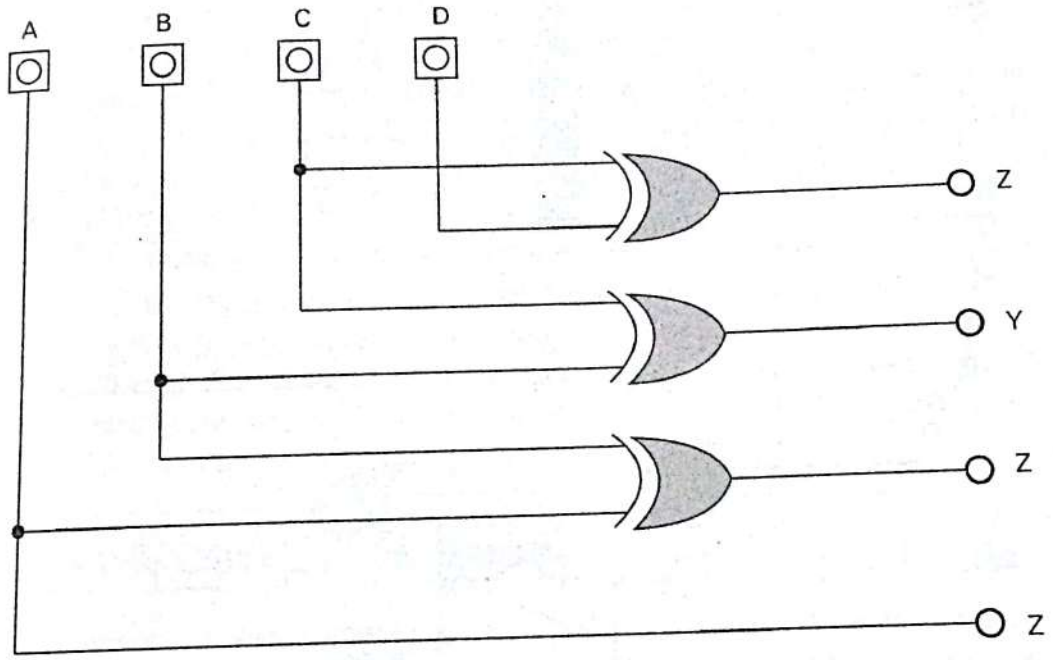


Figure 1.1 *Logic Diagram for BCD to Gray code conversion*

Self Complementing Code

The 2421, the excess - 3 and the 84-2-1 codes are examples of self - complementing codes. Such codes have the property that the 9's complement of a decimal number is obtained directly by changing 1's to 0's and 0's to 1's (i.e., by complementing each bit in the pattern). For example, decimal 395 is represented in the excess - 3 code as 0110 1100 1000. The 9's complement of 604 is represented as 1001 0011 0111, which is obtained simply by complementing each bit of the code (as with the 1's complement of binary numbers).

Excess-3 Code:

Excess- 3 code is also known as (XS-3), Stibitz code, biased representation or Excess-N and is complementary BCD code. It is used in older computers with a pre-specified number N and is

represented by four bits as BCD value plus 3. The primary advantage of XS-3 coding over BCD coding is that a decimal number can be nines' complemented (for subtraction) as easily as a binary number can be ones' complemented

Truth Table

Input BCD				Output (Excess - 3)			
A	B	C	D	W	X	Y	Z
0	0	0	0	0	1	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

Table1.2 : *Truth Table For BCD t oExcess-3 code conversion.*

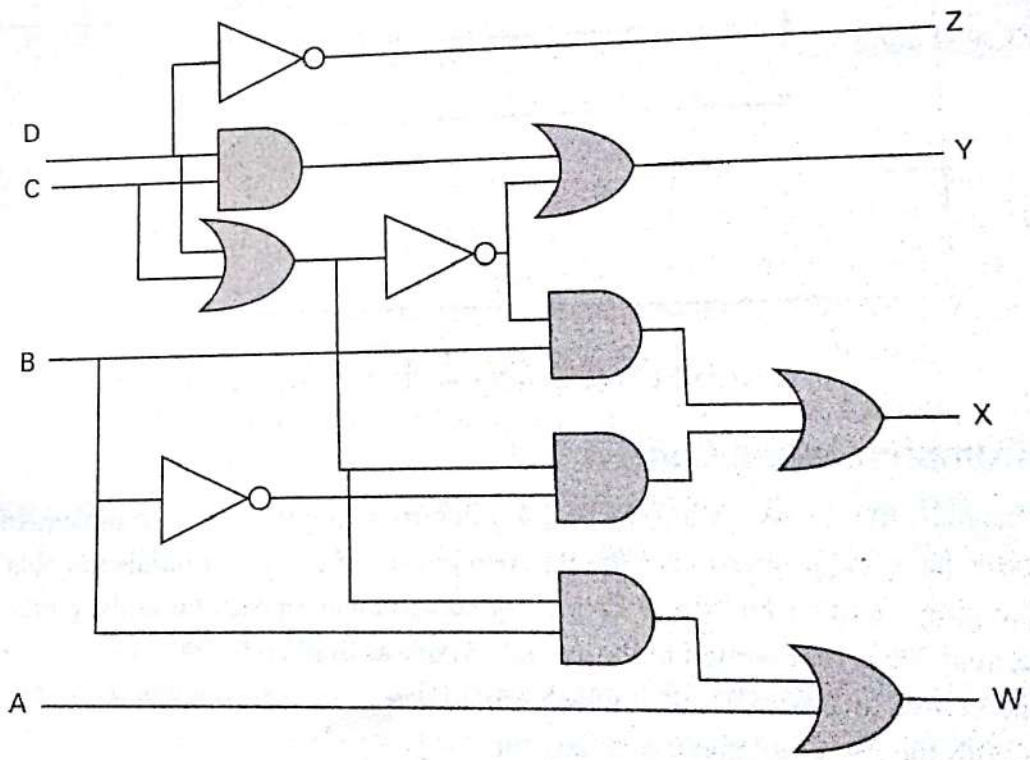


Figure 1.2 Logic diagram for BCD to Excess-3 code conversion.

Extended Binary Coded Decimal Interchange code (EBCDIC)

Extended Binary Coded Decimal Interchange code (EBCDIC) is an 8-bit character-coding scheme used primarily on IBM computers. A total of 256 (2^8) characters can be coded using this scheme. For example, the symbolic representation of letter A using Extended Binary Coded Decimal Interchange code is 11000001_2.

American standard code for information interchange (ASCII)

American standard code for information interchange (ASCII) is a 7-bit code, which means that only 128 characters i.e. 2^7 can be represented. However, manufactures have added an eight bit to this coding scheme, which can now provide for 256 characters. This 8-bit coding scheme is referred to as an 8-bit American standard code for information interchange. The symbolic representation of letter A using this scheme is 1000001_2.

This codes represent text in computers, communication equipment, and other devices that use text. It includes definitions for 128 characters: 33 are non-printing control characters(now mostly obsolete)that affect how text and space is processed;95 are printable characters, including the space, which is considered an invisible graphic ASCII reserves the first 32 codes (numbers 0–31 decimal) for control character and it is a very well-known fact that computers can manage internally only 0s (zeros) and 1s (ones).

For example, the ASCII code for the capital letter "A" is always represented by the order number 65, which is easily represent able using 0s and 1s in binary: 65 expressed as a binary number is 1000001. The standard ASCII table defines 128 character codes (from 0 to 127), of which, the first 32 are control codes (non-printable), and the remaining 96 character codes are represent able character

1.7 Error Detection Code

Binary information transmitted through some form of communication medium is subject to external noise that could change bits from 1 to 0, and vice-versa. An error detection code is a binary code that detects digital errors during transmission. The detected errors cannot be corrected but their presence is indicated. The usual procedure is to observe the frequency of errors. If errors occur infrequently at random, the particular erroneous information is transmitted again. If the error occurs too often, the system is checked for malfunction.

The most common error detection code used is the party bit. A party bit is an extra bit include with a binary message to make the total number of 1's either odd or even. A message of three bits and two possible parity bits is shown in table 3-7. The P(odd) bit is chosen in such a way as to make the sum of 1's even. In either case, the sum is taken over the message and the P bit. In any particular application, one or the other type of parity will be adopted. The even-parity scheme has the disadvantage of having a bit combination of all 0's, while in the odd parity there is always one bit (of the four bits that constitute the message and P) that is 1. Note that P(odd) is the complement of the P(even).

During transfer of information from one location to another, the parity bit is handled as follows. At the sending end, the message (in this case three bits) is applied to parity generator, where the required parity bit is generated. The message, including the parity bit, is transmitted to its destination. At the receiving end, all the oncoming bits (in this case, four) are applied to a parity checker that checks the proper parity adopted (odd or even). An error is detected if the checked parity does not conform to the adopted parity. The parity method detects the presence of one, three, or any odd number of errors. An even number of errors is not detected.

Message xyz	P(odd)	P)even)
000	1	0
001	0	1
010	0	1
011	1	0
100	0	1
101	1	0
110	1	0
111	0	1

Parity generator and checker networks are logic circuits constructed with exclusive-OR functions. This is because the exclusive-OR function of three or more variables is by definition an odd function. An odd function is a logic function whose value is binary 1 if, an only if, an odd number of variables are equal to 1. According to this definition, the P(even) function is the exclusive -OR of x, y, and z because it is equal to 1 when either one or all three of the variables are equal to 1. The P(odd) function is the complement of the P(even) function.

As an example, consider a 3-bit message to be transmitted with an odd parity bit. At the sending end, the odd-parity bit is generated by a parity generator circuit. As shown in Fig.1-3, this circuit consists of one exclusive-OR and one exclusive-NOR gate. Since P(even) is the exclusive-OR of x, y, z and P(odd) is the complement of P(even)), it is necessary to employ an exclusive-NOR gate for the needed complementation. The message and the odd-parity bit are transmitted to their destination where they are applied to a parity checker. An error has occurred during transmission of the parity of the four bits received is even, since the binary information transmitted was originally odd. The output of the parity checker would be 1 when error occurs, that is, when the number of 1's in the four inputs is even. Since the exclusive-OR function of the four inputs is an odd function, we again need to complement the output by using an exclusive-NOR gate.

It is worth noting that the parity generator can use the same circuit as the parity checker if the fourth input is permanently held at a logic-0 value. The advantage of this is that the same circuit can be used for both parity generation and parity checking.

It is evident from the example above that even-parity generators and checkers can be implemented with exclusive-OR functions. Odd-parity net-works need an exclusive-NOR at the output to complement the function.

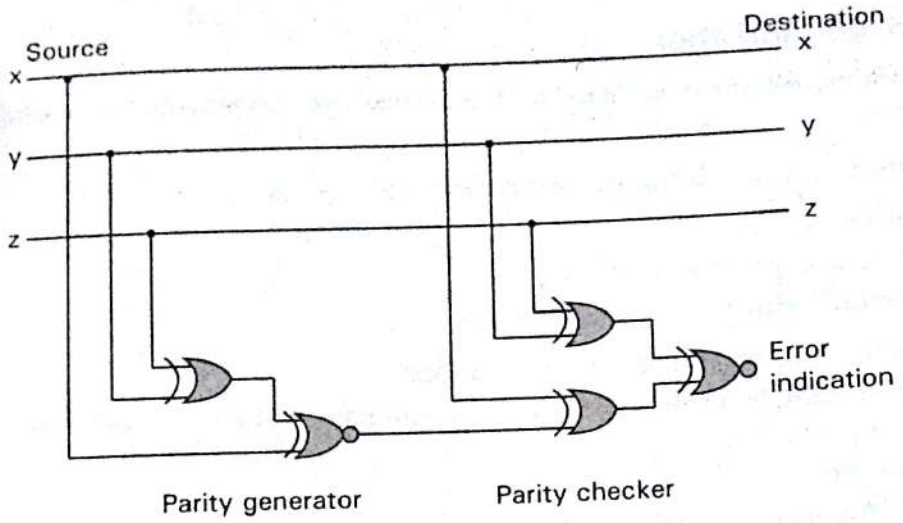


Fig. 1-3: *Error detection with odd parity bit*

1.8 Computer Architecture and Organization

Computer Architecture

Some Definition of Computer Architecture

"The design of integrated system which provides a useful tool to the programmer." (Baer)

"The study of the structure, behaviour, and design of computers." (Hayes)

"The design of the system specification at a general or subsystem level." (Abd-Alla)

"The art of designing a machine that will be a pleasure to work with." (Foster)

"The interface between the hardware and the lowest level software." (Hennessy and Patterson)

- **Keyword:**
  - Design
  - System
  - Structure
  - Art Tool Interface

Therefore, computer architecture refers to

- Attributes of a system *visible* to programmers
- Attributes that have a direct impact on the execution of programs

Attributes

- Instruction set
- Data representation
- I/O mechanisms
- Addressing techniques

Computer Organization

Organization refers to operational units and their interconnections that realize the architectural specifications.

Attributes: hardware details transparent to programmers

- Control signals
- Computer/peripheral interface
- Memory technology

Architecture & organization comparison

- *Architecture* is attributes visible to programmers whereas *Organization* is how features are implemented
- **Example:**
  - Architecture: multiply instruction?
  - Organization: hardware multiply unit or done by repeated addition? (how is it implemented?)
- **Example:**
  - IBM/370 architecture
  - different models (organizations)
- **Family Concept**
  - All Intel x86 family share the same basic architecture
  - The IBM System/370 family share the same basic architecture
  - This gives code compatibility (at least backwards)
- Organization differs between different versions

Structure & Function

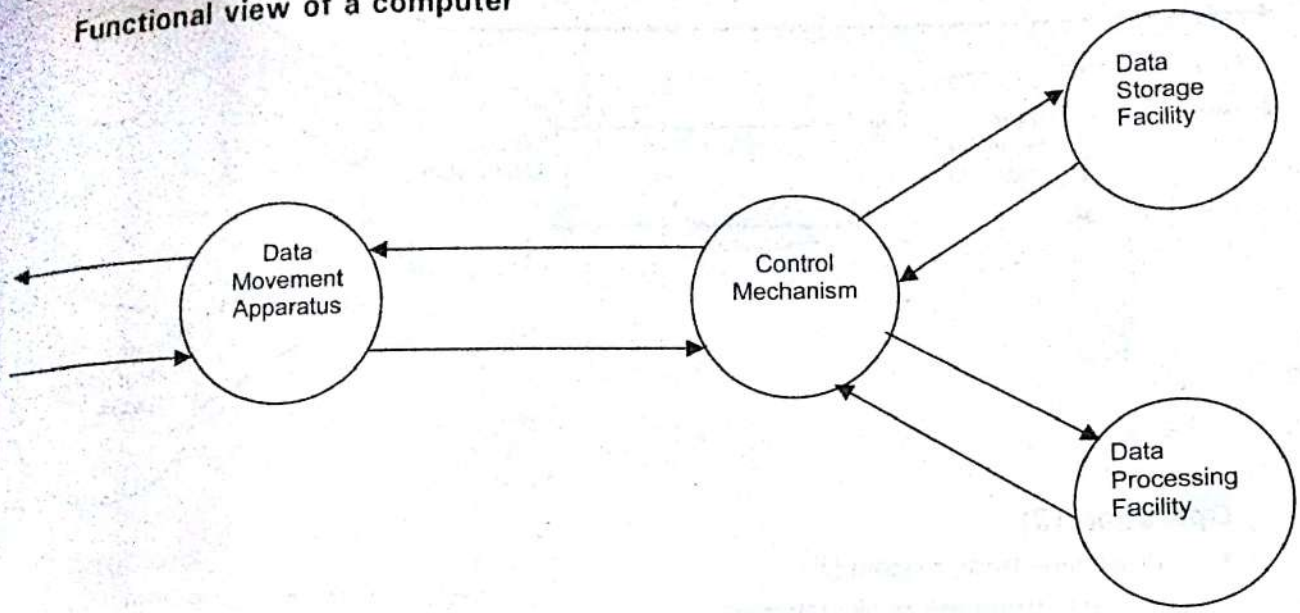
- **Computer** is a complex system that's why its description and design is always challenging.
- **Hierarchic system:**
  - A set of interrelated subsystems, each subsystem hierarchic in structure until some lowest level of elementary subsystems is reached
- At each level of the system, the designer is concerned with structure and function.
- Structure is the way in which components relate to each other
- Function is the operation of individual components as part of the structure

Function

- **Basic computer functions:**
  - Data processing
  - Data storage
  - Data movement
  - Control

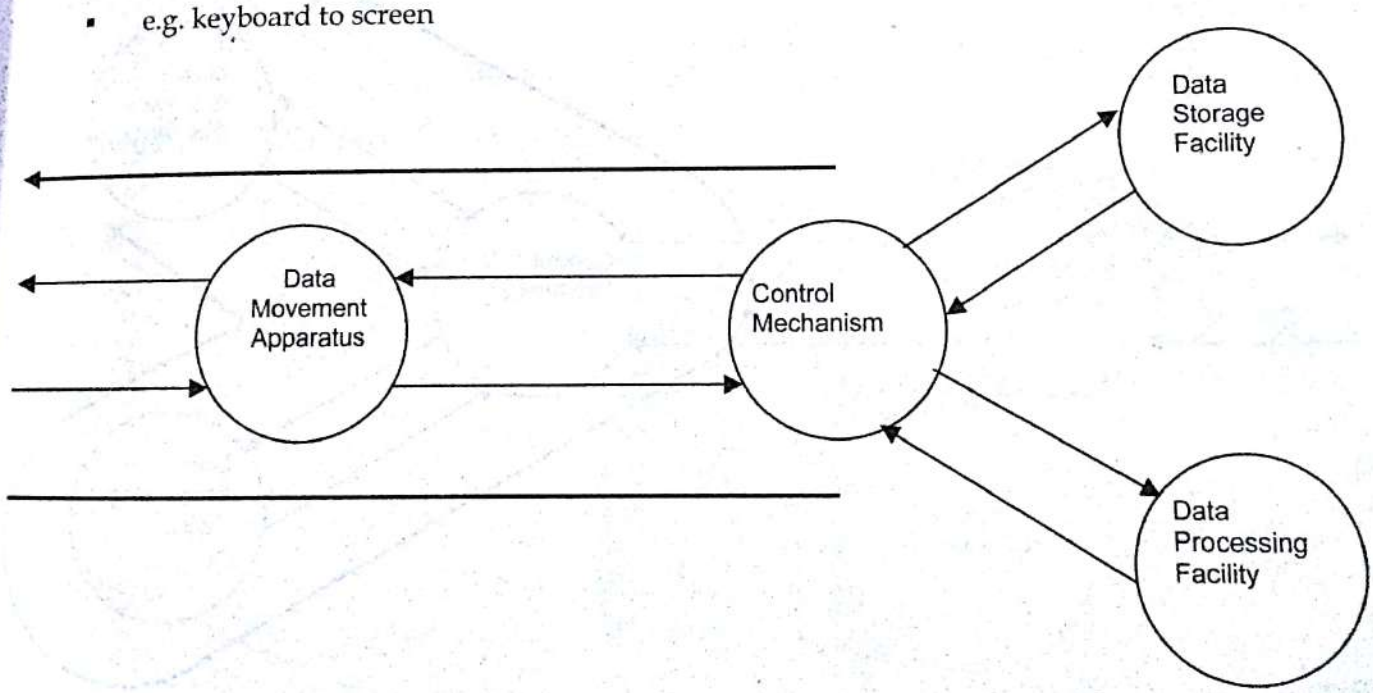
Functional view

Functional view of a computer



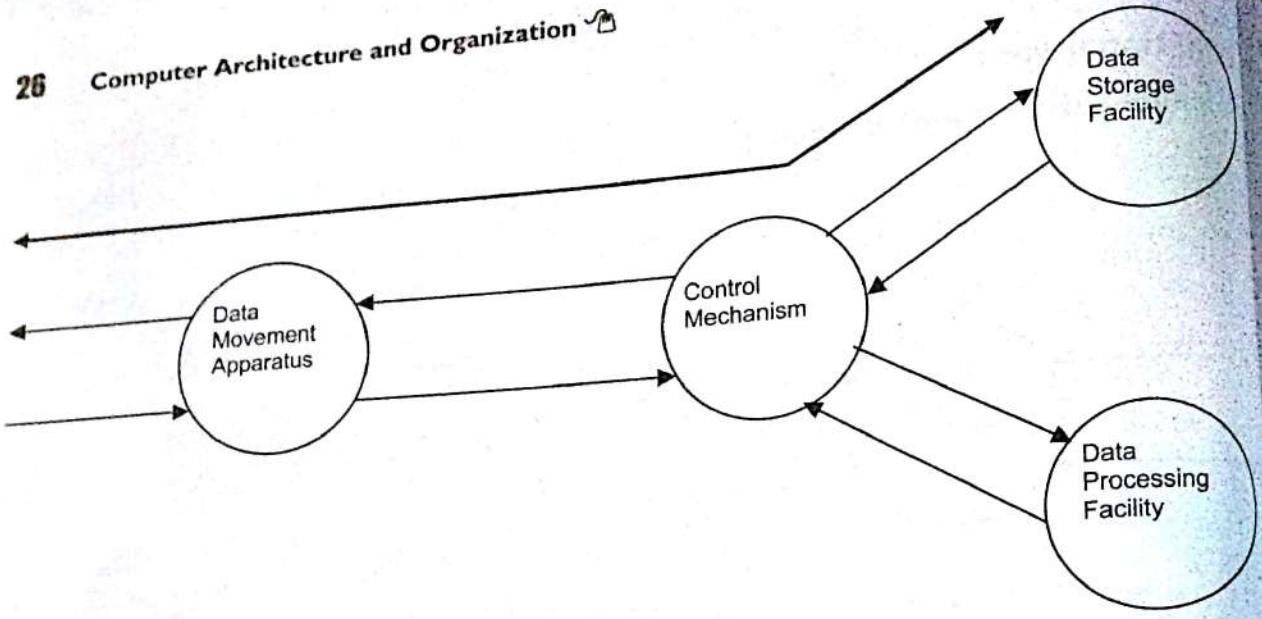
Data movement

- e.g. keyboard to screen



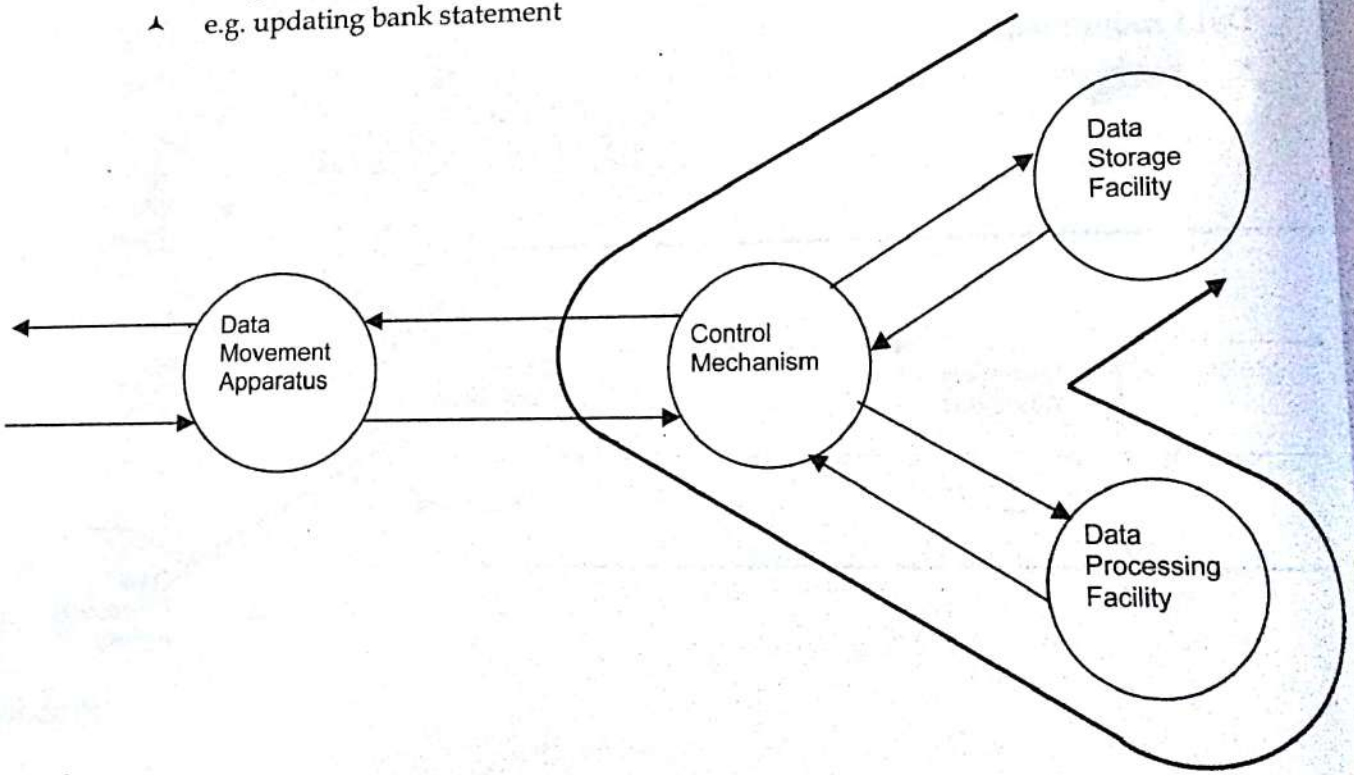
Storage

- e.g. Internet download to disk



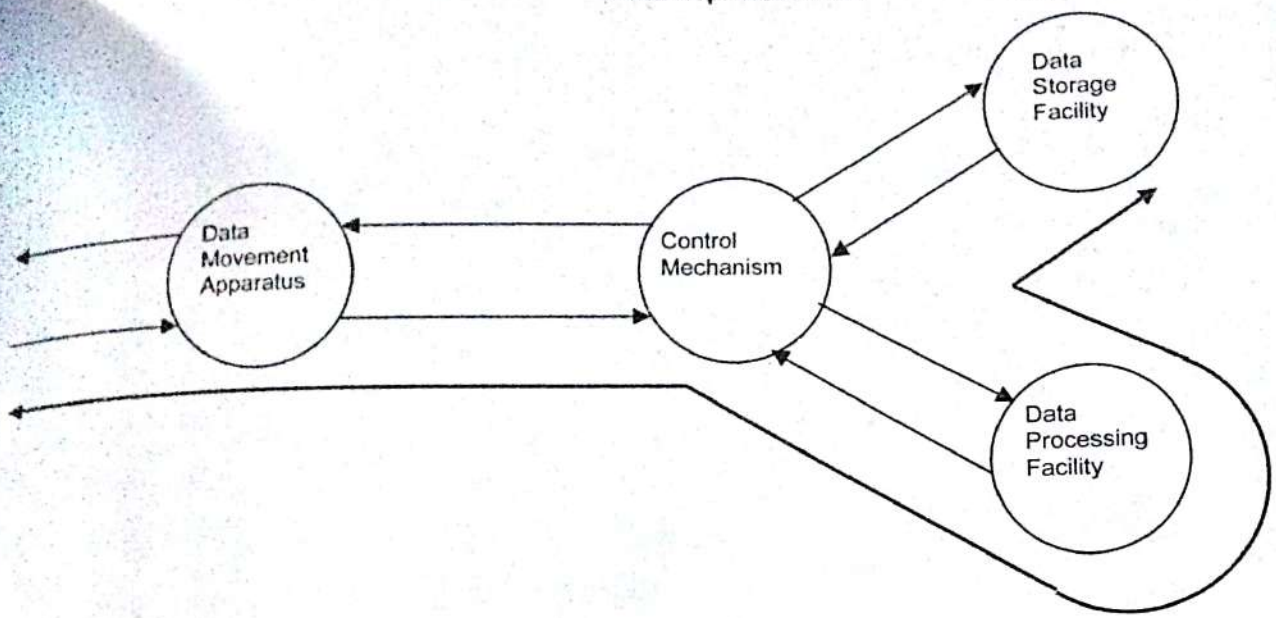
Operation (3)

- Processing from/to storage
  - e.g. updating bank statement



Operation (4)

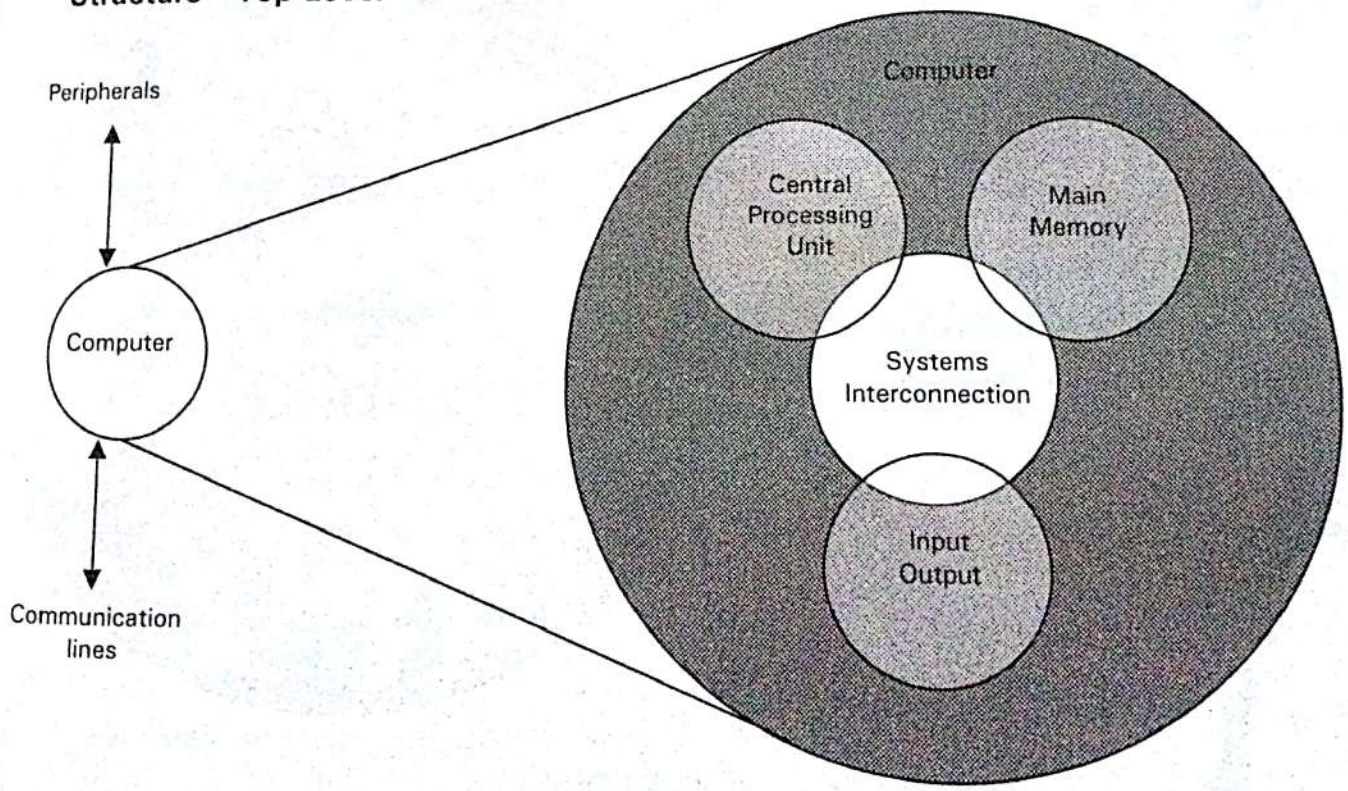
- Processing from storage to I/O
  - e.g. printing a bank statement



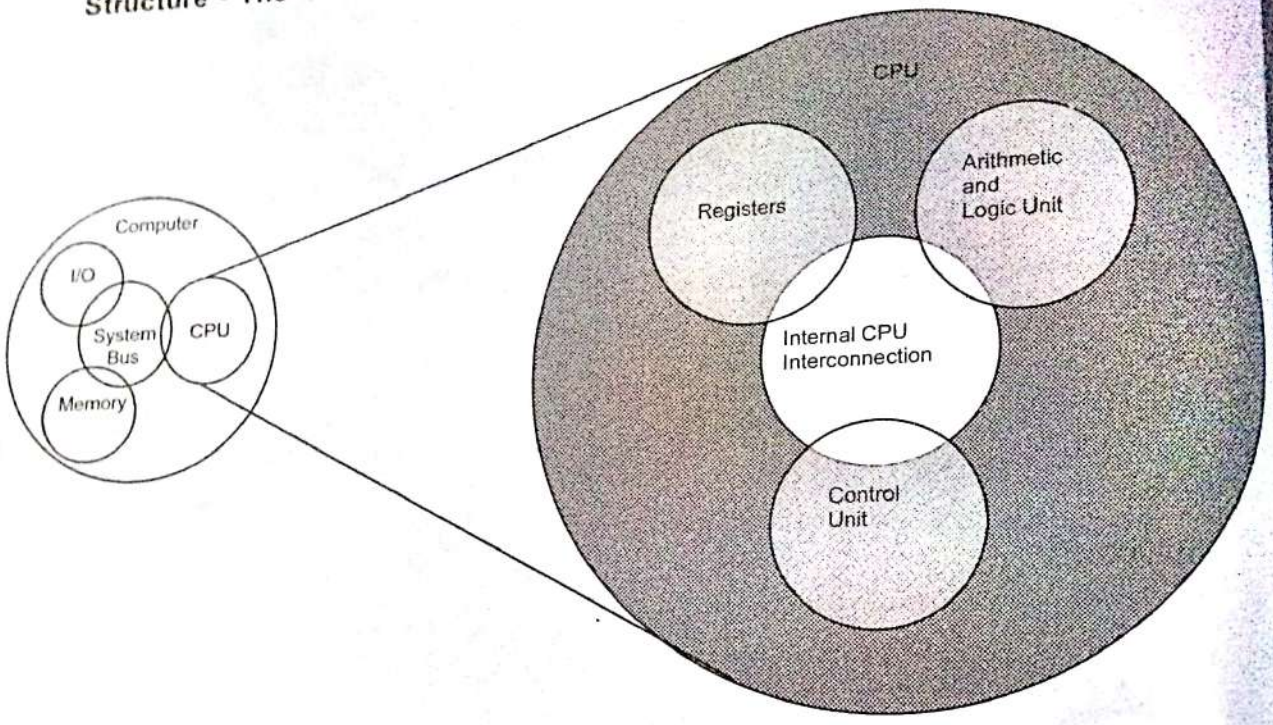
Structure

- Four main structural components
  - CPU
  - Main Memory
  - I/O Devices
  - System Interconnection

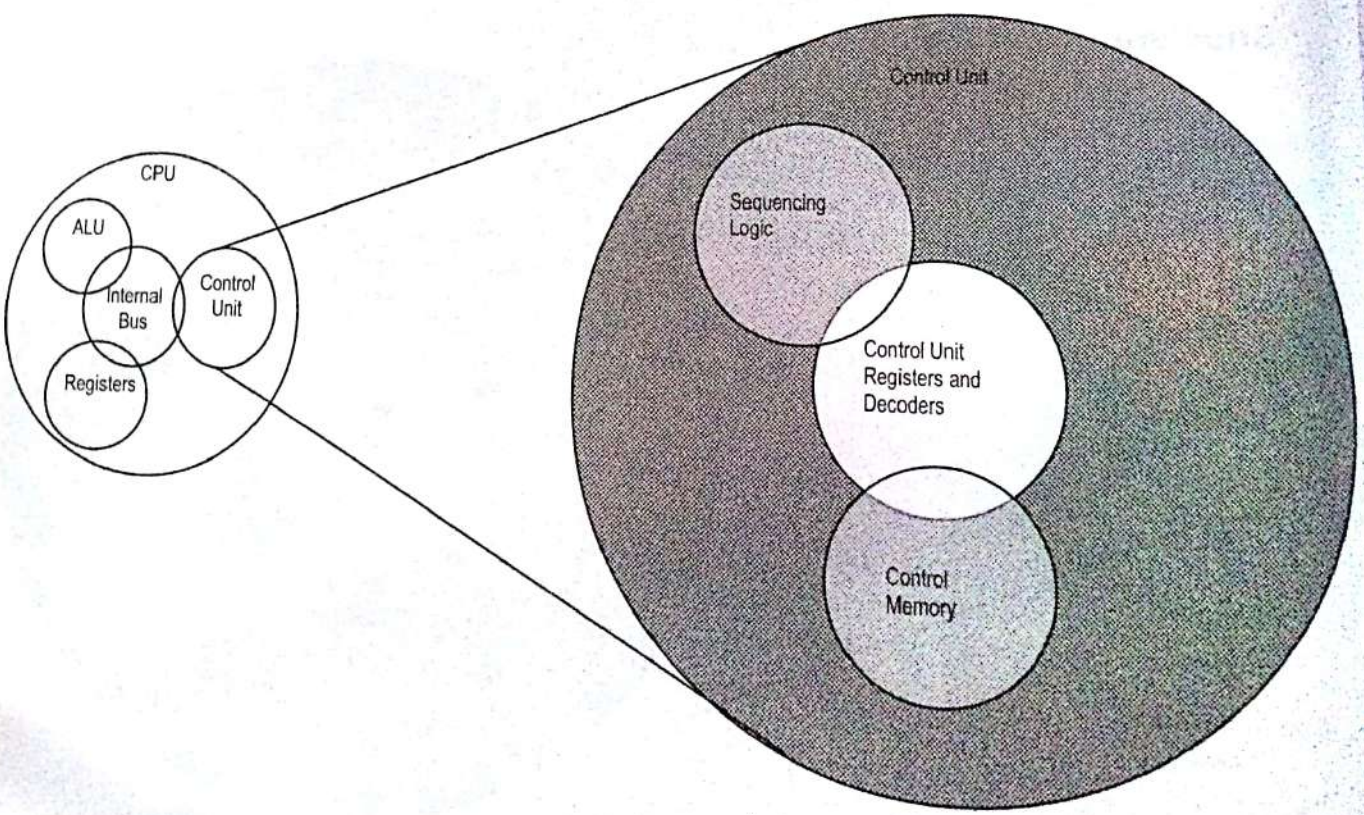
Structure - Top Level



Structure - The CPU



Structure - The Control Unit



Chapter 2

Register Transfer and Microoperations

Objective

This chapter introduces a register transfer language and shows how it is used to express microoperations in symbolic form. Symbols are defined for arithmetic, logic, and shift microoperations. A composite arithmetic , logic shift unit is developed to show the hardware design of the most common microoperations.

OUTLINE

2.1 Register Transfer Language
2.2 Register Transfer
2.3 Bus and Memory Transfers
2.4 Arithmetic Microoperations
2.5 Logic Microoperations
2.6 Shift Microoperations
2.7 Arithmetic Logic shift Unit

2.1 INTRODUCTION

Computers are classified according to **functionality, physical size** and **purpose**. Functionality, Computers could be **analog, digital or hybrid**. Digital computers process data that is in discrete form whereas analog computers process data that is continuous in nature. Hybrid computers on the other hand can process data that is both discrete and continuous.

In digital computers, the user input is first converted and transmitted as electrical pulses that can be represented by two unique states ON and OFF. The ON state may be represented by a "1" and the off state by a "0".The sequence of ON'S and OFF'S forms the electrical signals that the computer can understand. A digital signal rises suddenly to a peak voltage of +1 for some time then suddenly drops -1 level on the other hand an analog signal rises to +1 and then drops to -1 in a continuous version.

2.2 REGISTER TRANSFER LANGUAGE

Digital systems are composed of modules that are constructed from digital components, such as registers, decoders, arithmetic elements, and control logic. The modules are interconnected with common data and control paths to form a digital computer system. The operations 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. Examples are shift, count, clear, and load. Some of the digital components from before are registers that implement microoperations. The internal hardware organization of a digital computer is best defined by specifying:

- The set of registers it contains and their functions
- The sequence of microoperations performed on the binary information stored
- The control that initiates the sequence of microoperations

It uses symbols, rather than words, to specify the sequence of microoperations. The symbolic notation used is called a *register transfer language*. A programming language is a procedure for writing symbols to specify a given computational process. RTL defines symbols for various types of microoperations and describe associated hardware that can implement the microoperations

2.3 REGISTER TRANSFER

Designate computer registers by capital letters to denote its function. The register that holds an address for the memory unit is called MAR. The program counter register is called PC

IR is the instruction register and R_1 is a processor register. The individual flip-flops in an n-bit register are numbered in sequence from 0 to n-1 Refer to Figure 4.1 for the different representations of a register

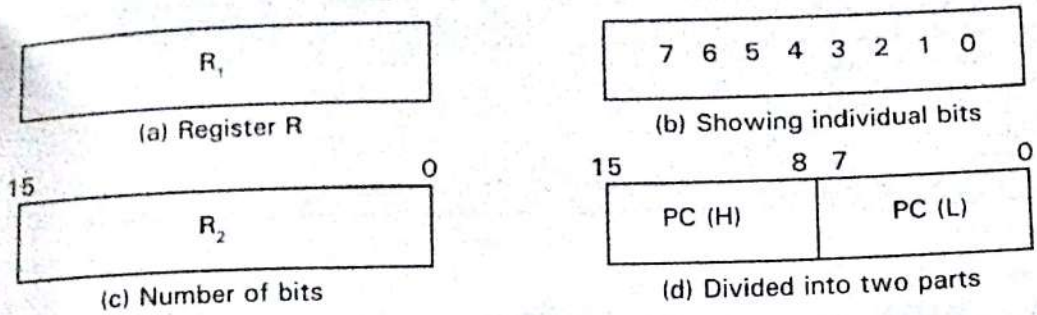


Figure 2.1: *Different representations of a register*

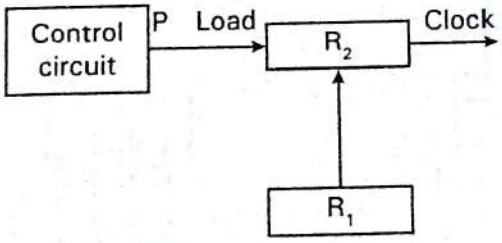
- Designate information transfer from one register to another by $R_2 \leftarrow R_1$
- This statement implies that the hardware is available
  - The outputs of the source must have a path to the inputs of thedestination
  - The destination register has a parallel load capability
- If the transfer is to occur only under a predetermined control condition, designate it by

  or, If(P = 1) then $(R_2 \rightarrow R_1)$

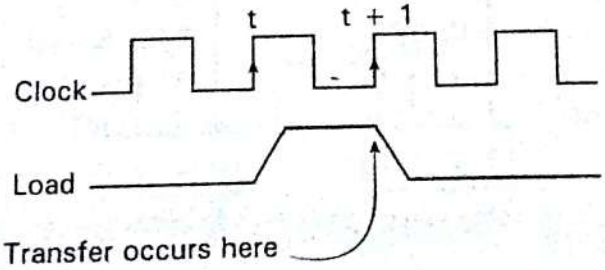
  $P : R_2 \leftarrow R_1$

where P is a control function that can be either 0 or 1

- Every statement written in register transfer notation implies the presence of the required hardware construction



(a) Block diagram



(b) Timing diagram

Figure 2.2: *Transfer from R_1 to R_2 when P = 1*

- It is assumed that all transfers occur during a clock edge transition
- All microoperations written on a single line are to be executed at the same time

  $T: R_2 \leftarrow R_1, R_1 \leftarrow R_2$

Symbol	Description	Examples
Letters and numerals	Denotes a register	MAR, R2
Parentheses ( )	Denotes a part of register	R2(0 – 7), R2(L)
Arrow ←	Denotes transfer of information	R2 ← R1
Comma ,	Separates two microoperations	R2 ← R1, R1 ← R2

Table 2.1 : *Basic Symbols for Register Transfer*

2.4 BUS AND MEMORY TRANSFERS

Rather than connecting wires between all registers, a common bus is used. A bus structure consists of a set of common lines, one for each bit of a register where control signals determine which register is selected by the bus during each transfer. Multiplexers can be used to construct a common bus and it selects the source register whose binary information is then placed on the bus. The select lines are connected to the selection inputs of the multiplexers and choose the bits of one register

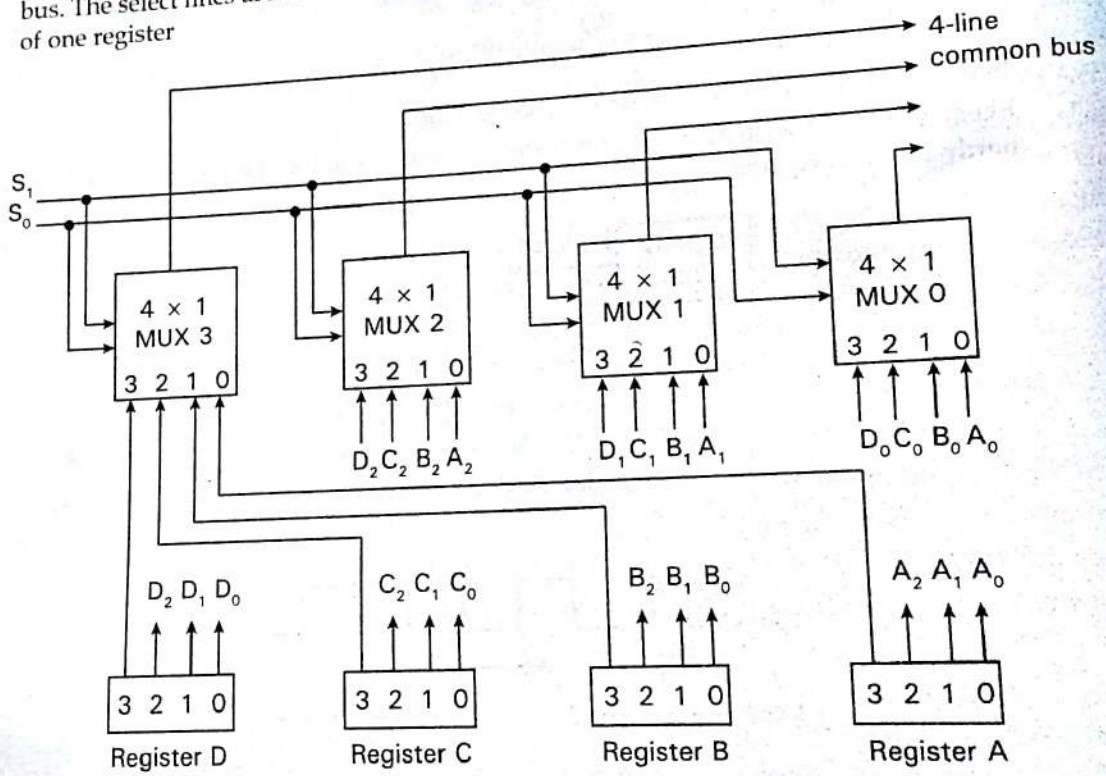


Figure 2.3: *Bus system for four registers*

S_1	SS=0	Register Selected
0	0	A
0	1	B
1	0	C
1	1	D

Table 2.2 : *Function table for bus of Fig. 2.3*

In general, a bys system will multiplex k registers of n bits each to produce an n- line common bus. This requires n multiplexers - one for each bit. The size of each multiplexer must be $k \times 1$ and the number of select lines required is $\log k$. To transfer information from the bus to a register, the bus lines are connected to the inputs of all destination registers and the corresponding load control line must be activated rather than listing each step as BUS ← C, R_1 ← BUS, use R_1 ← C, since the bus is implied. Instead of using multiplexers, *three-state gates* can be used to construct the bus system. A three-state gate is a digital circuit that exhibits three states where two of the states are signals equivalent to logic 1 and 0 and the third state is a *high-impedance* state - this behaves like an open circuit, which means the output is disconnected and does not have a logic significance

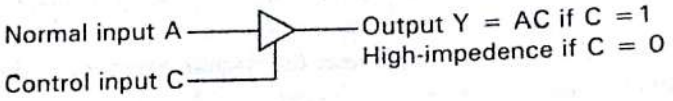


Figure 2.4: *Graphical symbols for three-state buffer*

The three-state buffer gate has a normal input and a control input which determines the output state: With control 1, the output equals the normal input; with control 0, the gate goes to a high-impedance state; this enables a large number of three-state gate outputs to be connected with wires to form a common bus line without endangering loading effects

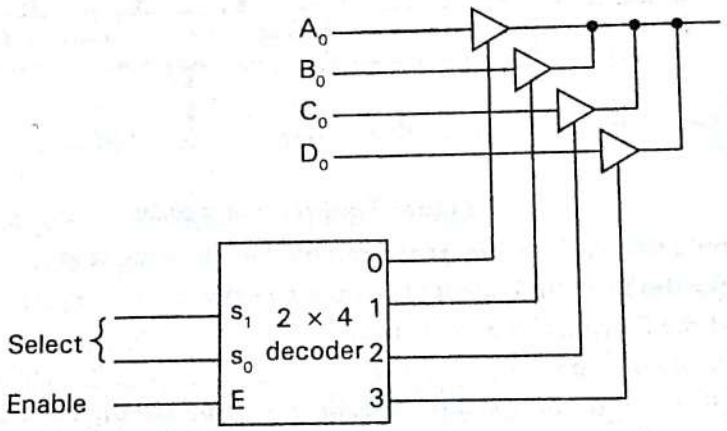


Figure 2.5: *Bus line with three state-buffers*

2.5 ARITHMETIC MICROOPERATIONS

- There are four categories of the most common microoperations:
  - **Register transfer:** transfer binary information from one register to another
  - **Arithmetic:** perform arithmetic operations on numeric data stored in registers
  - **Logic:** perform bit manipulation operations on non-numeric data stored in registers
  - **Shift:** perform shift operations on data stored in registers

The basic arithmetic microoperations are addition, subtraction, increment, decrement, and shift. An example of addition: $R_3 \leftarrow R_1 + R_2$ and subtraction is most often implemented through complementation and addition. Example of subtraction: $R_3 \leftarrow R_1 + \overline{R_2} + 1$

Symbolic designation	Description
$R3 \leftarrow R1 + R2$	Contents of R1 plus R2 transferred to R3
$R3 \leftarrow R1 - R2$	Contents of R1 minus R2 transferred to R3
$R2 \leftarrow \overline{R2}$	Complement of the contents of R2 (1's complement)
$R2 \leftarrow \overline{R2} + 1$	2's complement the content of R2 (negate)
$R3 \leftarrow R1 + \overline{R2} + 1$	R1 plus the 2's complement of R2 (subtraction)
$R1 \leftarrow R1 + 1$	Increment the contents of R1 by one
$R1 \leftarrow R1 - 1$	Decrement the contents of R1 by one

Table 2.3 : *Arithmetic Microoperations*

Multiply and divide are not included as microoperations. A microoperation is one that can be executed by one clock pulse. Multiply (divide) is implemented by a sequence of add and shift microoperations (subtract and shift). To implement the add microoperation with hardware, we need the registers that hold the data and the digital component that performs the addition. A full-adder adds two bits and a previous carry.

A *binary adder* is a digital circuit that generates the arithmetic sum of two binary numbers of any length where a binary added is constructed with full-adder circuits connected in cascade. An *n*-bit binary adder requires *n* full-adders.

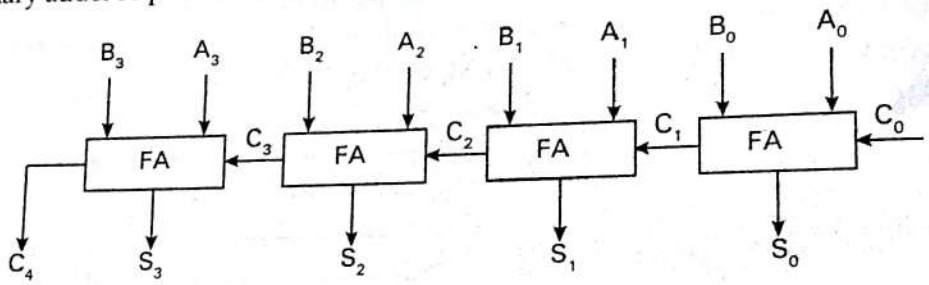


Figure 2.6: *4-bit binary adder*

- The subtraction A-B can be carried out by the following steps
  - Take the 1's complement of B (invert each bit)
  - Get the 2's complement by adding 1
  - Add the result to A
- The addition and subtraction operations can be combined into one common circuit by including an XOR gate with each full-adder

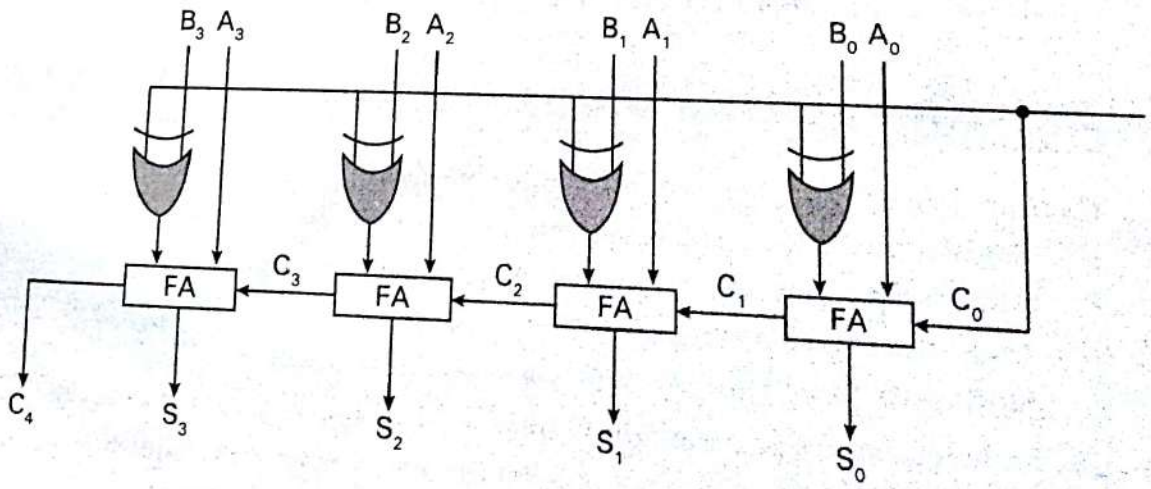


Figure 2.7: *4-bit adder-subtractor*

The increment microoperation adds one to a number in a register. This can be implemented by using a binary counter - every time the count enable is active, the count is incremented by one. If the increment is to be performed independent of a particular register, then use half-adders connected in cascade. An *n*-bit binary incrementer requires *n* half-adders.

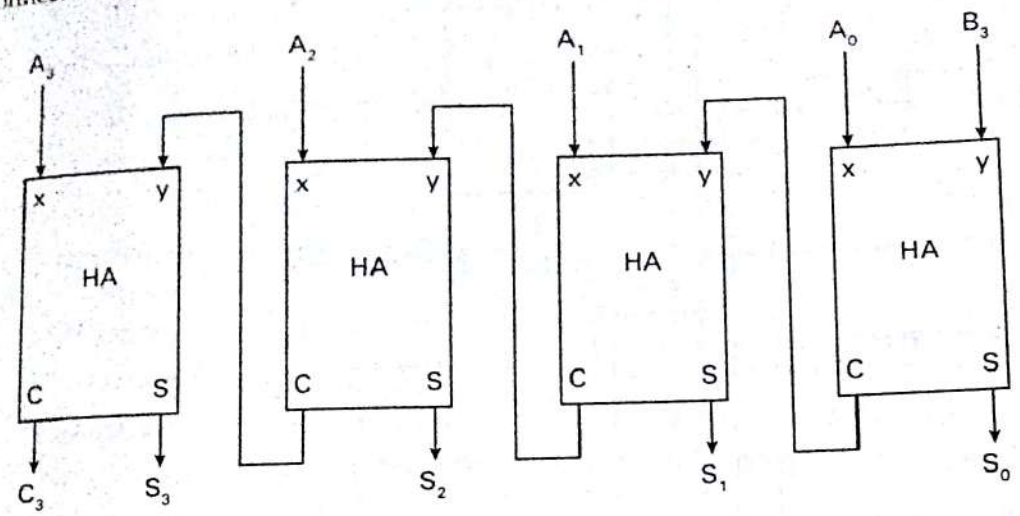


Figure 2.8: *4-bit binary incrementer*

Arithmetic Circuit

Each of the arithmetic microoperations can be implemented in one composite arithmetic circuit. The basic component is the parallel adder and multiplexers are used to choose between the different operations. The output of the binary adder is calculated from the following sum:

$$D = A + Y + C_{in}$$

- **Addition:** when $S_1S_0 = 00$, Y = B. if $C_{in} = 0$, the output d = A =+ B. If $C_{in} = 1$, output D = A + B + 1
- **Subtraction:** when $S_1SS_0 = 01$. Y = B. If $C_{in} = 1$, then $D = A + \overline{B} = 1 = AA - B$. When $C_{in} = 0$ then $D = A + \overline{B} == A - B - 1$.
- **Increment:** When $S_1S_0 = 10$, Y = 0 (transfer), the output $d = A + 0 + C_{in}$ (increment). This gives D = A (transfer) when $C_{in} = 0$ and D = A + 1 (increment) when $C_{in} = 1$.
- **Decrement:** When $S_1S_0 = 11$, all is insulted into Y inputs hence Dd= A – L when $C_{in} = 0$. This is because a number with all 1's is equal to the 2's compliment of 1 (the 2's compliment of binary 0001 is 1111). Adding a number a to 2'ss compliment of 1 produces F = AA + 2's complement off 1 = A – 1. when $C_{in} = 1$, then D = A – 1 + 1 = A which causes a direct transfer from input A to output D.

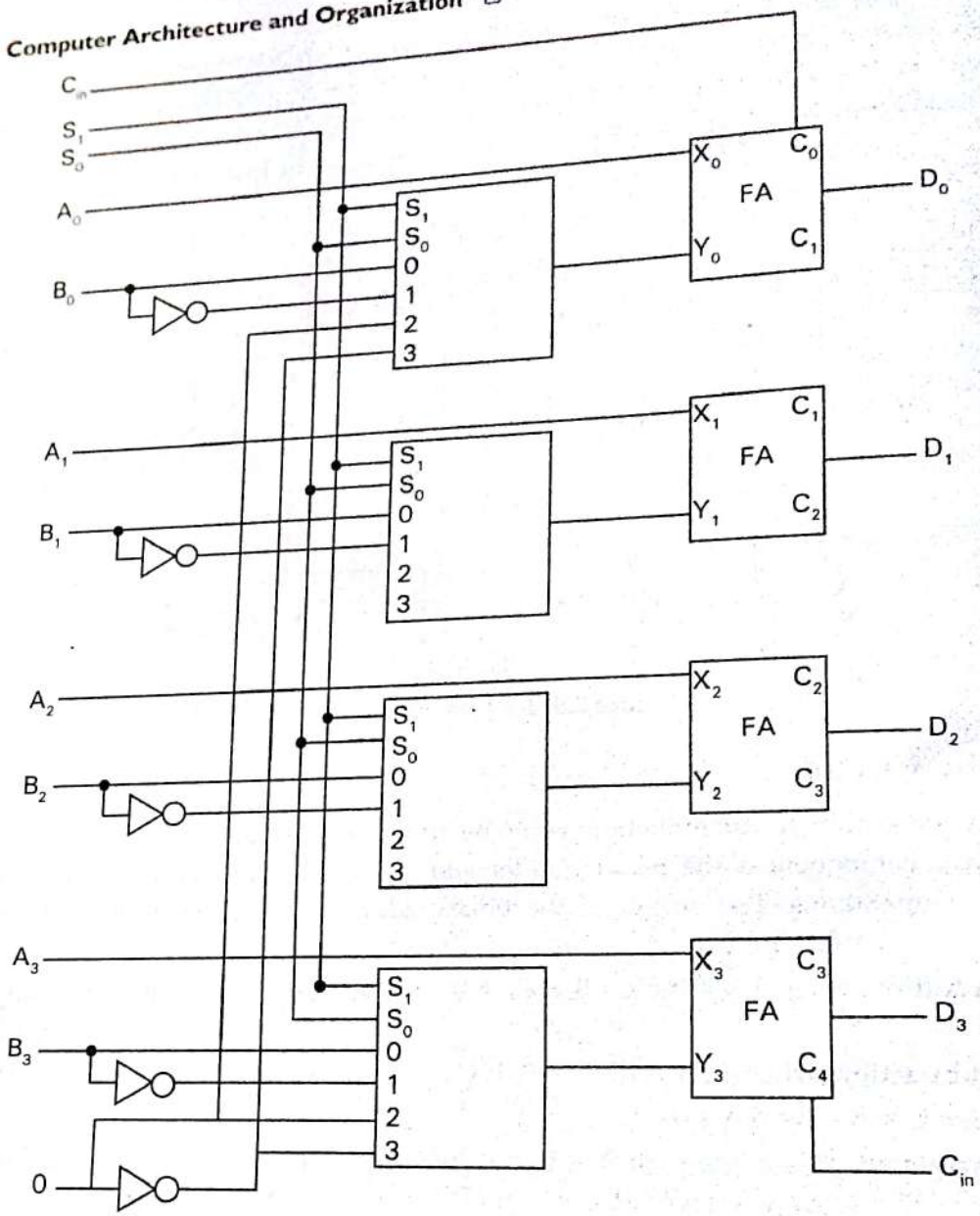


Figure 2.9: *4-bit arithmetic circuit*

Select			Input Y	Output $D = A + Y + C_{in}$	Microoperation
S_1	S_o	C_{in}			
0	0	0	B	$D = A + B$	Add
0	0	1	B	$D = A + B + 1$	Add with carry
0	1	0	B	$D = A + \overline{B}$	Subtract with borrrow
0	1	1	B	$D = A + \overline{B} + 1$	Subtract
1	0	0	0	$D = A$	Transfer A
1	0	1	0	$D = A + 1$	Increment A
1	1	0	1	$D = A - 1$	Decrement A
1	1	1	1	$D = A$	Transfer A

Table 2.4: Arithmetic circuit function table

2.6 LOGIC MICROOPERATIONS

Logic operations specify binary operations for strings of bits stored in registers and treat each bit separately. Example: the XOR of R1 and R2 is symbolized by

$$P: R_1 \leftarrow R_1 \oplus R_2$$

Example: $R_1 = 1010$ and $R_2 = 1100$

1010	Content of R_1
1100	Content of R_2
0110	Content of R_1 after P = 1

- Symbols used for logical microoperations:
  - **OR:** $\vee$
  - **AND:** $\wedge$
  - **XOR:** $\oplus$

The + sign has two different meanings: logical OR and summation, when + is in a microoperation, then summation and when + is in a control function, then OR. Example:

$$P + Q: R_1 \leftarrow R_2 + R_3,\ R_4 \leftarrow R_5 \vee R_6$$

There are 16 different logic operations that can be performed with two binary variables

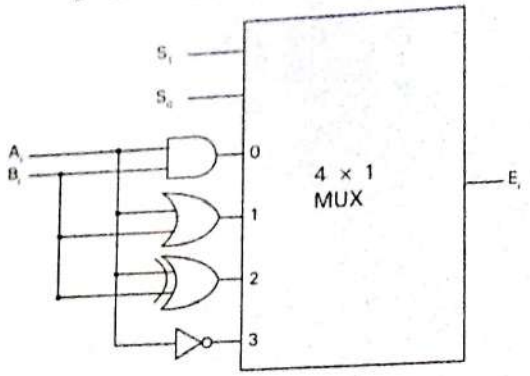
x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Table 2-5: *Truth tables for 16 functions of two variables*

Boolean function	Microoperation	Name
$F_0 = 0$	$F \leftarrow 0$	Clear
$F1 = xy$	$F \leftarrow A \wedge B$	AND
$F_2 = xy'$	$F \leftarrow A \wedge \overline{B}$	
$F_3 = x$	$F \leftarrow A$	Transfer A
$F_4 = x'y$	$F \leftarrow \overline{A} \wedge B$	
$F_5 = y$	$F \leftarrow B$	Transfer B
$F_6 = x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
$F_7 = x + y$	$F \leftarrow A \vee B$	OR
$F_8 = (x + y)'$	$F \leftarrow \overline{A \vee B}$	NOR
$F_9 = (x \oplus y)'$	$F \leftarrow \overline{A \oplus B}$	Exclusive-NOR
$F_{10} = y'$	$F \leftarrow \overline{B}$	Complement B
$F_{11} = x + y'$	$F \leftarrow A \vee \overline{B}$	
$F_{12} = x'$	$F \leftarrow \overline{A}$	Complement A
$F_{13} = x' + y$	$F \leftarrow \overline{A} \vee B$	
$F_{14} = (xy)'$	$F \leftarrow \overline{A \wedge B}$	NAND
$F_{15} = 1$	$F \leftarrow$ all 1's	Set to all 1's

Table 2-6: *Sixteen logic microoperations*

The hardware implementation of logic microoperations requires that logic gates be inserted for each bit or pair of bits in the registers. All 16 microoperations can be derived from using four logic gates



(a) Logic diagram

S_1	S_0	Output	Operation
0	0	$E = A \wedge B$	AND
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	$E = \overline{A}$	Complement

(b) Function table

Figure: 2-10: *One stage of logic circuit*

- Logic microoperations can be used to change bit values, delete a group of bits, or insert new bit values into a register
- The *selective-set* operation sets to 1 the bits in A where there are corresponding 1's in B
  - 1010 A before
  - 1100 B (logic operand) 1110 A after
  - $A \leftarrow A \vee B$
- The *selective-complement* operation complements bits in A where there are corresponding 1's in B
  - 1010 A before
  - 1100 B (logic operand) 0110 A after
  - $A \leftarrow A \oplus B$
- The *selective-clear* operation clears to 0 the bits in A only where there are corresponding 1's in B
  - 1010 A before
  - 1100 B (logic operand) 0010 A after
  - $A \leftarrow A \wedge \overline{B}$
- The *mask* operation is similar to the selective-clear operation, except that the bits of A are cleared only where there are corresponding 0's in B
  - 1010 A before
  - 1100 B (logic operand) 1000 A after
  - $A \leftarrow A \wedge B$
- The *insert* operation inserts a new value into a group of bits
- This is done by first masking the bits to be replaced and then Oring them with the bits to be inserted

0110 1010	A before
0000 1111	B (mask)
0000 1010	A after masking
0000 1010	A before
1001 0000	B (insert)
1001 1010	A after insertion

- The *clear* operation compares the bits in A and B and produces an all 0's result if the two number are equal
  - 1010 A
  - 1010 B
  - 0000 $A \leftarrow A \oplus B$

2.7 SHIFT MICROOPERATIONS

Shift microoperations are used for serial transfer of data. They are also used in conjunction with arithmetic, logic, and other data- processing operations and there are three types of shifts: logical, circular, and arithmetic. A *logical shift* is one that transfers 0 through the serial input. The symbols *shl* and *shr* are for logical shift-left and shift-right by one position $R_1 \leftarrow$ shl R_1. The *circular shift* (aka rotate) circulates the bits of the register around the two ends without loss of information. The symbols *cil* and *cir* are for circular shift left and right.

Symbolic designation	Description
$R \leftarrow$ shl R	Shift-left register R
$R \leftarrow$ shr R	Shift-right register R
$R \leftarrow$ cil R	Circular shift-left register R
$R \leftarrow$ cir R	Circular shift-right register R
$R \leftarrow$ ashl R	Arithmetic shift-left R
$R \leftarrow$ ashr R	Arithmetic shift-right R

Table2-7: *Shift microoperations*

The *arithmetic shift* shifts a signed binary number to the left or right, to the left is multiplying by 2, to the right is dividing by 2 and arithmetic shifts must leave the sign bit unchanged. A sign reversal occurs if the bit in R_{n-1} changes in value after theshift, this happens if the multiplication causes an overflow. An overflow flip-flop V_s can be used to detect the overflow $V_s = R_{n-1} \oplus R_{n-2}$

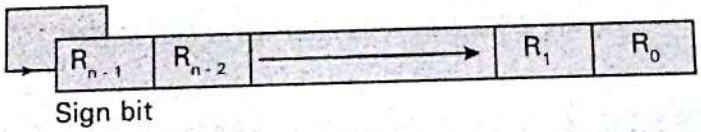
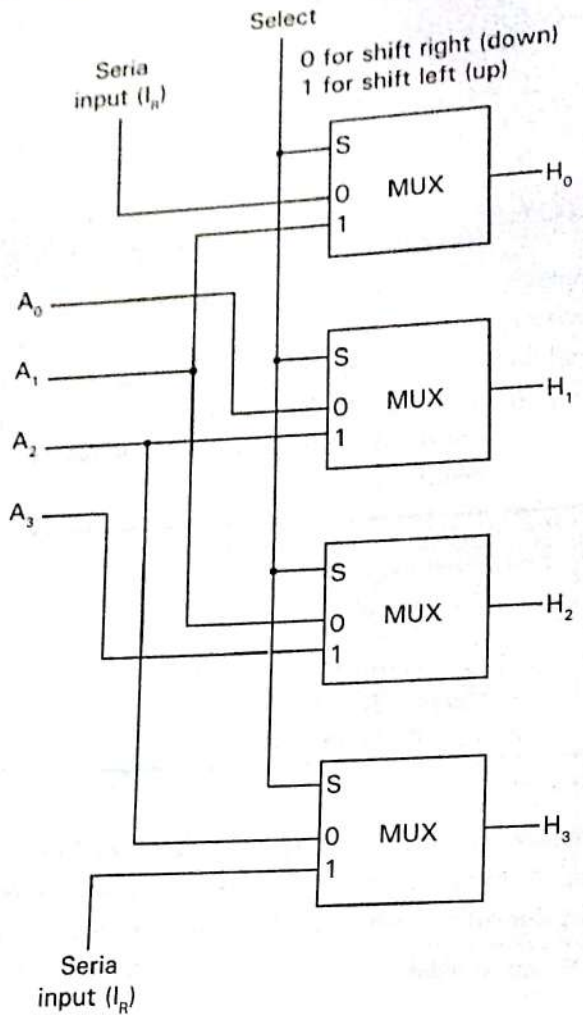


Figure 2-11: *Arithmetic shift right*

A bi-directional shift unit with parallel load could be used to implement this kind of shifting operation. Here, two clock pulses are necessary with this configuration: one to load the value and another to shift. In a processor unit with many registers it is more efficient to implement the shift operation with a combinational circuit. The content of a register to be shifted is first placed onto a common bus and the output is connected to the combinational shifter, the shifted number is then loaded back into the register. This can be constructed with multiplexers.

Function Table				
Select	Output			
S	H_0	H_1	H_2	H_3
0	I_R	A_0	A_1	A_2
1	A_1	A_2	A_3	I_L



Function Table

Select	Output			
S	H_0	H_1	H_2	H_3
0	I_R	A_0	A_1	A_2
1	A_1	A_2	A_3	I_L

Figure 2-12: *4-bit combinational circuit shifter*

2.8 Arithmetic Logic Shift Unit

The arithmetic logic unit (ALU) is a common operational unit connected to a number of storage registers. To perform a microoperation, the contents of specified registers are placed in the inputs of the ALU. The ALU performs an operation and the result is then transferred to a destination register. The ALU is a combinational circuit so that the entire register transfer operation from the source registers through the ALU and into the destination register can be performed during one clock pulse period.

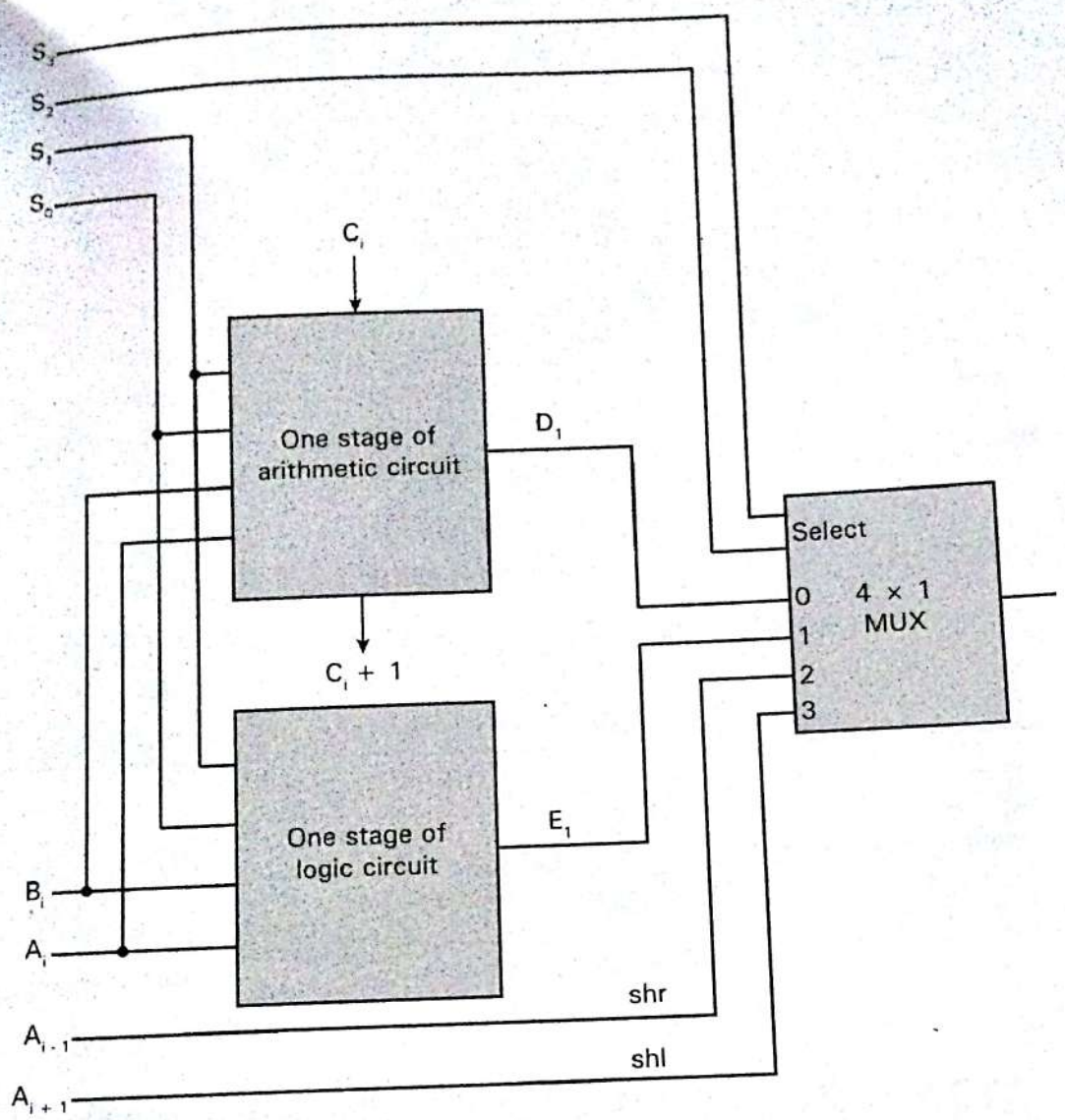


Figure 2-13; One stage of arithmetic logic-shift unit

The arithmetic, logic and shift circuits introduced in previous sections can be combined into one ALU with common selection variables. One stage of an arithmetic logic shift unit is shown in Fig. 2.3. The subscript designates a typical stage. Inputs A=i and B_i are applied to both the arithmetic and logic units. A particular microoperration is selected with inputs S_1 and S_0. A 4 × 2 multiplier at the output chooses between an arithmetic output E_i and a logic output in H_i. The data in multiplier are selected with inputs SS_3 and S_2. The other two data inputs to the multiplier receive inputs A_{i-1} for the shift right operation and A_{i+1} for the shift left operation.

Operation Select					Operation	Function
S_3	S_2	S_1	S_0	C_{in}		
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	$F = A + \overline{B}$	Subtract with borrow
0	0	1	0	1	$F = A + \overline{B} + 1$	Subtraction
0	0	1	1	0	$F = A - 1$	Decrement A
0	0	1	1	1	$F = A$	Transfer A
0	1	0	0	×	$F = A \wedge B$	AND
0	1	0	1	×	$F = A \vee B$	OR
0	1	1	0	×	$F = A \oplus B$	XOR
0	1	1	1	×	$F = \overline{A}$	Complement A
1	0	×	×	×	F = shr r	Shift right A into F
1	1	×	×	×	F = shl A	Shift left A into F

Table 2-8: *Function table for arithmetic logic shift unit*

Chapter 3

Basic Computer Organization and Design

Objective

This chapter presents the organization and design of a basic digital computer. Although the computer is simple compared to commercial computers, it is nevertheless encompasses enough functional capabilities to demonstrate the power of a stored program general purpose device. By going through the detailed steps of the design presented in this chapter, the student will be able to understand the inner working of digital computer.

OUTLINE

3.1 Instruction Codes
3.2 Computer Registers
3.3 Computer Instructions
3.4 Timing and Control
3.5 Instruction Cycle
3.6 Memory-Reference Instructions
3.7 Input-Output and Interrupt
3.8 Complete Computer Description
3.9 Design of Basic Computer
3.10 Design of Accumulator Logic

3.1 INSTRUCTION CODE

An instruction code is a group of bits that instruct the computer to perform a specific operation.

Operation Code

The operation code of an instruction is a group of bits that define such operations as add, subtract, multiply, shift, and complement. The number of bits required for the operation code of an instruction depends on the total number of operations available in the computer. The operation code must consist of at least n bits for a given 2 (or less) distinct operations.

Accumulator (AC)

Computers that have a single-processor register usually assign to it the name accumulator (AC) accumulator and label it AC. The operation is performed with the memory operand and the content of AC.

Stored Program Organization

The simplest way to organize a computer is to have one processor register and an instruction code format with two parts. The first part specifies the operation to be performed and the second specifies an address and the memory address tells the control where to find an operand in memory. This operand is read from memory and used as the data to be operated on together with the data stored in the processor register. The following figure 3.1 shows this type of organization.

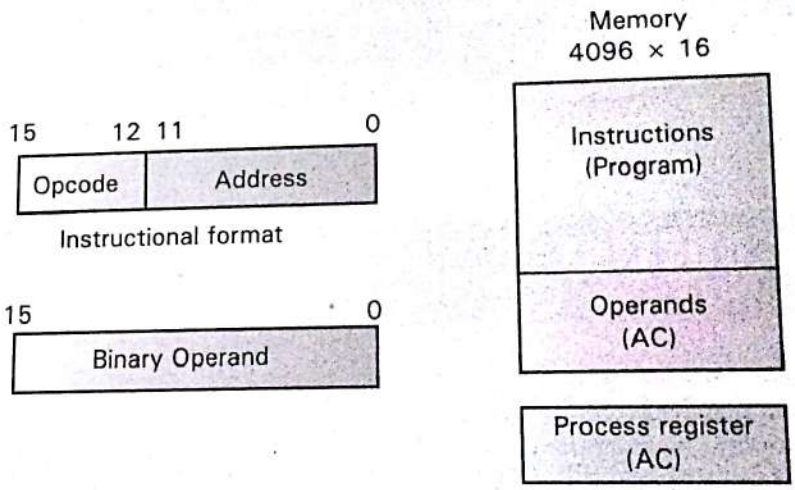


Figure 3.1: *Stored Program Organization*

Instructions are stored in one section of memory and data in another. For a memory unit with 4096 words, we need 12 bits to specify an address since $2^{12} = 4096$. If we store each instruction code in one 16-bit memory word, we have available four bits for operation code (abbreviated op code) to specify one out of 16 possible operations, and 12 bits to specify the address of an operand. The control reads a 16-bit instruction from the program portion of memory.

It uses the 12-bit address part of the instruction to read a 16-bit operand from the data portion of memory. It then executes the operation specified by the operation code.

Computers that have a single-processor register usually assign to it the name accumulator and label it AC. If an operation in an instruction code does not need an operand from memory, the rest of the bits in the instruction can be used for other purposes. For example, operations such as clear AC, complement AC, and increment AC operate on data stored in the AC register. They do not need an operand from memory. For these types of operations, the second part of the instruction code (bits 0 through 11) is not needed for specifying a memory address and can be used to specify other operations for the computer.

Direct and Indirect Addressing of Basic Computer

Thesecondpartofaninstructionformatspecifiestheaddressofanoperand, the instruction is said to have a **direct address**. In **Indirect address**, the bits in the second part of the instruction designate an address of a memory word in which the address of the operand is found. One bit of the instruction code can be used to distinguish between a direct and an indirect address. It consists of a 3-bit operation code, a 12-bit address, and an indirect address mode bit designated by I where, the mode bit is 0 for a direct address and 1 for an indirect address. A direct address instruction is shown in Figure 3.2. It is placed in address 22 in memory. The I bit is 0, so the instruction is recognized as a direct address instruction. The op code specifies an ADD instruction, and the address part is the binary equivalent of 457. The control finds the operand in memory at address 457 and adds it to the content of AC.

The instruction in address 35 shown in Figure 3.2 has a mode bit I = 1, recognized as an indirect address instruction. The address part is the binary equivalent of300. The control goes to address 300 to find the address of the operand. The address of the operand in this case is 1350. The operand found in address 1350 is then added to the content of AC. The indirect address instruction needs two references to memory to fetch an operand.

- The first reference is needed to read the address of the operand
- Second reference is for the operand itself.

The memory word that holds the address of the operand in an indirect address instruction is used as a pointer to an array of data.

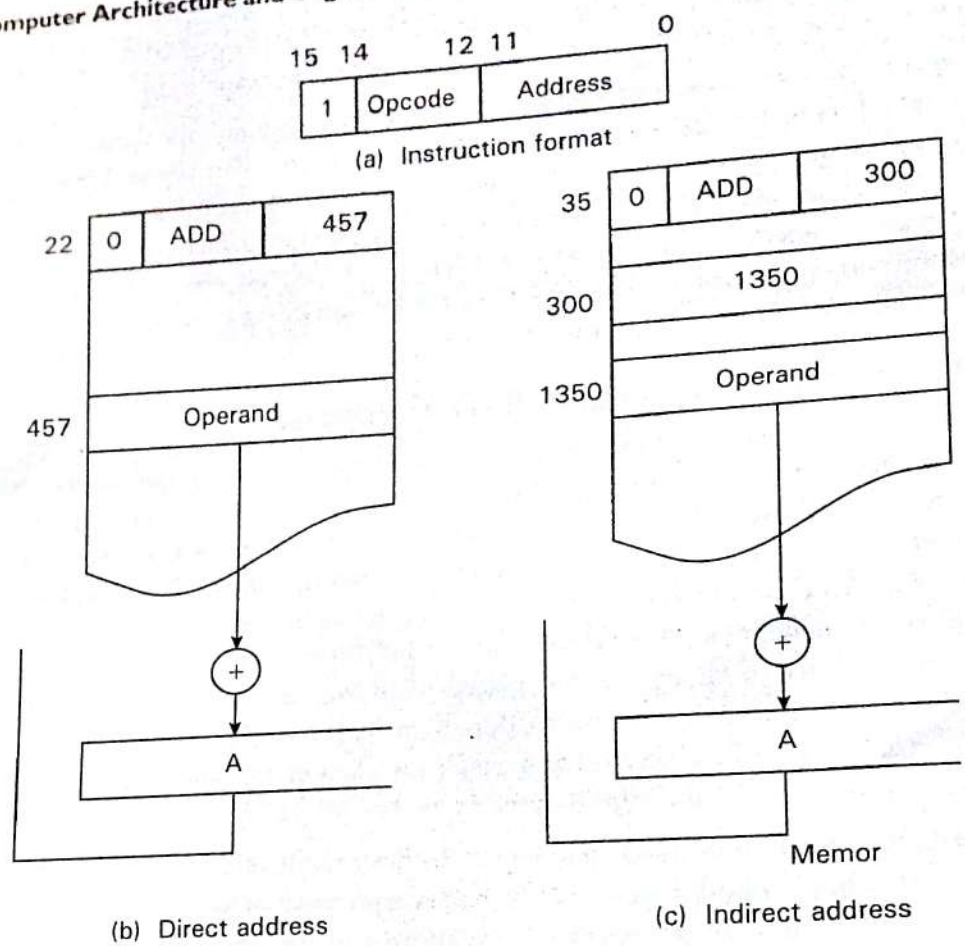


Figure 3.2: *Direct and indirect advertising of a basic computer*

Direct Address	Indirect Address
When the second part of an instruction code specifies the address of an operand, the instruction is said to have a direct address.	When the second part of an instruction code specifies the address of a memory word in which the address of the operand, the instruction is said to have a direct address.
For instance the instruction MOV R0 00H. R0, when converted to machine language is the physical address of register R0. The instruction moves 0 to R0.	For instance the instruction MOV @R0 00H, when converted to machine language, @R0 becomes whatever is stored in R0, and that is the address used to move 0 to. It can be whatever is stored in R0.

3.2 COMPUTER REGISTERS

It is necessary to provide a register in the control unit for storing the instruction code after it is read from memory. The computer needs processor registers for manipulating data and a

register for holding a memory address. These requirements dictate the register configuration shown in Figure 3.3.

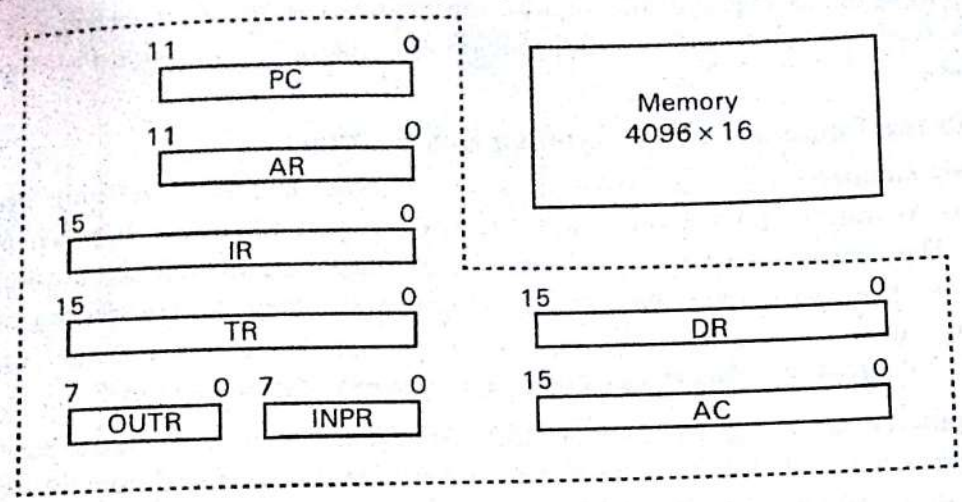


Figure 3.3: Basic Computer Register and Memory

The data register (DR) holds the operand read from memory and the accumulator (AC) register is a general purpose process in register. The instruction read from memory is placed in the instruction register(IR). The temporary register (TR) is used for holding temporary data during the processing.

The memory address register (AR) has 12bits and the program counter (PC) also has 12 bits and it holds the address of the next instruction to be read from memory after the current instruction is executed. Instruction words are read and executed in sequence unless a branch instruction is encountered. A branch instruction calls for a transfer to a nonconsecutive instruction in the program. Two registers are used for input and output. The input register (INPR) receives an 8-bit character from an input device. The output register (OUTR) holds an 8-bit character for an output device.

Register Symbol	Bits	Register Name	Function
DR	16	Data register	Holds memory operand
AR	12	Address register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction register	Holds instruction code
PC	12	Program counter	Holds address of instruction
TR	16	Temporary register	Holds temporary data
INPR	8	Input register	Holds input character
OUTR	8	Output register	Holds output character

Table 3.1: *List of Registers for Basic Computer*

Common Bus System for basic computer register

The common bus is required in computer for communication with registers and memory to decrease the hardware complexity and troubleshoot problems but it has performance bottleneck limitation.

What is the Requirement of Common Bus System?

The basic computer has eight registers, a memory unit and a control unit. Paths must be provided to transfer information from one register to another and between memory and register. The number of wires will be excessive if connections are between the outputs of each register and the inputs of the other registers. An efficient scheme for transferring information in a system with many register is to use a common bus. The connection of the registers and memory of the basic computer to a common bus system is shown in figure 3.4.

The outputs of seven registers and memory are connected to the common bus. The specific output that is selected for the bus lines at any given time is determined from the binary value of the selection variables S2, S1, and S0. The number along each output shows the decimal equivalent of the required binary selection. The particular register whose LD (load) input is enabled receives the data from the bus during the next clock pulse transition. The memory receives the contents of the bus when its write input is activated. The memory places its 16-bit output onto the bus when the read input is activated and S2 S1 S0 = 1 11.

Four registers, DR, AC, IR, and TR have 16 bits each and two registers, AR and PC, have 12 bits each since they hold a memory address. When the contents of AR or PC are applied to the 16-bit common bus, the four most significant bits are set to 0's. When AR and PC receive information from the bus, only the 12 least significant bits are transferred into the register. The input register INPR and the output register OUTR have 8 bits each and communicate with the eight least significant bits in the bus. INPR is connected to provide information to the bus but OUTR can only receive information from the bus.

Five registers have three control inputs: LD (load), INR (increment), and CLR (clear).Two registers have only a LD input. AR must always be used to specify a memory address; therefore memory address is connected to AR. The 16 inputs of AC come from an adder and logic circuit. This circuit has three sets of inputs.

- Set of 16-bit inputs come from the outputs of AC.
- Set of 16-bits come from the data register DR.
- Set of 8-bit inputs come from the input register INPR.

The result of an addition is transferred to AC and the end carry-out of the addition is transferred to flip-flop E (extended AC bit). The clock transition at the end of the cycle transfers the content of the bus into the designated destination register and the output of the adder and logic circuit into AC.

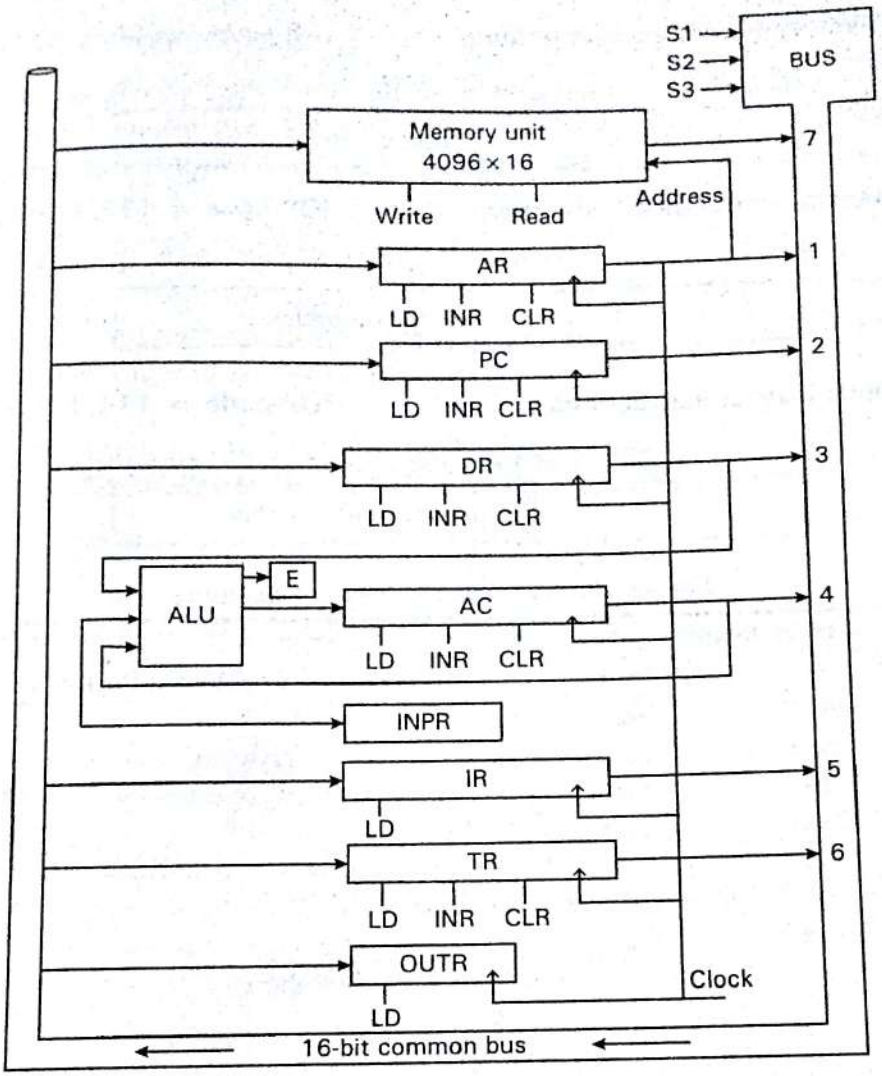


Figure 3.4: *Basic computer registers connected to a common bus*

3.3 Computer Instructions Format with its Types

The basic computer has three instruction code formats, as shown in figure 3.5.

Each format has 16bits. The operation code (op code) part of the instruction contains three bits and the meaning of the remaining 13 bits depends on the operation code encountered. A **memory-reference instruction** uses 12 bits to specify an address and one bit to specify the addressing mode I. I is equal to 0 for direct address and to 1 for indirect address. The **register reference instructions** are recognized by the operation code 111 with a 0 in the leftmost bit (bit 15) of the instruction. A register-reference instruction specifies an operation on or a test of the AC register. An operand from memory is not needed; therefore, the other 12 bits are used to specify the operation or test to be executed. An **input-output instruction** does not need a reference to memory and is recognized by the operation code 111 with a 1 in the leftmost bit of the instruction. The remaining 12 bits are used to specify the type of input-output operation or test performed.

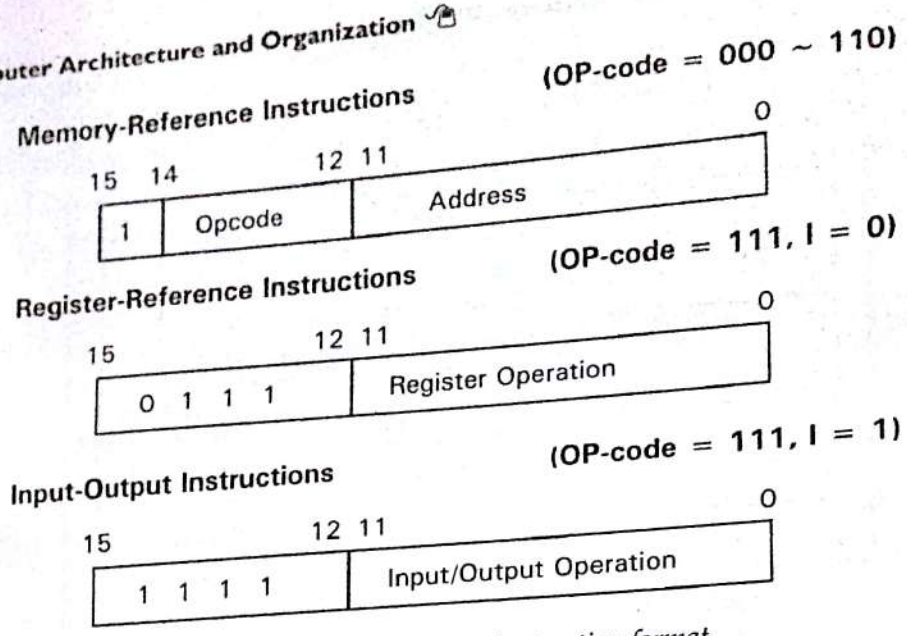


Figure 3.5: *Basic computer instruction format*

Symbol	Hexadecimal code I = 0	Hexadecimal code I = 1	Description
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Store content of AC in memory
STA	3xxx	Bxxx	Branch unconditionally
BUN	4xxx	Cxxx	Branch and save return address
BSA	5xx	Dxxx	Increment and skip if zero
ISZ	6xx	Exxx	
CLA	7800		Clear AC
CLE	7400		Clear E
CMA	7200		Complement Ac
CME	7100		Complement E
CIR	7080		Circulate right AC and E
INC	7020		Circulate left AC and E
SPA	7010		Increment AC
SNA	7008		Skip next instruction if AC positive
SZA	7004		Skip next instruction if AC negative
SZE	7002		Skip next instruction if E is 0
HLT	7001		Halt computer
INP	F800		Input character to AC
OUT	F400		Output character from AC
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt on
IOF	F040		Interrupt off

Table 3.2: *Basic computer instructions*

Instruction Set Completeness

A computer should have a set of instructions so that that user can construct machine language programs to evaluate any function that is known to be compatible. The set of instructions are said to be complete if the computer includes a sufficient number of instructions in each of the following categories:

- Arithmetic, logical and shift instructions
- Instructions for moving information to and from memory and processor registers.
- Program control instructions logo then with instructions that check station conditions.
- Input and output instructions.

3.4 TIMING AND CONTROL UNIT

The block diagram of the control unit is shown in figure 3.6. Components of Control unit are

- Two decoders
- A sequence counter
- Control logic gates

An instruction read from memory is placed in the instruction register (IR). In control unit the IR is divided into three parts: I bit, the operation code (12-14)bit, and bits 0 through 11. The operation code in bits 12 through 14 are decoded with a 3 X 8decoder.

Bit-15 of the instruction is transferred to a flip-flop designated by the symbol I. The eight outputs of the decoder are designated by the symbols D0 through D7. Bits 0 through 11 are applied to the control logic gates. The 4 - bit sequence counter can count in binary from 0 through 15.The outputs of counter are decoded into 16 timing signals T0 throughT_{15}.

The sequence counter SC can be incremented or cleared synchronously. Most of the time, the counter is incremented to provide the sequence of timing signals out of 4 X 16 decoder. Once in awhile, the counter is cleared to 0, causing the next timing signal to be T0. As an example, consider the case where SC is incremented to provide timing signals T0, T_1, T_2, T_3 and T_4 in sequence. At time T_4, SC is cleared to 0 if decoder output D3 is active. This is expressed symbolically by the statement

$$D_3T_4: SC \leftarrow 0$$

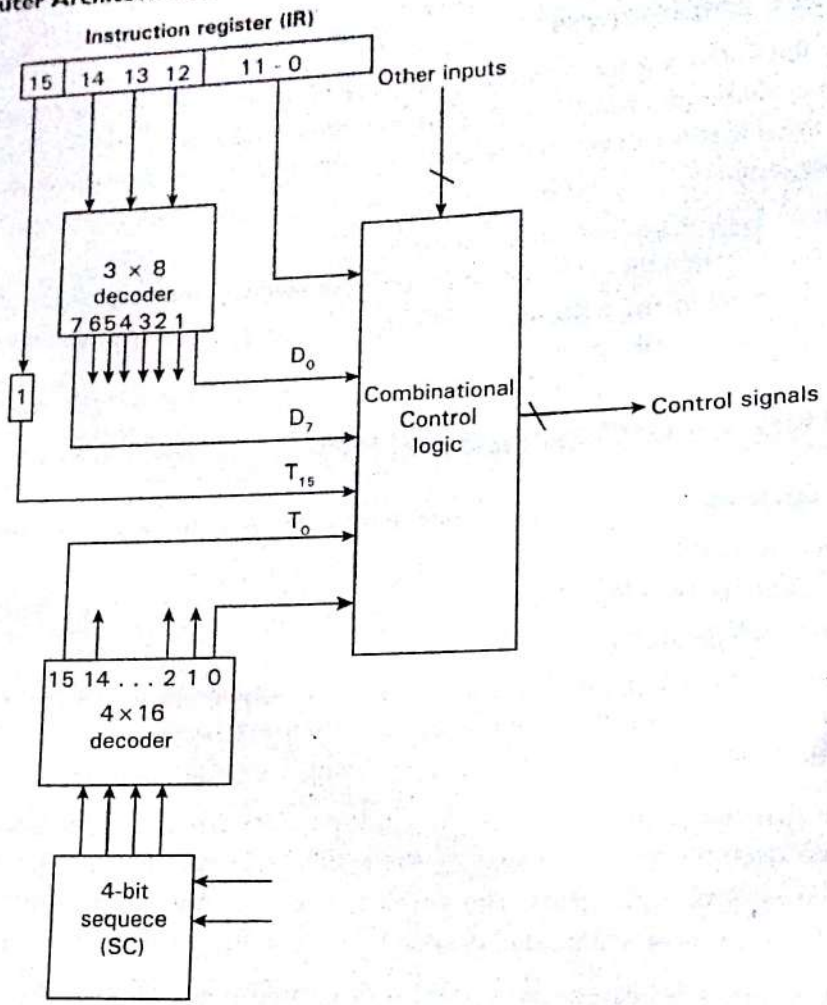


Figure 3.6: *Control unit of basic computer*

Timing Diagram

The timing diagram figure 3.7 shows the time relationship of the control signals. The sequence counter SC responds to the positive transition of the clock. Initially, the CLR input of SC is active. The first positive transition of the clock clears SC to 0, which in turn activates the timing T_0 out of the decoder. T_0 is active during one clock cycle. The positive clock transition labeled T_0 in the diagram will trigger only those registers whose control inputs are connected to timing signal T_0. SC is incremented with every positive clock transition, unless its CLR input is active. This procedures the sequence of timing signals T_0, T_1, T_2, T_3 and T_4, and so on. If SC is not cleared, the timing signals will continue with T_5, T_6, up to T_{15} and back toT_0.

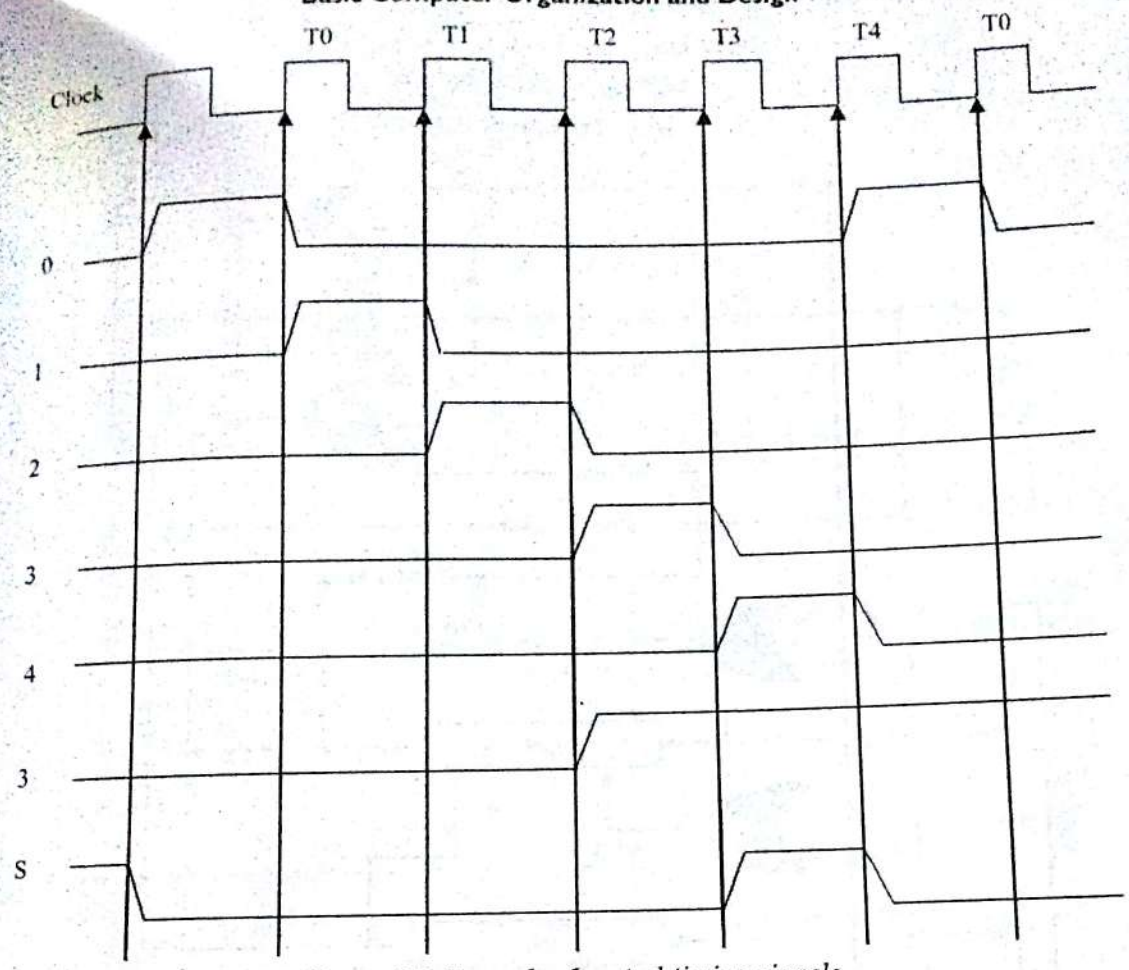


Figure 3.7: *Example of control timing signals*

The last three waveforms shows how SC is cleared when D3T4 = 1. Output D3 from the operation decoder becomes active at the end of timing signal T_2. When timing signal T_4 becomes active, the output of the AND gate that implements the control function D_3T_4 becomes active. This signal is applied to the CLR input of SC. On the next positive clock transition the counter is cleared to 0. This causes the timing signal $T_0$to become active instead of T_5 that would have been active if SC were incremented instead of cleared.

3.5 INSTRUCTION CYCLE

- A program residing in the memory unit of the computer consists of a sequence of instructions. In the basic computer each instruction cycle consists of the following phases:
  - Fetch an instruction from memory.
  - Decode the instruction.
  - Read the effective address from memory if the instruction has an indirect address.
  - Execute the instruction.
- After step 4, the control goes back to step 1 to fetch, decode and execute the next instruction.
- This process continues unless a HALT instruction is encountered.

T_0 : $AR \leftarrow PC$

T_1 : $IR \leftarrow M[AR], \quad PC \leftarrow PC + 1$

T_2 : $D_0, \ldots, D_7 \leftarrow$ Decode IR (12 – 14), $AR \leftarrow IR(0 - 11), I \leftarrow IR(15)$

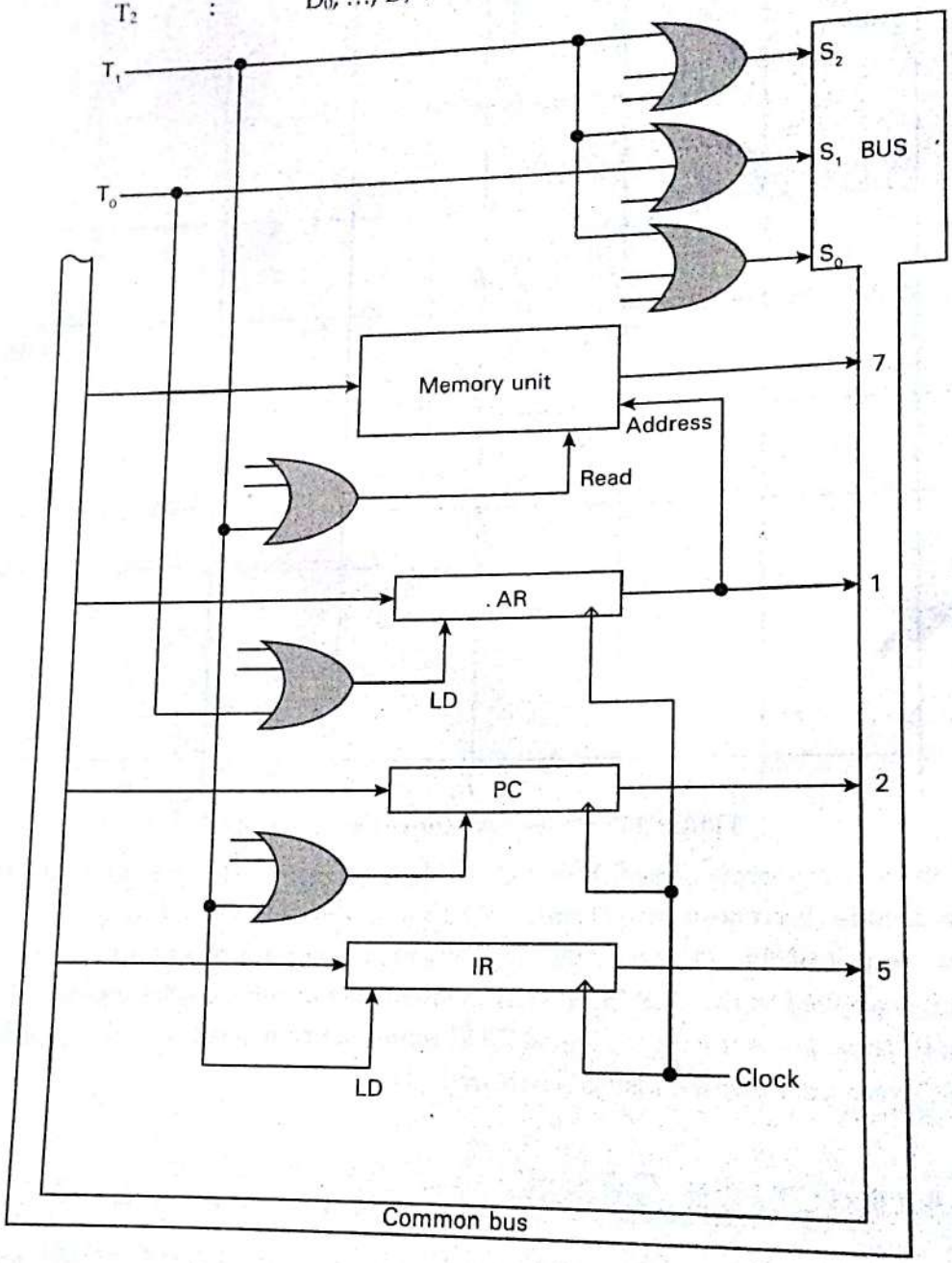


Figure 3.8: *Register transfer for the fetch phase*

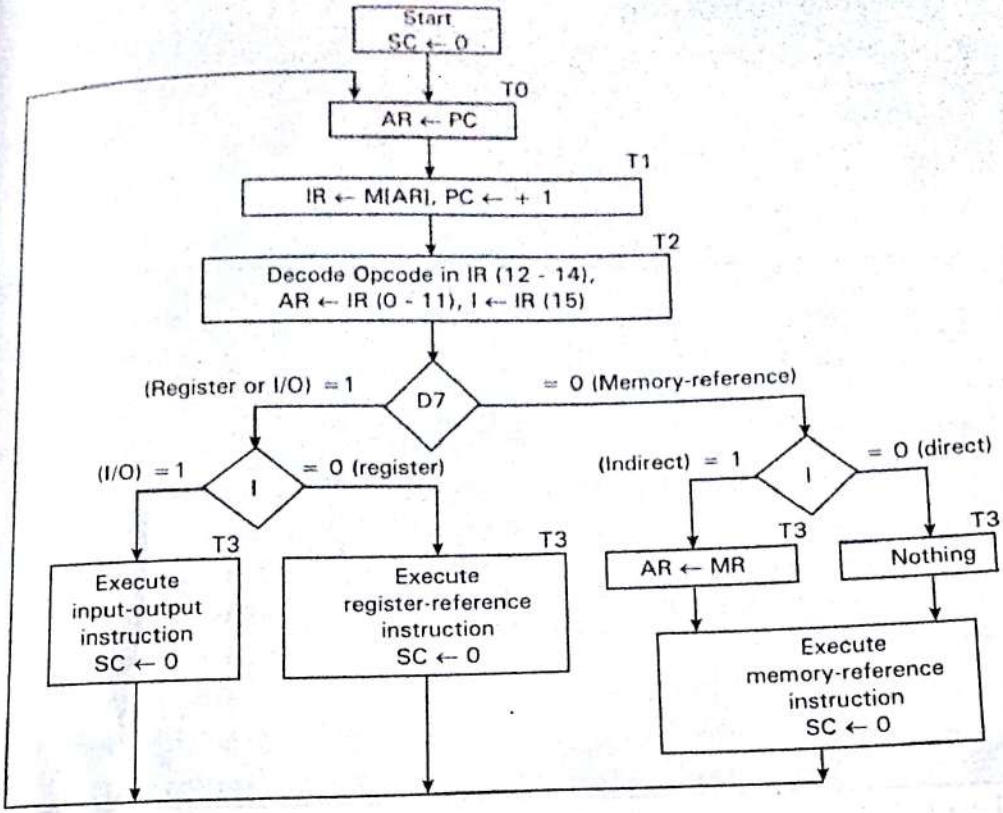


Figure 3.9: *Flowchart for instruction cycle (initial configuration)*

The flowchart presents an initial configuration for the instruction cycle and shows how the control determines the instruction type after the decoding. If D7 = 1, the instruction must be register-reference or input-output type. If D7 = 0, the operation code must be one of the other seven values 110, specifying a memory- reference instruction. Control then inspects the value of the first bit of the instruction, which now available in flip-flop I. If D7 = 0 and I = 1, we have a memory-reference instruction with an indirect address. It is then necessary to read the effective address from memory.

The three instruction types are subdivided into four separate paths. The selected operation is activated with the clock transition associated with timing signal T_3.This can be symbolized as follows:

D'7 I T3 :	AR ← M[AR]
D7 I' T3 :	Execute a register-reference instruction
D7 I T3 :	Execute an input-output instruction

When a memory-reference instruction with I = 0 is encountered, it is not necessary to do anything since the effective address is already in AR. However, the sequence counter SC must be incremented when D'7 I T3 = 1, so that the execution of the memory-reference instruction can be continued with timing variableT4. A register-reference or input-output instruction can be executed with the click associated with timing signal T3. After the instruction is executed, SC is cleared to 0 and control returns to the fetch phase with T0 =1. SC is either incremented or cleared to 0 with every positive clock transition.

Register reference instruction

- When the register-reference instruction is decoded, D7 bit is set to1.
- Each control function needs the Boolean relation D7 I'T3

15	12 11	0
0 1 1 1	Register Operation	

Table 3.3 : *Execution of register-reference instructions*

	r:	SC←0	Clear SC
CLA	rB_{11}:	AC ←0	Clear AC
CLE	rB_{10}:	E ←0	Clear E
CMA	rB_9:	AC ←AC'	Complement AC
CME	rB_8:	E ←E'	Complement E
CIR	rB_7:	AC ←shr AC, AC(15) ←E, E ←AC(0)	Circular Right
CIL	rB_6:	AC ←shl AC, AC(0) ←E, E ←AC(15)	Circular Left
INC	rB_5:	AC ←AC + 1	Increment AC
SPA	rB_4:	if (AC(15) = 0) then (PC ←PC+1)	Skip if positive
SNA	rB_3:	if (AC(15) = 1) then (PC ←PC+1	Skip if negative
SZA	rB_2:	if (AC = 0) then (PC ←PC+1)	Skip if AC is zero
SZE	rB_1:	if (E = 0) then (PC ←PC+1)	Skip if E is zero
HLT	rB_0:	S ←0 (S is a start-stop flip-flop)	Halt computer

- These 12 bits are available in IR (0-11). They were also transferred to AR during timeT_2.
- These instructions are executed at timing cycleT_3.
- The first seven register-reference instructions perform clear, complement, circular shift, and increment microoperations on the AC or E registers.
- The next four instructions cause a skip of the next instruction in sequence when condition is satisfied. The skipping of the instruction is achieved by incrementing PC.
- The condition control statements must be recognized as part of the control conditions. The AC is positive when the sign bit in AC(15) = 0; it is negative when AC(15) = 1. The content of AC is zero (AC = 0) if all the flip-flops of the register are zero.
- The HLT instruction clears a start-stop flip-flop S and stops the sequence counter from counting. To restore the operation of the computer, the start-stop flip-flop must be set manually.

3.6 MEMORY REFERENCE INSTRUCTIONS

- When the memory-reference instruction is decoded, D7 bit is set to0.

15	14	12 11	0
I	000~110	Address	

Table 3.4: *Memory-reference instructions*

Symbol	Operation Decoder	Symbolic Description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR], E \leftarrow C_{out}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC, PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1$, if $M[AR] + 1 = 0$ then $PC \leftarrow PC+1$

- The effective address of the instruction is in the address register AR and was placed there during timing signal T_2 when I = 0, or during timing signal T_3 when I =1.
- The execution of the memory-reference instructions starts with timing signalT_4.

AND to AC

This is an instruction that performs the AND logic operation on pairs of bits in AC and the memory word specified by the effective address. The result of the operation is transferred to AC.

D_0T_4: $DR \leftarrow M[AR]$

D_0T_5: $AC \leftarrow AC \wedge DR, SC \leftarrow 0$

ADD to AC

This instruction adds the content of the memory word specified by the effective address to the value of AC. The sum is transferred into AC and the output carry C_{out} is transferred to the E (extended accumulator) flip-flop.

D_1T_4: $DR \leftarrow M[AR]$

D_1T_5: $AC \leftarrow AC + DR, E \leftarrow C_{out}, SC \leftarrow 0$

LDA: Load to AC

This instruction transfers the memory word specified by the effective address to AC.

D_2T_4: $DR \leftarrow M[AR]$

D_2T_5: $AC \leftarrow DR, SC \leftarrow 0$

STA: Store AC

This instruction stores the content of AC into the memory word specified by the effective address.

D_3T_4: $M[AR] \leftarrow AC, SC \leftarrow 0$

BUN: Branch Unconditionally

This instruction transfers the program to instruction specified by the effective address. The BUN instruction allows the programmer to specify an instruction out of sequence and the program branches (or jumps) unconditionally.

D_4T_4: $PC \leftarrow AR, SC \leftarrow 0$

BSA: Branch and Save Return Address

This instruction is useful for branching to a portion of the program called a subroutine or procedure. When executed, the BSA instruction stores the address of the next instruction in sequence (which is available in PC) into a memory location specified by the effective address.

$$M[AR] \leftarrow PC,\ PC \leftarrow AR + 1$$

$$M[135] \leftarrow 21,\ PC \leftarrow 135 + 1 = 136$$

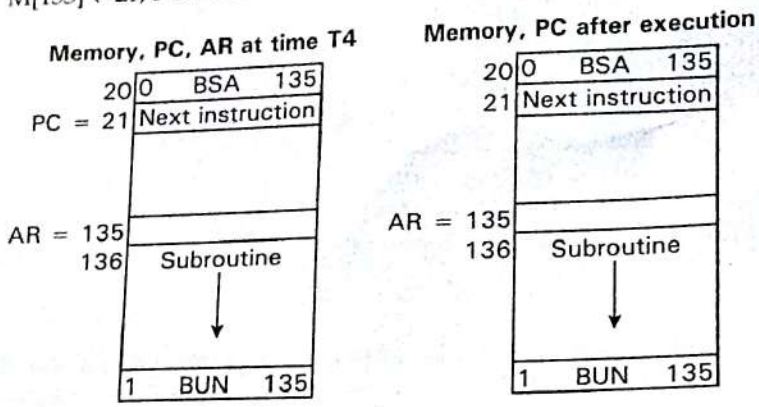


Figure 3.10: *Example of BSA instruction execution*

It is not possible to perform the operation of the BSA instruction in one clock cycle when we use the bus system of the basic computer. To use the memory and the bus properly, the BSA instruction must be executed with a sequence of two microoperations:

D_5T_4: $M[AR] \leftarrow PC,\ AR \leftarrow AR + 1$

D_5T_5: $PC \leftarrow AR,\ SC \leftarrow 0$

ISZ: Increment and Skip if Zero

These instruction increments the word specified by the effective address, and if the incremented value is equal to 0, PC is incremented by 1. Since it is not possible to increment a word inside the memory, it is necessary to read the word into DR, increment DR, and store the word back into memory.

D_6T_4: $DR \leftarrow M[AR]$

D_6T_5: $DR \leftarrow DR + 1$

D_6T_6: $M[AR] \leftarrow DR$, if $(DR = 0)$ then $(PC \leftarrow PC + 1)$, $SC \leftarrow 0$

Control Flowchart

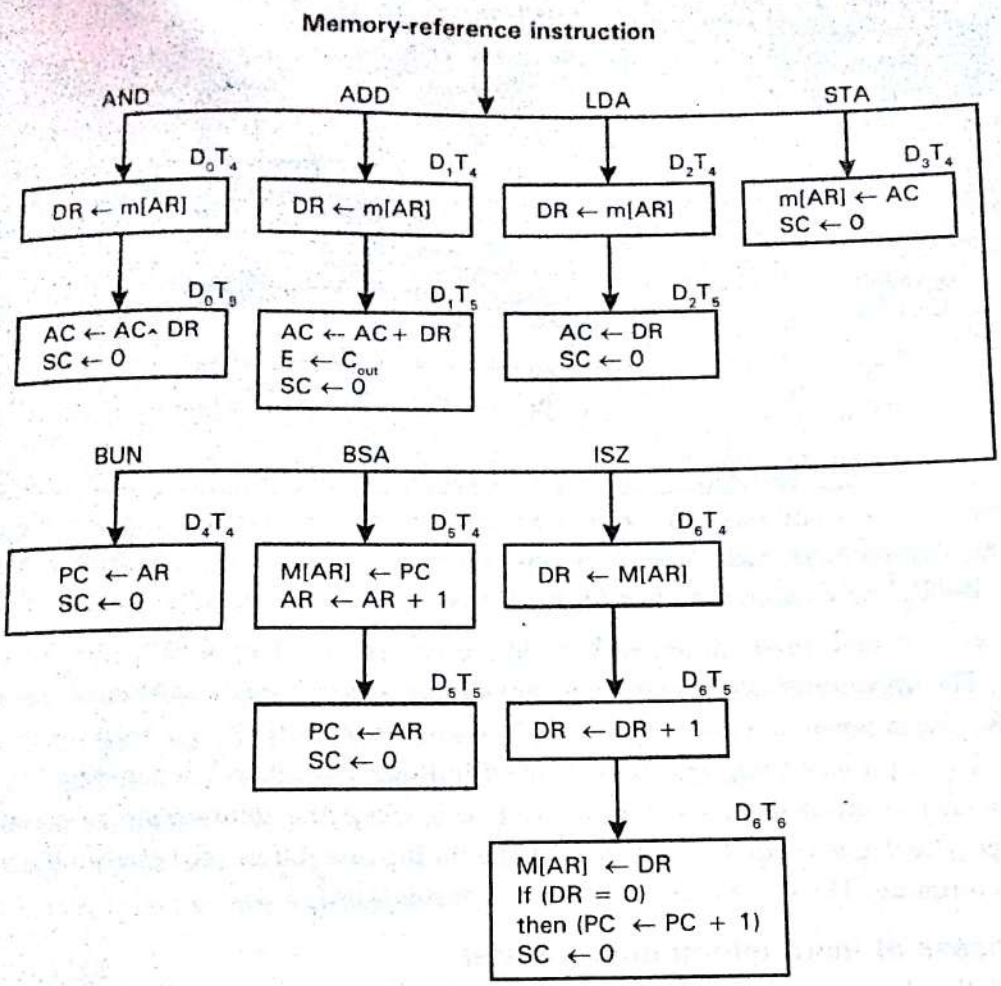


Figure 3.11: *Flowchart for memory-reference instructions*

3.7 INPUT-OUTPUT CONFIGURATION OF BASIC COMPUTER

A computer can serve no useful purpose unless it communicates with the external environment. To exhibit the most basic requirements for input and output communication, we will use a terminal unit with a keyboard and printer.

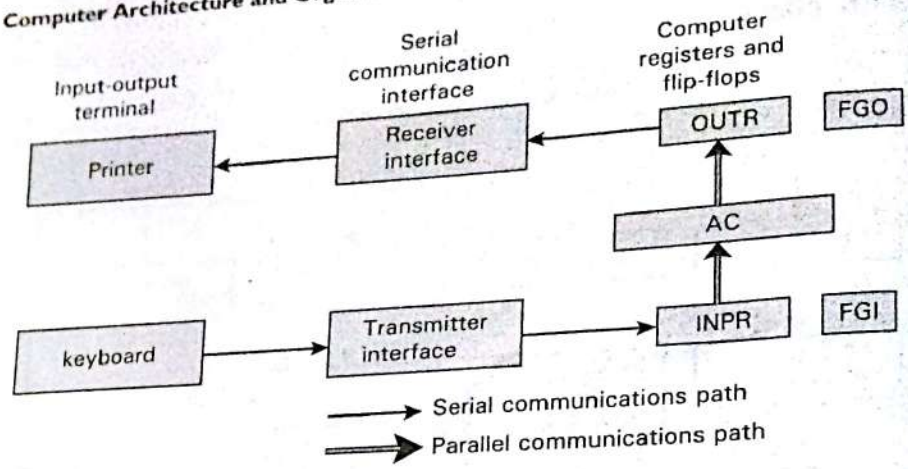


Figure 3.12: *Input-output configuration*

The terminal sends and receives serial information and each quantity of information has eight bits of an alphanumeric code. The serial information from the keyboard is shifted into the input register INPR. The serial information for the printer is stored in the output register OUTR.

These two registers communicate with a communication interface serially and with the AC in parallel. The transmitter interface receives serial information from the keyboard and transmits it to INPR. The receiver interface receives information from OUTR and sends it to the printer serially. The 1-bit input flag FGI is a control flip-flop. It is set to 1 when new information is available in the input device and is cleared to 0 when the information is accepted by the computer. The flag is needed to synchronize the timing rate difference between the input device and the computer. The process of information transfer is as follows:

The process of input information transfer

- Initially, the input flag FGI is cleared to 0. When a key is struck in the keyboard, an 8-bit alphanumeric code is shifted into INPR and the input flag FGI is set to1.
- As long as the flag is set, the information in INPR cannot be changed by striking another key. The computer checks the flag bit; if it is 1, the information from INPR is transferred in parallel into AC and FGI is cleared to 0.
- Once the flag is cleared, new information can be shifted into INPR by striking another key.

The process of outputting information

- The output register OUTR works similarly but the direction of information flowis reversed.
- Initially, the output flag FGO is set to 1. The computer checks the flag bit; if it is 1, the information from AC is transferred in parallel to OUTR and FGO is cleared to 0. The output device accepts the coded information, prints the corresponding character, and when the operation is completed, it sets FGO to1.
- The computer does not load a new character into OUTR when FGO is 0 because this condition indicates that the output device is in the process of printing the character.

Input-Output instructions

Input and output instructions are needed for transferring information to and from AC register, for checking the flag bits, and for controlling the interrupt facility. Input-outputinstructionshaveanoperationcode1111andarerecognizedbythecontrol when D7 = 1 and I =1. The remaining bits of the instruction specify the particular operation. The control functions and microoperations for the input-output instructions are listed below.

INP	AC(0-7) ←INPR, FGI ←0	Input char. to AC
OUT	OUTR ←AC(0-7), FGO ←0	Output char. from AC
SKI	if(FGI = 1) then (PC ←PC + 1)	Skip on input flag
SKO	if(FGO = 1) then (PC ←PC + 1)	Skip on output flag
ION	IEN ←1	Interrupt enable on
IOF	IEN ←0	Interrupt enable off

Table 3.5: *Input Output Instructions*

The INP instruction transfers the input information from INPR into the eight low-order bits of AC and also clears the input flag to 0. The OUT instruction transfers the eight least significant bits of AC into the output register OUTR and clears the output flag to 0.

The next two instructions in Table 3.5 check the status of the flags and cause a skip of the next instruction if the flag is 1. The instruction that is skipped will normally be a branch instruction to return and check the flag again. The branch instruction is not skipped if the flag is 0.If the flag is 1, the branch instruction is skipped and an input or output instruction is executed. The last two instructions set and clear an interrupt enable flip-flop IEN. The purpose of IEN is explained in conjunction with the interrupt operation.

Interrupt Cycle

The way that the interrupt is handled by the computer can be explained by means of the flowchart shown in figure 3.13. An interrupt flip-flop R is included in the computer. When R = 0, the computer goes through an instruction cycle. During the execute phase of the instruction cycle IEN is checked by the control. If it is 0, it indicates that the programmer does not want to use the interrupt, so control continues with the next instruction cycle. If IEN is 1, control checks the flag bits. If both flags are 0, it indicates that neither the input nor the output registers are ready for transfer of information. In this case, control continues with the next instruction cycle. If either flag is set to 1 while IEN = 1, flip-flop R is set to1. At the end of the execute phase, control checks the value of R, and if it is equal to 1,it goes to an interrupt cycle instead of an instruction cycle.

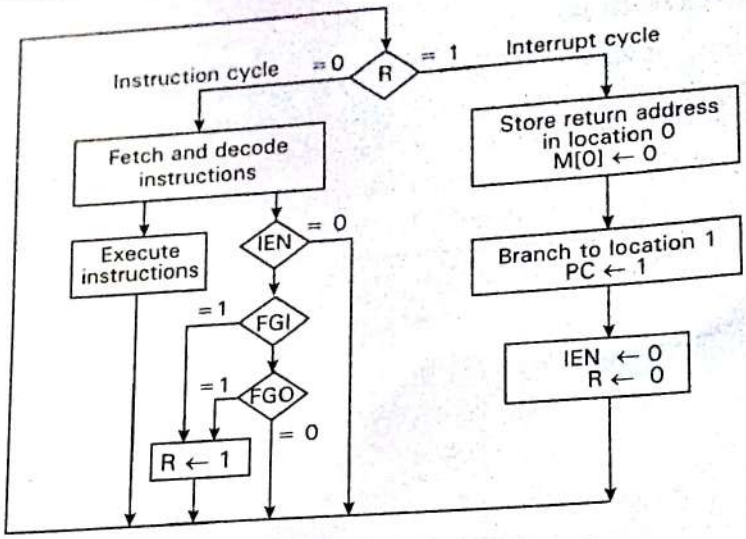


Figure 3.13: *Flowchart for interrupt cycle*

The interrupt cycle is a hardware implementation of a branch and save return address operation. The return address available in PC is stored in a specific location where it can be found later when the program returns to the instruction at which it was interrupted. This location may be a processor register, a memory stack, or a specific memory location. Here we choose the memory location at address 0 as the place for storing the return address. Controltheninsertsaddress1intoPCandclearsIENandRsothatnomoreinterruptions can occur until the interrupt request from the flag has been serviced. An example that shows what happens during the interrupt cycle is shown in Figure 3.14:

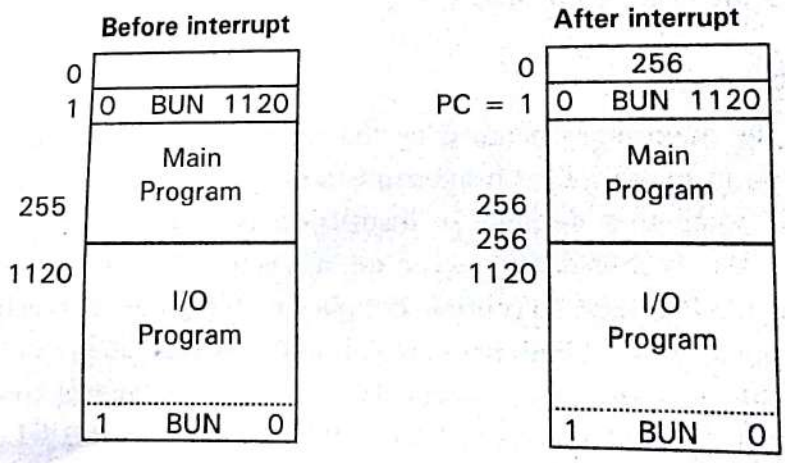


Figure 3.14: *Demonstration of the interrupt cycle*

Suppose that an interrupt occurs and R=1, while the control is executing the instruction at address 255. At this time, the return address 256 is in PC. The programmer has previously placed an input-output service program in memory starting from address 1120 and a BUN 1120 instruction at address 1. The content of PC (256) is stored in memory location 0, PC is set to 1, and R is cleared to 0. At the beginning of the next instruction cycle, the instruction that is read from memory is in address 1 since this is the content of PC. The branch instruction at address 1 causes the program to transfer to the input-output service program at address 1120.

This program checks the flags, determines which flag is set, and then transfers the required input or output information. Once this is done, the instruction ION is executed to set IEN to 1 (to enable further interrupts), and the program returns to the location where it was interrupted.

The instruction that returns the computer to the original place in the main program is a branch indirect instruction with an address part of 0. This instruction is placed at the end of the I/O service program. The execution of the indirect BUN instruction results in placing into PC the return address from location0.

Register transfer statements for the interrupt cycle

The flip-flop is set to 1 if IEN =1 and either FGI or FGO are equal to1.Thiscanhappen with any clock transition except when timing signals T_0, T_1 or T_2 are active. The condition for setting flip-flop R= 1 can be expressed with the following register transfer statement:

$$T_0'T_1'T_2'(IEN)(FGI + FGO): R \leftarrow 1$$

The symbol + between FGI and FGO in the control function designates a logic OR operation. This is AND with IEN and $T_0'T_1'T_2'$. The fetch and decode phases of the instruction cycle must be modified and Replace T_0, T_1, T_2 with $R'T_0$, $R'T_1$, $R'T_2$. Therefore the interrupt cycle statements are :

RT_0: $AR \leftarrow 0, TR \leftarrow PC$

RT_1: $M[AR] \leftarrow TR, PC \leftarrow 0$

RT_2: $PC \leftarrow PC + 1, IEN \leftarrow 0, R \leftarrow 0, SC \leftarrow 0$

During the first timing signal AR is cleared to 0, and the content of PC is transferred to the temporary register TR. With the second timing signal, the return address is stored in memory at location 0and PC is cleared to 0. The third timing signal increments PC to 1, clears IEN and R, and control goes back to T_0 by clearing SC to 0. The beginning of the next instruction cycle has the condition RT_0 and the content of PC is equal to 1. The control then goes through an instruction cycle that fetches and executes the BUN instruction in location 1.

3.8 Complete Computer Description

Flow chart for computer operation

The final flowchart of the instruction cycle, including the interrupt cycle for the basic computer, is shown in Figure 3.15. The interrupt flip-flop R may be set at any time during the indirect or execute phases. The control returns to timing signal T_0 after SC is cleared to 0.

If R = 1, the computer goes through an interrupt cycle. If R = 0, the computer goes through an instruction cycle. If the instruction is one of the memory-reference instructions, the computer first checks if there is an indirect address and then continues to execute the decoded instruction according to the flow chart. If the instruction is one of the register-reference instructions, it is executed with one of the microoperations register reference. If it is an input-output instruction, it is executed with one of the microoperation's input- output reference.

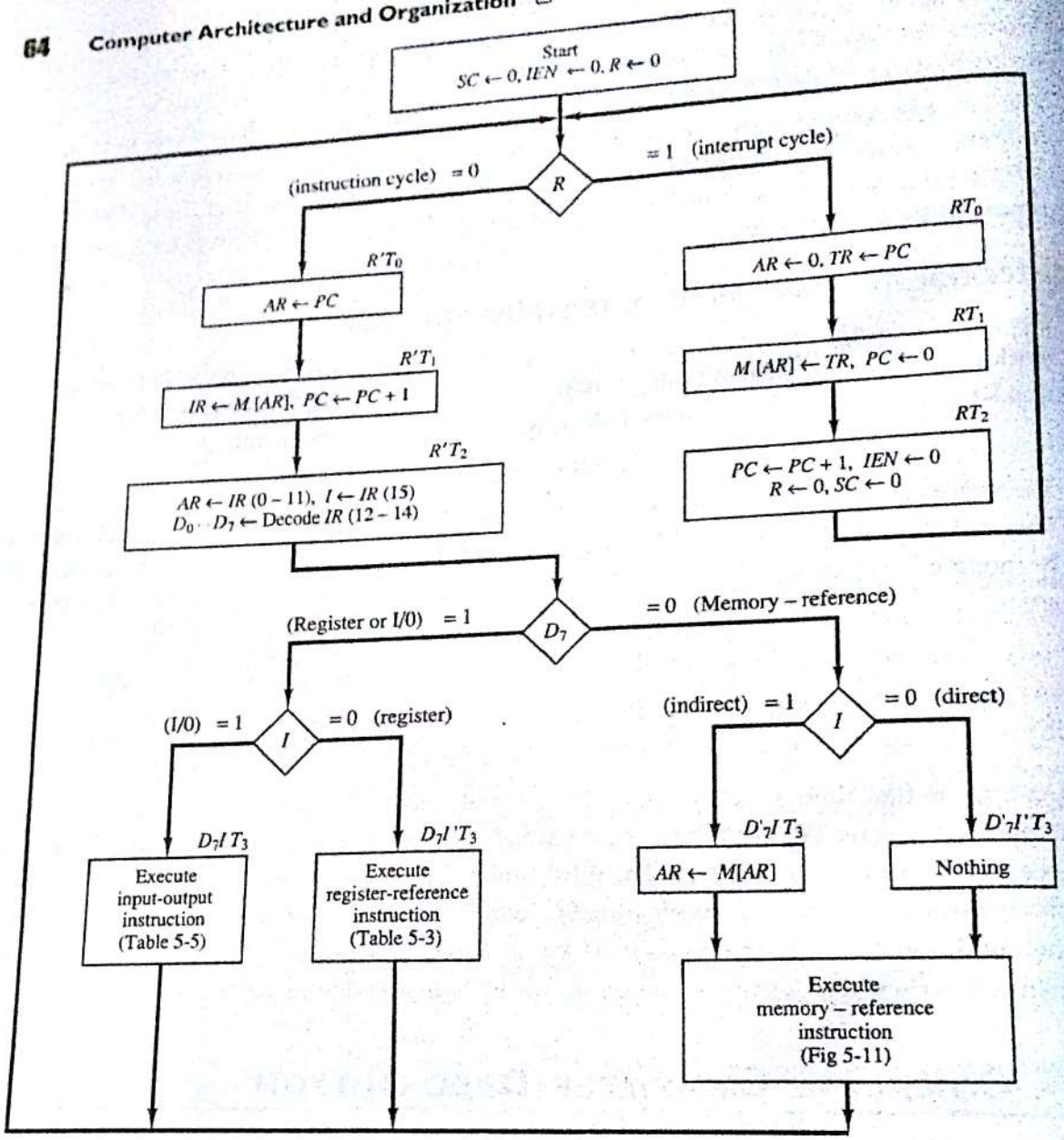


Figure 3.15: *Flowchart for computer operation*

Fetch	$F'T_0$:	$AR \leftarrow PC$
	$R'T_1$:	$IR \leftarrow M[AR],\ PC \leftarrow PC + 1$
Decode	$R'T_2$:	$D_0, \ldots, \leftarrow$ Decode IR (12 - 14),
		$AR \leftarrow IR\ (0 - 11),\ I \leftarrow IR\ (15)$
Indirect interrupt:	$D_7'IT_3$:	$AR \leftarrow M[AR]$
	$T_0'T_1'T_2'(IEN)\ (FGI + FGO)$:	$R \leftarrow 1$
	RT_0:	$AR \leftarrow 0,\ TR \leftarrow PC$
	RT_1:	$M[AR] \leftarrow TR,\ PC \leftarrow 0$
	RT_2:	$PC \leftarrow PC + 1,\ IEN \leftarrow 0,\ R \leftarrow 0,\ SC \leftarrow 0$
Memory Reference:		
AND		
	D_0T_4:	$DR \leftarrow M[AR]$
	D_0T_5:	$AC \leftarrow AC \wedge DR,\ SC \leftarrow 0$

ADD	D_1T_4 : DR ← M[AR]
	D_1T_5 : AC ← AC + DR, E ← C_{out}, SC ←
LDA	D_2T_4 : DR ← M[AR]
	D_2T_5 : AC ← DR, SC ← 0
STA	D_3T_5 : M[AAR] ← AAC, SC ← 0
BUN	D_4T_4 : PC ← AR, SC ← 0
BSA	D_5T_4 : M[AR] ← PC, AR ← AR + 1
	D_5T_5 : PC ← AR, SC ← 0
ISZ	D_6T_4 : DR ← M[AR]
	D_6T_5 : DR ← DR + 1
	D_6T_6 : M[AR] ← DR, If (DR = 0) then (PC ← PC + 1), SC ← 0
Register Reference:	
	$D_7I'T_3$ = r (common to all register reference instruction)
	IR(i) = B_i (i = 0, 1, 2, …, 11)
	r : SC ← 0
CLA	rB_{11} : AC ← 0
CLE	rB_{10} : E ← 0
CMA	rB_9 : AC ← $\overline{AC}$
CME	rB_8 : E ← $\overline{E}$
CI	rB_7 : AC ← shr AC, AC (15) ← E, E ← AC(0
CIL	rB_6 : AC ← shl AC, AC (0) ← E, E ← AC(15)
INC	rB_5 : AC ← AC + 1
SPA	rB_4 : If (AC(15) = 0) then (PC ← PC + 1)
SN	rB_3 : If (AC(15) = 1) then (PC ← PC + 1)
SZA	rB_2 : If (AC = 0) then (PC ← PC + 1)
SZE	rB_1 : If (E = 0) then (PC ← PC + 1)
HLT	rB_0 : S ← 0
Input-Output:	
	$D_7I'T_3$ = p (common to all input-output instruction)
	IR(i) = B_i (i = 6, 7, 8, …, 11)
	r : SC ← 0
INP	$_pB_{11}$: AC(0-7) ← INPR, FGI ← 0
OUT	$_pB_{10}$: OUTR ← AC(0-7), FGO ← 0
SKI	$_pB_9$: If (FGI = 1) then (PC ← PC + 1)
SKO	$_pB_8$: If (FG0 = 1) then (PC ← PC + 1)
ION	$_pB_7$: IEN ← 1
IOF	$_pB_6$: IEN ← 0

Table 3.6: *Control functions and microoperations for the basic computer*

3.9 Design of Basic Computer

Hardware Components of BC

A memory unit: 4096 x 16.

Registers: AR, PC, DR, AC, IR, TR, OUTR, INPR, and SC

Flip-Flops(Status): I, S, E, R, IEN, FGI, and FGO

Decoders: a 3x8 Opcode decoder

a 4x16 timing decoder

Common bus: 16 bits

Control logic gates: Adder and Logic circuit connected to AC

Control Logic Gates

- Input Controls of the nine registers
- Read and Write Controls of memory
- Set, Clear, or Complement Controls of the flip-flops
- S2, S1, S0 Controls to select a register for the bus
- AC, and Adder and Logic circuit

Control of AR register

Scan all of the register transfer statements that change the content of AR:

$R'T_0$:	$AR \leftarrow PC$	LD(AR)
$R'T_2$:	$AR \leftarrow IR(0\text{-}11)$	LD(AR)
D'_7IT_3:	$AR \leftarrow M[AR]$	LD(AR)
RT_0:	$AR \leftarrow 0$	CLR(AR)
D_5T_4:	$AR \leftarrow AR + 1$	INR(AR)

Now,

$LD(AR) = R'T_0 + R'T_2 + D'_7IT_3$

$CLR(AR) = RT_0$

$INR(AR) = D_5T_4$

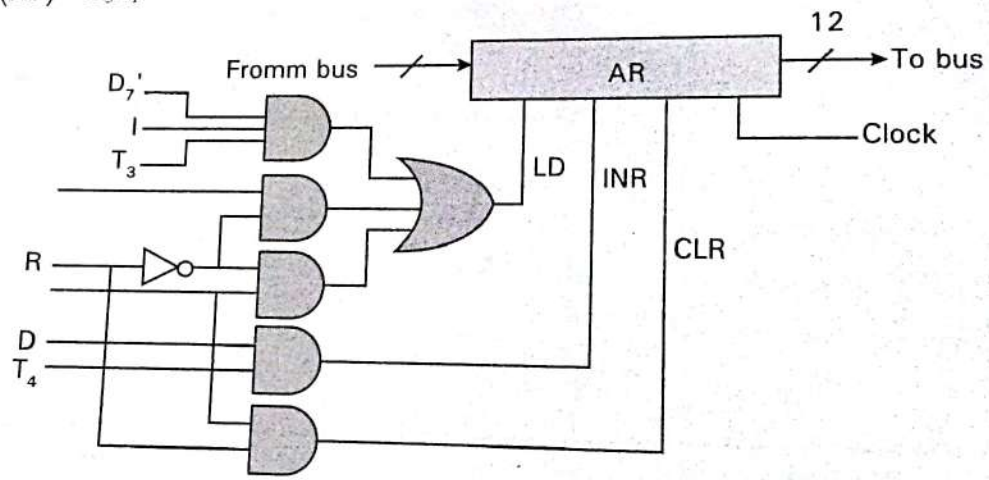


Figure 3.16: *Control Gates Associated with AR*

Control of IEN Flip-Flop

pB_7: $IEN \leftarrow 1$ (I/O Instruction)

pB_6: $IEN \leftarrow 0$ (I/O Instruction)

RT_2: $IEN \leftarrow 0$ (Interrupt)

=>

Set (IEN) = pB_7

Clear(IEN) = $pB_6 + RT_2$

$p = D_7IT_3$ (Input/Output Instruction)

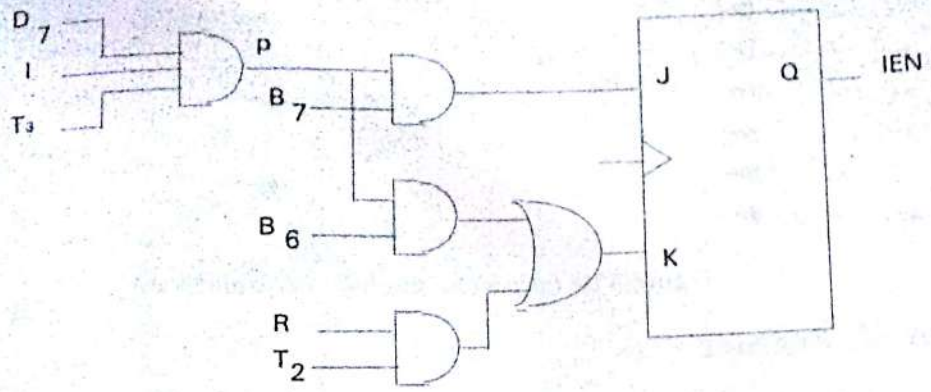


Figure 3.17: *Control Input for IEN*

Control of Common Bus

16-bit common bus is controlled by three selection inputs S2, S1 and S0. The decimal number associated with each component of the bus determines the equivalent binary number that must be applied to the selection inputs to select the registers. This is described by the truth table given below:

Table 3.7: *Encoder for Bus Selection Circuit*

Inputs							Outputs			Register selected for bus
x_1	x_2	x_3	x_4	x_5	x_6	x_7	S_2	S_1	S_0	
0	0	0	0	0	0	0	0	0	0	None
1	0	0	0	0	0	0	0	0	1	AR
0	1	0	0	0	0	0	0	1	0	PC
0	0	1	0	0	0	0	0	1	1	DR
0	0	0	1	0	0	0	1	0	0	AC
0	0	0	0	1	0	0	1	0	1	IR
0	0	0	0	0	1	0	1	1	0	TR
0	0	0	0	0	0	1	2	1	1	Memory

To find the logic that makes x1=1 we scan and extract all the statements that use AR as source.

D_4T_4: $PC \leftarrow AR$

D_5T_5: $PC \leftarrow AR$

=>

$x_1 = D_4T_4 + D_5T_5$

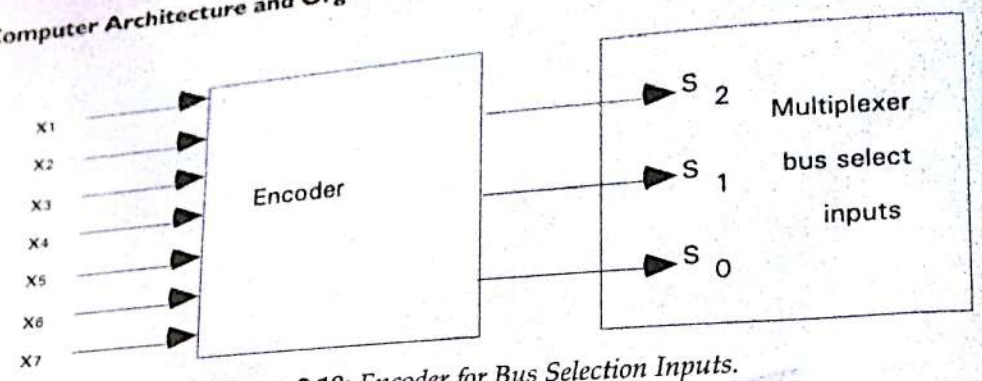


Figure 3.18: *Encoder for Bus Selection Inputs.*

Control of AC Register

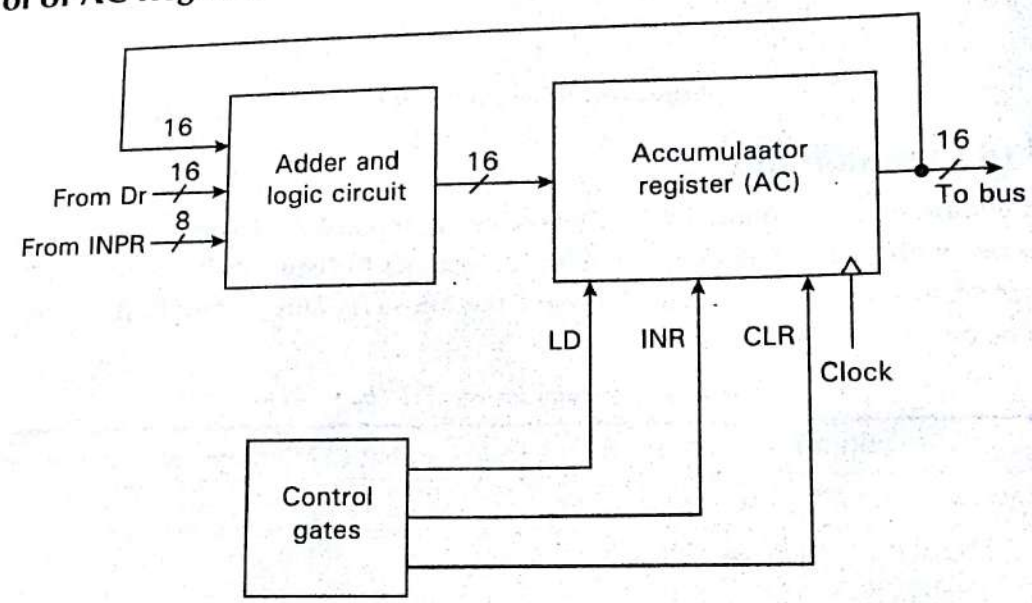


Figure 3.19: *Circuits Associated with AC.*

All the statements that change the content of AC

D_0T_5:	$AC \leftarrow AC \wedge DR$	AND with DR
D_1T_5:	$AC \leftarrow AC + DR$	Add with DR
D_2T_5:	$AC \leftarrow DR$	Transfer from DR
pB_{11}:	$AC(0\text{-}7) \leftarrow INPR$	Transfer from INPR
rB_9:	$AC \leftarrow AC'$	Complement
rB_7 :	$AC \leftarrow shr\ AC,\ AC(15) \leftarrow E$	Shift right
rB_6 :	$AC \leftarrow shl\ AC,\ AC(0) \leftarrow E$	Shift left
rB_{11} :	$AC \leftarrow 0$	Clear
rB_5 :	$AC \leftarrow AC + 1$	Increment

=>

$LD(AC) = D_0T_5 + D_1T_5 + D_2T_5 + pB_{11} + rB_9 + rB_7 + rB_6$

$CLR(AC) = rB_{11}$

$INR(AC) = rB_5$

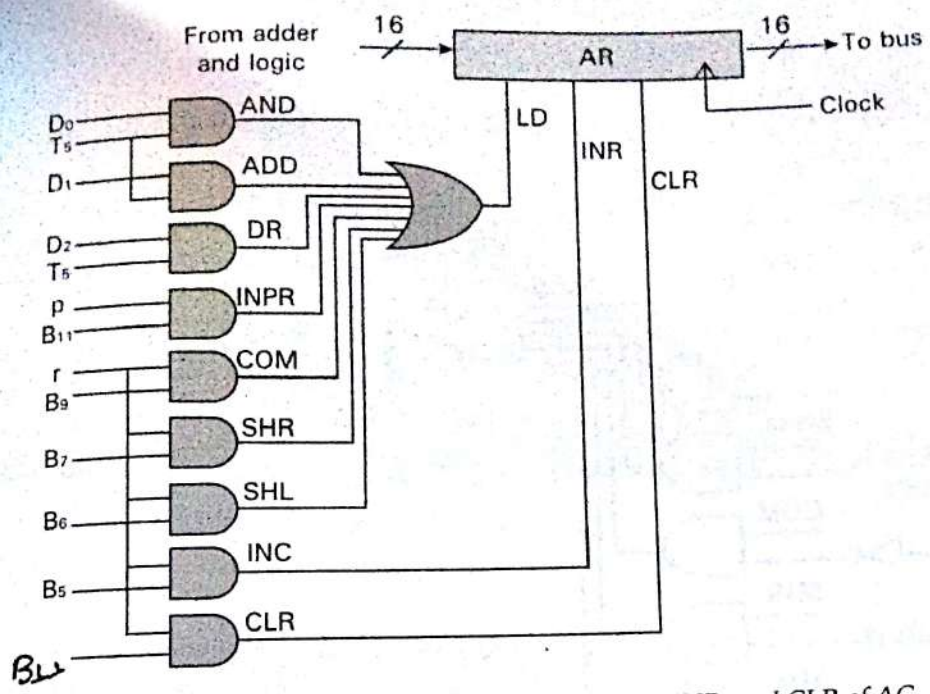


Figure 3.20: *Gate structure for controlling the LD, INR, and CLR of AC.*

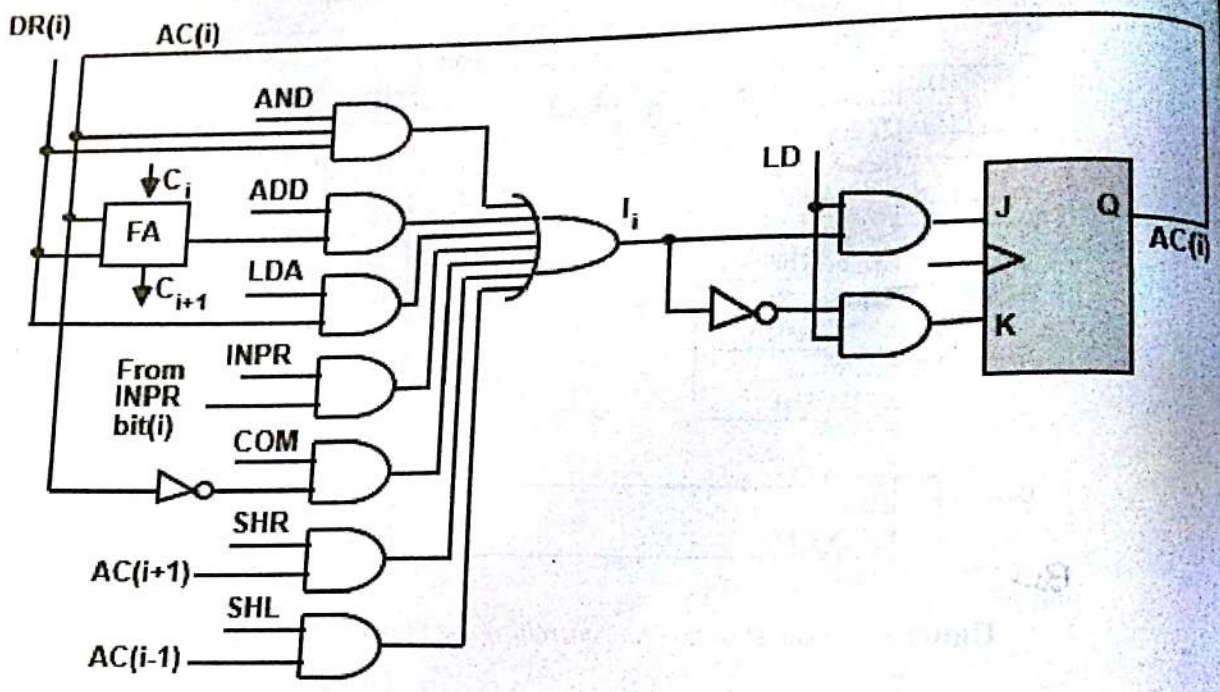
3.10 Adder and Logic Circuit

The adder and logic circuit can be sub-divided into 16 stages, with each stage corresponding to one bit of AC. The load (LD) input is connected to the inputs of the AND gates. Figure 3.21 shows one such ac register stage (with the OR gates removed). The input is labeled I_i and the output AC (i). When the LD input is enabled, the 16 inputs I_i for i = 0, 1, 2, ..., 15 are transferred to ac.

One stage of the adder and logic circuit consists of seven AND gates, one OR gate and full-adder (FA), as shown in Fig. 3.21 is connected to the output marked ADD in Fig. 3.21.

The AND operation is achieved by ANDing AC(i) with the corresponding bit in the data register DR (i). The ADD operation is obtained using a binary adder.

Figure 3.21: *One stage of adder and logic circuit*



Chapter 4

PROGRAMMING THE BASIC COMPUTER

Objective

This chapter utilizes the twenty five instruction of the basic computer to illustrate techniques used in assembly language programming. Programming examples are presented for number of data processing tasks. The relationship between binary program and symbolic code is explained by examples. The basic operations of an assembler are presented to show the translation from symbolic code to an equivalent binary program.

OUTLINE

4.1 Introduction
4.2 Machine Language
4.3 Assembly Language
4.4 The Assembler
4.5 Program Loops
4.6 Programming Arithmetic and Logic Operations
4.7 Subordinates
4.8 Input-Output Programming

4.1 INTRODUCTION

A computer system as it was mentioned before in chapter 1, it is subdivided into two functional parts:

Hardware, which consists of all physical components and all associated device or equipment. Parts of the hardware have been explained in chapter five concerning the basic computer. More details will be discussed in the following chapters. **Software,** which refers to the programs that are written for the computer.

The system software of a computer is divided into two main parts, the operating system that consists of the programs included in a system software package and the application programs, in which those programs are written by the user for solving particular problems.

A program written by a user maybe either dependent or independent of the computer that runs this program. For example, a program written in standard Pascal is machine independent because most computers provide a translator program that translates the standard Pascal program to the binary code of the computer available in the particular installation. In the other hand the translator program is machine dependent since it must translate the Pascal program to the binary code recognized by the hardware of the particular computer used.

Generally, one can be familiar with various aspects of computer software without being familiar with details of how the computer hardware operates. In the same way, it is possible to design parts of the hardware without knowledge of its software capabilities. However, the people concerned with computer architecture, they should have knowledge of both hardware and software because these two branches influence each other.

In spite of that, the computer executes only the programs that are written in binary form, there are variety types of programming languages that a programmer can write his program. In these cases there must be a translator to translate these programs to the binary form before the computer can execute them.

Table 4.1: *computer Instruction*

Symbol	Hexadecimal code	Description
AND	0 or 8	AND M to AC
ADD	1 or 9	Add M to AC, carry to E
LDA	2 or A	Load AC in M
STA	3 or B	Store AC in M
BUN	4 or C	Branch unconditionally to m
BSA	5 or D	Save return address in m and branch to m + 1
ISZ	6 or 6	Increment M and skip if zero
CLA	7800	Clear AC
CLE	7400	Clear E
CMA	7200	complement AC
CME	7100	Complement E
CIR	7080	Circulate right E and AC
CIL	7040	Circulate left E and AC
INC	7020	Increment AAC

SPA	7010	Skip if AC is positive
SNA	1008	Skip if AC is negative
SZA	7004	Skip if AC is zero
SZE	7002	Skip if E is zero
HLT	7001	Halt computer
INP	F800	Input information and clear flag
OUT	F400	Output information and clear flag
SKI	F200	Skip if input flag is on
SKO	F100	Skip if output flag is on
ION	F080	Turn interrupt on
IOF	F040	Turn interrupt off

Programs written for computer may be in one of the following forms: -

1. Binary code form

This is the machine language form, which is defined as a collection of all the fundamental instructions that the machine can execute, expressed together with the operands as a pattern of 1's and 0's.

2. Octal or hexadecimal code form

This achieved by translating of the binary code to the equivalent octal or hexadecimal representation.

3. Symbolic code form (assembly language)

In this form, the alphanumeric equivalent of the machine language is used. Alphanumeric mnemonics are used as an aid to the programmer, instead of the1's and 0's that the machine interpret. Each symbolic instruction can be translated into one binary coded instruction. The translator that translates the program from assembly language to machine language called Assembler.

4. High-level programming languages

These programming languages are closer to the programmer language and far from the machine language. These languages reflects the procedures used in solution of problems rather than be concerned with the computer hardware behavior. Examples of these languages are C, Pascal, Fortran…etc.

Programs in these programming languages are written in a sequence of statements, each statement must be translated into a sequence of binary instructions before the program can be executed in a computer.

The program that translates a high-level language program to binary called a Compiler.

4.2 MACHINE LANGUAGE

A program in machine language is a sequence of instructions and operands in binary that list the exact representation of instructions as they appear in computer memory.

1. Table 4.2 shows a binary program for adding the two numbers. Here the program is stored in the memory from location 0 to 6.Looking in this type of program, there is a difficulty to understand what is to be achieved when executed. However, the computer hardware recognizes this type of instruction code.

Table 4.2: *Binary program to add two numbers*

Location	Instruction Code
0000 0000 0000	0010 0000 0000 0100
0000 0000 0001	0001 0000 0000 0101
0000 0000 0010	0011 0000 0000 0110
0000 0000 0011	0111 0000 0000 0001
0000 0000 0100	0000 0000 0001 0011
0000 0000 0101	1111 1111 1111 1110

2. Table 4.3 shows a hexadecimal program for adding the two numbers. The hexadecimal representation of the program is more convenient to use than the binary form for the programmer; however, one must convert each hexadecimal digit to its equivalent 4-bit number when the program is entered into the computer.

Table 4.3: *Hexadecimal program to add two numbers*

Hexadecimal Program	
location	Instruction
000	2004
001	1005
002	3006
003	7001
004	0013
005	FFFE
006	0000

3. Table 4.4 shows a program with symbolic operation codes for adding the two numbers instead of their binary or hexadecimal equivalent. Here the address parts of memory-reference instructions, as well as operands, remain in their hexadecimal value. A column of comments is included for explaining the function of each instruction. Symbolic programs are easier and preferable to handle. These symbols can be converted to their binary code equivalent to produce the binary program.

Table 4.4: *Program with symbolic operation codes*

location	Instruction	Comments
000	LDA 004	Loud first operand Into AC
001	LDD 005	Add second operand to AC
002	STA006	Store sum in location 006
003	HLT	Halt computer
004	0013	The first operand
005	FFFE	The second operand
006	0000	Store the sum here

Note: *Negative numbers must be converted to binary in the signed 2's complement representation in the above three programming forms.*

4. Table 4.5 shows an assembly language program for adding the above two numbers. The symbols ORG, DEC, and END are called Pseudo Instructions and not a machine instructions. The purpose of ORG Pseudo instruction is to specify an origin, i.e. indicates the memory location of the first instruction of the program is below it. The purpose of DEC Pseudo instruction is to specify a decimal operand. The purpose of END Pseudo instruction is to indicate the end of the program. The three lines having symbolic addresses A, B, and C, their value is specified by they being present as a label in the first column.

Table 4.5: *Assembly language program to add two numbers*

Location	Instruction	Comments
	ORG0	/Origin of program is location 0
	LDA A	/Load operand from location A
	ADD B	/Add operand from location B
	STA C	/Store sum in location C
	HLT	/Halt computer
A,	DEC13	/First decimal operand
B,	DEC -2	/Second decimal operand
C,	DEC0	/Sum stored in location C
	END	End of symbolic program

5. Program given below shows an equivalent FORTRAN program for adding the two numbers. The FORTRAN program is translated into the binary values listed in the program of table 4.2, by what is called a Compiler Program.

Exhibit: *Fortran Program to add two numbers*

```
INTEGER A, B, C
DATA A, 13 B, -2
C = A + B
END
```

4.3 Assembly Language

Every computer has its own particular assembly language. The rules for writing assembly language programs are documented and published in manuals which are usually available from the computer manufacturer.

Rules of an assembly language

The line of an assembly language program is divided to three columns called fields as follows: -

- **The Label Field.** This field may be empty or it may specify a symbolic address. A symbolic address has the following characteristics: -
  - It consists up to three alphanumeric characters, the first character must be a letter, and the other two if they exist may be letters or numerals.
  - The programmer chooses the symbolar bitrarily.
  - A symbolic address is terminated by a comma.
- **The Instruction Field.** This field specifies a machine instruction or a pseudo instruction. The instruction field may specify one of the following:-
  - A memory reference instruction (MRI).
  - A register reference Instruction (RRI).
  - Input-output instruction (IOI).
  - A pseudo instruction with or without operand.
- A memory reference instruction (MRI) occupies two or three symbols separated by spaces:-
  - The first part must be a three letter symbol defining an MRI.
  - The second part is the symbolic address, which specifies the memory location of an operand.
  - The third part is the letter I, this may or may not present. If it's present, it denotes indirect address, while if I is missing, it denotes direct addressing.

Pseudo instruction as mentioned before is not a machine instruction but an instruction which gives information to the assembler about some phase of the translation. There are four pseudo instructions as shown in table 4.6.

Table 4.6: *Definition of pseudo instructions*

Symbol	Information for the Assembler
ORG N	N indicates the memory location for the instruction or operand listed in the following line.
END	Indicates the end of the program.
DEC N	N indicates the decimal signed number to be converted to binary.
HEX N	N indicates the hexadecimal signed number to be converted to binary.

- **The Comment Field.** This field may be empty or it may include a comment. Comments must be preceded by a slash for the assembler to recognize the beginning of a comment field. Comments are inserted for explanation process only and are neglected during the binary translation process.

Assignment

Translate the following assembly language program to subtract two numbers to its equivalent binary from.

Table 4.7: *Assembly language program to subtract two numbers*

ORG 100	/Origin of program is location 100
LDA SUB	/Load subtrahend to AC
CMA	/Complement AC
INC	/Increment AC
ADD MIN	/Add minuend to AC
STA DIF	/store difference
HLT	/Halt computer

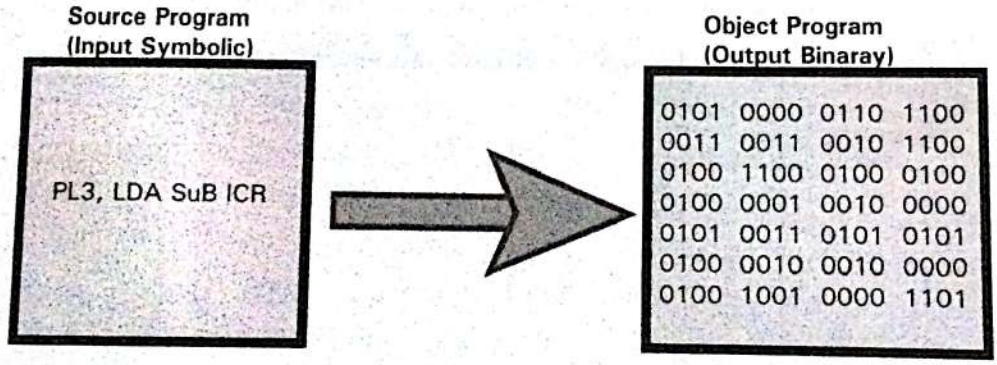
MIN.	DEC 83	/Minuend
SUB.	DEC - 23	/subtrahend
DIF.	HEX 0	/Difference stored here
	END	/End of symbolic program

4.4 THE ASSEMBLER AND THE COMPILER PROGRAMS

The Assembler Program is a program that translates a symbolic language program called the source program into binary program called the object program. An assembler is a program that operates on character strings and produces an equivalent binary interpretation. A Compiler Program is a system program that translates a program written in a high level programming language such as Pascal to a machine language program. A compiler may use an assembly language as an intermediate step in the translation or may translate the program directly to binary.

The Assembler

A program that accepts a symbolic language program and produces its binary machine language equivalent **OR** A program that operates on character strings and produces an equivalent binary interpretation.



Representation of Symbolic Program in Memory

Prior to starting the assembly process, the symbolic program must be stored in memory. Each character is represented by an 8-bit code. The high-order bit is always 0 and the other seven bits are as specified by ASCII. A line of code is stored in consecutive memory location with two characters in each location (since 16-bit memory word used). Label is terminated with comma and operand and addresses symbol terminated with space. End of line recognized by their CR (Carriage Return) code.

Table 4.8: *Hexadecimal character code*

Character	Code	Character	Code	Character	Code
A	41	Q	51	6	36
B	42	R	52	7	37
C	43	S	53	8	38
D	44	T	54	9	39
E	45	U	55	space	20
F	46	V	56	(	28
G	47	W	57	)	29
H	48	X	58	*	2a
I	49	Y	59	+	2B
J	4A	Z	5A	,	2C
K	4B	0	30	-	2D
L	4C	1	31	.	2E
M	4D	2	32	/	2F
N	4E	3	33	=	3D
O	4F	4	34	CR	0D (carriage return)
P	50	5	35		

ASCII value of A = $(65)_{10} = (41)_{16}$

The input for the assembler program is the user's symbolic language program in ASCII

Table 4.9: *Computer representation of the line of code*

Memory word	Symbol	Hexadecimal code	Binary representation
1	P L	50 4C	0101 0000 0100 1100
2	3 ,	33 2C	0011 0011 0010 1100
3	L D	4C 44	0100 1100 0100 0100
4	A	41 20	0100 0001 0010 0000
5	S U	53 55	0101 0011 0101 0101
6	B	42 20	0100 0010 0010 0000
7	I CR	49 0D	0100 1001 0000 1101

(/=2F): If the line of the code has comment, the assembler neglects all characters in the comment field and keeps checking for a CR code.

Two Pass Assembler

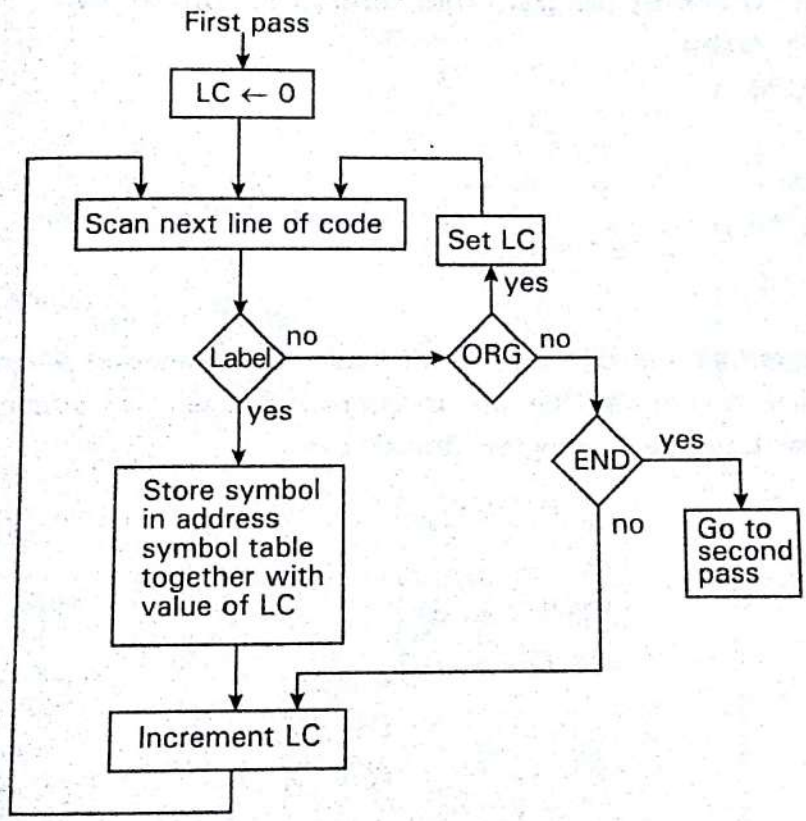
A two pass assembler scans the entire symbolic program twice

First Pass: it reads the entire source file, looking only for label definitions. All labels are collected, assigned values, and placed in the symbol table in this pass. No instructions are assembled and, at the end of the pass, the symbol table should contain all the labels defined in the program.

Table 4.10: *Address symbol table for program*

Memory word	Symbol or (LC)*	Hexadecimal code		Binary representation			
1	M I	4D	49	0100	1101	0100	1001
2	N ,	4E	2C	0100	1110	0010	1100
3	(LC)	01	06	0000	0001	0000	0110
4	S U	53	55	0101	0011	0101	0101
5	B ,	42	2C	0100	0010	0010	1100
6	(LC)	01	07	0000	0001	0000	0111
7	D I	44	49	0100	0100	0100	1001
8	F ,	46	2C	0100	0110	0010	1100
9	(LC)	01	08	0000	0001	0000	1000

Figure 4.1: *Flowchart for first pass of assembler*



Second Pass: the instructions are again read and are assembled, using the symbol table. Table lookup procedure is a search of table entries to determine whether a specific item matches one of the item stored in the tale. The assembler uses four tables:

- Pseudoinstruction table
- MRI (Memory Reference Instruction) table
- Non-MRI table (Register reference or Input-Output instruction)
- Address symbol table

Table 4.11: *Definition of Pseuddoinstructions*

Symbol	Information for the assembler
ORG N	Hexadecimal number N is the memory location for the instruction or operand listed in the following line
END	Denotes the end of symbolic program
DEC N	Signed decimal number N to be converted to binary
HEX N	Hexadecimal number N to be converted to binary

4.5 PROGRAM LOOPS

A program loop is a sequence of instructions that are executed many times, each time with a different set of date. Program loops are specified in fortran by a DO statement.

Example of a FORTRAN program that forms the sum of 100

```
DIMENSION A(100)
INTEGER SUM, A
SUM=0
DO 3 J=1, 100
3 SUM=SUM+A(J)
```

Compiler

A system program that translates a program written in a high-level programming language is known a compiler. A compiler may use an assembly language as an intermediate step in the translation or may translate the program directly to binary.

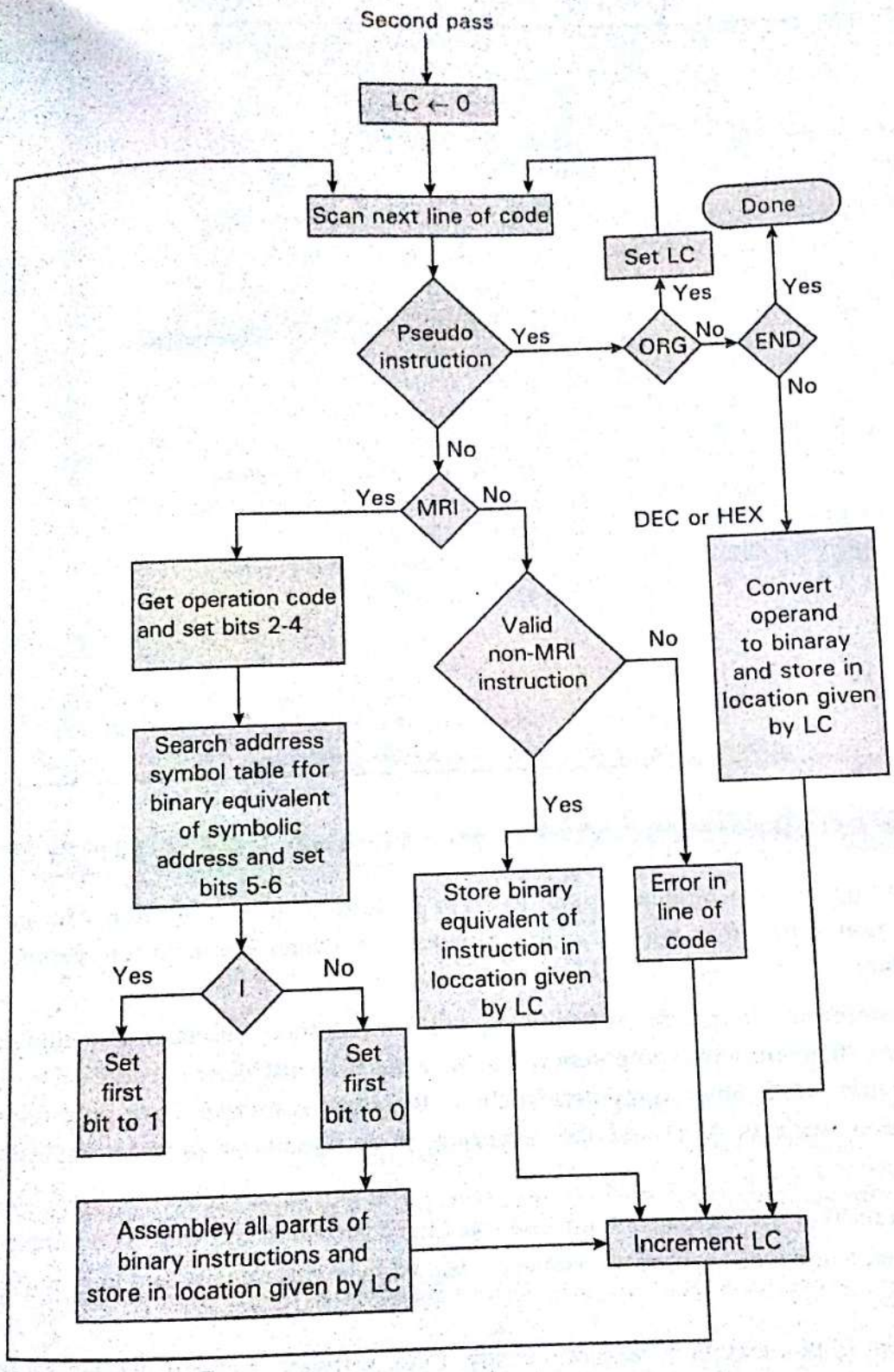


Figure 4.2: *Flowchart for second pass of assembler*

Table 4.12: *Symbolic Program to Add 100 numbers*

1		ORG	100	/Origin of program is HEX 100
2		LDA	ADS	/Load first address of operands
3		STA	PTR	/Sore in pointer
4		LDA	NBR	/Load minus 100
5		STA	CTR	/Store in counter
6		CLA		/Clear accumulator
7	LOP,	ADD	PTR I	/Add an operand to AC
8		ISZ	PTR	/Increment pointer
9		ISZ	CTR	/Increment counter
10		BUN	LOP	/Repeat loop again
11		STA	SUM	/Store sum
12		HLT		/Halt
13	ADS,	HEX	150	/First address of operands
14	PTR,	HEX	0	/This location reserved for a pointer
15	NBR,	DEC	100	/Constant to initialized counter
16	CTR,	HEX	0	/This location reserved for a counter
17	SUM,	HEX	0	/Sum of stored here
18		ORG	150	/Origin of operands is HEX 150
19		DEC	75	/First operand
.				
.				
.				
118		DEC 23		/Last operand
119		END		/End of symbolic program

4.6 PROGRAMMING ARITHMETIC AND LOGIC OPERATIONS

Depending on the computer, some of them perform a given operation with one machine instruction; others may require a large number of machine instructions to perform the same operation.

For example the four basic arithmetic operations (addition, subtraction, multiplication, and division), there are some computers which have machine instruction to add, subtract, multiply, and divide while other computers such as the basic computer have only one arithmetic instruction, such as ADD and the remaining three operations must be implemented by a programs.

Operations that are executed with one machine instruction are said to be implemented by hardware, while those operations that are executed by a program are said to be implemented by software.

Hardware implementation is more costly than software implementation because of the additional circuits needed to implement the operation, while software implementation has a drawback of long execution time results from more instructions needed to implement the operation.

In this section, we will demonstrate the software implementation for some of arithmetic and logic operation using the basic computer.

1. Multiplication Operation

To simplify the process, assume unsigned positive binary numbers and each number have no more than 8-bitsso the result of multiplication does not exceeds the word capacity of 16-bits.

The program for multiplying two numbers based on the procedure used to multiply numbers with paper and pencil. The following example with four significant explains this multiplication process.

Multiplicand	X =	00000111
Multiplier	Y =	00001010
		00000000
		00001110
		00000000
		00111000
		01000110

As shown in the above example, the multiplication process consists of checking the bits of the multiplier Y and adding the multiplicand X shifted left from one line to the next.

A memory location denoted by P is reserved to store intermediate sums, since the computer can add only two numbers at a time. The intermediate sums are called partial products since they hold a partial product until all numbers are added.

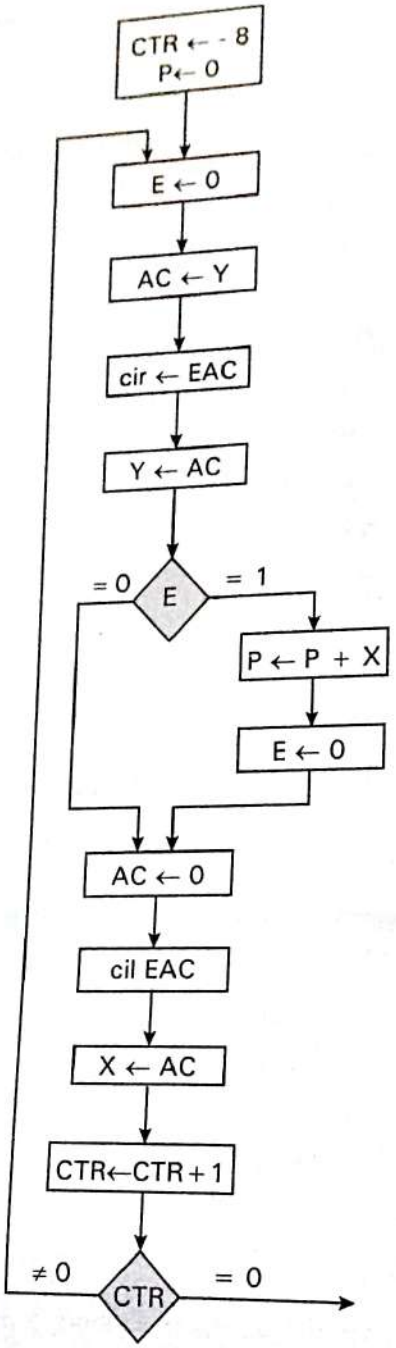
The following example shows how the result of multiplication is achieved in the memory location P for four significant digit of multiplier and multiplicand.

Initial value of P	0000 0000
P after of 1st pass	0000 0000
P after of 2nd pass	0000 1110
P after of 3rd pass	0000 1110
P after of 4th pass	0100 0110

Final Result

<u>Note.</u> The computer can use numbers with eight significant bits to produce a product of up to 16 bits.

The flowchart shown in figure 4.3 explains the step-by-step the multiplication operation.



X holds the multiplicand
Y holds the multiplier
P forms ythe product

Example with four significant digits

			P	
X =	0000	1111		
Y =	0000	1011		
	0000	1111	0000	0000
	0001	1110	0000	1111
	0000	1111	0010	1101
	0111	1000	0110	1101
	0000	1111	1010	0101

Figure 4.3: *Flowchart for multiplication program*

Initially, location X holds the multiplicand and location Y holds the multiplier. A counter CTR is set to -8 and location P is cleared to zero.

The following program lists the instructions for multiplying two unsigned numbers X = 07 & Y =0A.

Note: *The above program to be a general program for multiplying any two unsigned numbers; the initialization of the multiplicand and multiplier into locations X & Y respectively must be included. The initialization process also must include the initialization of the counter CTR to -8and P to zero. After*

generalizing the program according to the note above, the resultant program will demonstrate the software implementation of the multiplication operation.

Table 4.13: *Program to multiply two position numbers*

```
        ORG 100
LOP,    CLE         /Clear E
        LDA Y       /Load multiplier
        CIR         /Transfer multiplier bit to E
        STA y       /Store shifted multiplier
        SZE         /Check if bit is zero
        BUN ONE     /Bit is one; go to ONE
        BUN ZRO     /Bit is zero; go to ZRRO
ONE,    LDA X       /Load multiplicand
        ADD P       /Add to partial product
        STA P       /Store partial product
        CLE         /Clear E
ZRO,    LDA X       /Load multiplicand
        CIL         /Shift left
        STA X       /Store shifted multiplicand
        ISZ CTR     /Increment counter
        BUN LOP     /Counter not zero; repeat loop
        HLT         /Counter is zero; halt
CTR,    DEC -8      /This location serves as a counter
X,      HEX 000F    /Multiplicand stored here
Y,      HEX 000B    /Multiplier stored here
P,      HEX         /Product formed here
        END
```

2. Double-Precision Addition

A double precision number is a number that stored in two memory words of the memory.

To add two double precision numbers; first, every number is placed in two consecutive memory locations, AL & AH for the first number and BL & BH for the other; in which AL & BL holds the 16 least significant bits (b15 – b0) and BL & BH holds the 16 most significant bits (b31 - b16) of the two numbers.

The program for adding two numbers is listed below. First the two low-order portions are added and the carry transferred into E, the content of AC stored into the memory location CL (the 16 least significant bits of the sum). AC is cleared and the bit in E is circulated into the least significant bit of the accumulator AC(0). The two high-order portions are then added to the carry and the content of AC stored into the memory location CH (the 16 most significant bits of the sum).

Table 4.14: *Program to add two double precision numbers*

	LDA AL	/load A low
	ADD BL	/Load B low, carry in E
	STA CL	/Store in C low
	CLA	/Clear AC
	CIL	/Circulate to bring carry into AC (0)
	ADD AH	/Add A high and carry
	ADD BH	/Add B high
	STA CH	/Store in C high
	HLT	/Halt the computer
AL,	xxxx	/Location of the low significant part of the 1stnumber
AH,	xxxx	/Location of the high significant part of the 1stnumber
BL,	yyyy	/Location of the low significant part of the 2ndnumber
BH,	yyyy	/Location of the high significant part of the 2ndnumber
CL,	-	/Location of the low significant part of the result
CH,	-	/Location of the high significant part of the result

3. Logic Operations

In chapter six it is shown that the basic computer has three machine instructions that perform logic operations: AND, CMA, and CLA. Also LDA instruction can be considered as logic operation that transfers a logic operand into the AC. Any logic function can be implemented using the AND and complement operation CMA.

Example: Write a program to forms the OR logic operation of two logic 16-bit operands A & B given that $A \vee B = (A \cdot B)$.

The Program

	LDA A	/load the first operand A into AC (A)
	CMA	/Complement to get A complement
	STA TEM	/Store in temporary location TEM
	LDA B	/load the second operand B into AC(B)
	CMA	/Complement to get B complement
A,	AND TEM	/AND with the content of TEM ($A \wedge B$)
B,	CMA	/Complement the result to obtain A ($\overline{\overline{A}\overline{B}}$) = $A \vee B$
	xxxxxxx	/Location of the first operand
	x	/Location of the second operand

In a similar manner, the other logic operations can be implemented.

4. Shift Operations

The basic computer as it was shown before contains only the circular-shift instructions (CIR & CIL) which executes the circular-shift operations.

The logical and arithmetic shift operations can be programmed with a number of the computer instructions as follows:

a. Logical Shift operations

These shift operations requires zeroes to be loaded to the extreme positions. This is accomplished by clearing E and circulating the AC &E.

Examples

1. The logical shift-right operation can be implemented by the following two instructions:
   CLE CIR
2. The logical shift-left operation can be implemented by the following two instructions:
   CLE CIL

b. Arithmetic shift operations

These shift operations depends on how the negative numbers are represented. As mentioned before the basic computer uses the 2's complement representation for the negative numbers.

1. **Arithmetic right-shift:** The rules for arithmetic right-shift states that the sign bit in the leftmost position remain unchanged and the sign bit itself is shifted into the high-order bit position of the number.

   The program for the arithmetic right-shift requires setting E to the same value as the sign bit and circulate right as follows:

CLE	/Clear E to 0
SPA	/Skip if AC is positive; E remains0
CME	/AC is negative; set E to1
CIR	/Circulate E &AC

2. **Arithmetic left-shift:** The rules for arithmetic left-shift are that the added bit in the least significant position must be 0. This is accomplished by clearing E before circulate-left operation. Also the sign bit must not change during this shift. After the circulate instruction is executed, the sign bit moves into E, then it is necessary to compare the sign bit with the value of E after the operation. If the two bits are equal, the arithmetic shift has been correctly implemented. If they are not equal, an overflow occurs (i.e. the original number was too large).

   The simplified program for the arithmetic left-shift is as follows:

CLE	/Clear E to 0
CIL	/Circulate E &AC

4.7 SUBROUTINES

A Subroutine can be defined as a set of instructions that can be used in a program many times.

A Subroutine has general characteristics such as:

- It consists of a self -contained sequence of instructions that carries out a given task.

- A branch can be made to the subroutine from any part of the main program.
- All computers provide special instructions commonly called subroutine call to facilitate subroutine entry and return to the main program.
- The procedure for branching to a subroutine and returning to the main program is referred to as a subroutine linkage.
- The last instruction of the subroutine performs an operation commonly called subroutine return.

For basic computer the instruction BSA (Branch and Save return Address) is the link instruction between the main program and the subroutine.

Table 4.15 shows a program that demonstrates the use of subroutines. This program contains a subroutine that shifts the content of the accumulator four times to the left which is useful operation for processing binary-coded decimal numbers or alphanumerical characters.

Table 4.15: *Program to demonstrate the use of subroutines*

Location	Label	Instruction	Comments
		ORG 200	/Main program
200		LDA X	/Load X
201		BSA SH4	/Branch To subroutine
202		STA X	/Store shifted number
203		LDA Y	/Load Y
204		BSA SH4	/Branch to subroutine a second time
205		STA Y	/Store shifted number
206		HLT	/Halt the computer
207	X,	HEX1234	/Subroutine to shift left 4 times
208	Y	HEX 4321	/Store return address here
209	SH4	HEX 0	/Circulate left once
20A		CIL	/Circulate left once
20B		CIL	/Circulate left once
20C		CIL	/Circulate left once
20D		CIL	/Set AC(3-0) to zero
20E		AND MSK	/Return to main program
20F		BUN SH4I	/Mask operand
210	MSK,	HEXFFF0	
		END	

How the program is executed

1. The first number (HEX 1234) contained at location X (HEX 207) is loaded to the accumulator AC by the instruction LDAX.
2. When the next instruction BSA SH4 is executed, the control unit stores the return address (HEX 202) into the location defined by the symbolic address SH4 (location HEX 209), also transfers the value of SH4+1 (i.e. HEX 20A) into the program counter PC.
3. After execution of the instruction BSA SH4, the subroutine is executed starting from location 20A (since this is the content of PC in the next fetch cycle).

a. The four shift instructions (CIL) circulate the content of AC four times to the left.

b. In order to accomplish a logical shift instruction, the four least significant bits must be zero. This is achieved by maskingFFF0 with the content of AC using the AND operation (the instruction at location HEX20E).

c. The last instruction BUN SH4 I (indirect branch unconditional instruction) returns the computer to the main program (i.e. to location HEX202).

4. The main program continues by storing the shifted number into location X.

5. A new number (HEX4321) from location Y (HEX208) is then loaded into the AC and another branch is made to the subroutine. The same procedure will be repeated, but in this case the location SH4 will contain the return address HEX 205 since this is now the location of the next instruction after BSA. After the subroutine is executed the subroutine returns to the main program at location 205.

6. When the execution of the program is completed the contents of locations 207 & 208 are HEX 2340 & HEX 3210 respectively.

Table 4.16: *Program to demonstrate parameter linkage*

Location			
		ORG 200	
200		LDA X	/Load first operand into AC
201		BSA OR	/Branch to subroutine or
202		HEX 3AF6	/Second operand stored here
203		STA Y	/Subroutine returns here
204		HLT	
205	X,	HEX 7B95	/First operand stored here
206	Y,	HEX 0	/Result stored here
207	OR,	HEX 0	/Subroutine OR
208		CMA	/Complement first operand
209		STA TMP	/Store in temporary location
20A		LDA OR I	/Load second operand
20B		CMA	/Complement second operand
20C		AND TMP	/AND complemented first operand
20D		CMA	/Complement again to get OR
20E		ISZ OR	/Increment return address
20F		BUN OR I	/Return to main program
210	TMP,	HEX 0	/Temporary storage
		END	

Table 4.17: *Subroutine to move to block of data*

		/Main program
	BSA MVE	/Branch to subroutine
	HEX 100	/First address of source data
	HEX 200	/First address of destination data
	DEC -16	/Number of items to move
	HLT	
MVE,	HEX 0	/Subroutine MVE
	LDA MVE I	/Bring address of source

	STA PT1	/Store in first pointer
	ISZ MVE	/Increment return address
	LDA MVE I	/Bring address of destination
	STA PT2	/Store in second pointer
	ISZ MVE	/Increment return address
	LDA MVE I	/Bring number of items
	STA CTR	/Store in counter
	ISZ MVE	/Increment return address
LOP,	LDA PT1 I	/Load source item
	STA PT1 I	/Store in destination
	ISZ PT1	/Increment source pointer
	ISZ PT2	/Increment destination pointer
	ISZ CTR	/Increment counter
	BUN LOP	/Repeat 16 times
	BUN MVE I	/Return to main program
PT1,	-	
PT2,	-	
CTR,	-	

4.8 INPUT-OUTPUT PROGRAMMING

The character is stored in the computer as an 8-bit code. The character enters the computer by an INP instruction and transferred to the output device using an OUT instruction. The following set of instructions, input a character and stores it in memory.

Table 4.18: *Program to input and output one character*

CIF,	SKI	/check input flag
	BUNCIF	/flag = 0, branch to check again
	INP	/flag = 1, input character
	OUT	/print character
	STA CHR	/store character
CHR,	HLT	
	----	/store character here

(a) Input a character

	LDA	/load character into AC
COF,	CHRSKO	/check output flag
	BUNCOF	/flag = 0, branch to check again
	OUT	/flag = 1, output character
	HLT	
CHR,	HEX0057	/character is W

(b) Output the character

The program starts with the instruction SKI which checks the input flag to see if a character is available to transfer. If the input flag is 1, the next instruction BUN is skipped. The IN instruction transfers the binary-coded character into AC (0-7). By the OUT instruction, the character is printed by the output device. If the input flag is 0, the next instruction BUN is executed and a branch to return and check the input flag again. The two instructions SKI & BUN will be executed many times before a character is transferred into the accumulator, because the input device is much slower than the computer .The following set of instructions, print a character initially stored in memory.

The program starts with the instruction LDA to load the binary-coded character into the AC from the memory location CHR. The instruction SKO checks the output flag, if its 1, the character is transferred from AC to the printer. If the output flag is 0, the computer remains in a two instruction loop (SKO & BUN) checking the flag bit.

Character Manipulation

To pack two characters in one memory word for the purpose of character manipulation, the subroutine named IN2 that inputs two characters and packs the min to the AC is shown below.

Table 4.19: *Subroutine to input and place two characters*

IN2, FST,	-	/Subroutine entry
	SKI	/check input flag
	BUN FST	/flag = 0, branch to check again
	INP	/flag = 1, input character
	OUT	/output character
SCD,	BSA SH4	/Shift left four times
	BSA SH4	/Shift left four times
	SKI	/check input flag
	BUN	/flag = 0, branch to check again
	SCDINP	/flag = 1, input character
	OUT	/output character
	BUN IN2I	/Return

It is clear that this subroutine calls twice the subroutine SH4 to shift the accumulator left eight times.

Table 4.20: Program to share input character in a buffer

	LDA ADS	/Load first address to buffer
	STA PTR	/Initialize pointer
LOP,	BSA IN2	/Go to subroutine
	STA PTR I	/Store double character word in buffer
	ISZ PTR	/Increment pointer
	BUN LOP	/Branch to input more character
	HLT	
ADS,	HEX 500	/First address of buffer
PTR,	HEX 0	/Location for pointer

Program Interrupt

The interrupt facility is useful when two or more programs reside in the memory at the same time.

Even though two or more programs may reside in the computer memory, only one program can be executed at a given time and this program is referred to as the running program. The other programs are usually waiting for input or output data.

The function of the interrupt facility is to take care of the data transfer of one (or more) program while another program is currently being executed. The running program must include an ION instruction to turn the interrupt on. If the interrupt facility is not used, the program must include an IOF instruction to turn it off.

The interrupt facility allows the running program to proceed until the input or output device sets its ready flag.

Whenever a flag is set to 1, the computer completes the execution of the instruction in progress and then acknowledges the interrupt. The result of this action is that the return address is stored in location 0. The instruction in location 1 is then performed; this initiates a service routine for the input or output transfer.

Table 4.21: *Program to compare two words*

	LDA WD1	/Load first word
	CMA	
	INC	/Form 2's complement
	ADD WD2	/Add second word
	SZA	/Skip if AC is zero
	BUN UEQ	/Branch to "unequal" routine
	BUN EQL	/Branch to "equal" routine
WD1,	–	
WD2,	–	

The service routine can be stored anywhere in memory provided a branch to the start of the routine is stored in location 1.

The service routine must have instructions to perform the following tasks:-

1. Save the contents of processor registers.
2. Check which flag is set.
3. Service the device whose flag is set.
4. Restore contents of processor registers.
5. Turn the interrupt facility on.
6. Return to the running program.

The following program listed in table 4.23 explains an interrupt services.

1. Location 0 is reserved for the return address.
2. Location 1 has a branch instruction to the beginning of the service routine SRV.
3. The ION instruction found in the main program turns the interruption.

4. It is supposed an interrupt occurs while the computer is executing the instruction in location103.
5. The interrupt cycle stores the address of the following instruction (HEX 104) in location 0 and branches to location1.
6. The branch instruction in location 1 branches to the service routine SRV.
7. The service routine performs the six tasks listed above.

Table 4.22: *Program to service n interrupt*

Location	Label	Instruction	Comment
0	ZRO,	-	/return address stored here
1		BUN	/branch to service routine
100			/portion of running program
101		SRVCLA	/turn on interrupt facility
102		ION	
103		LDA X	/interrupt occur here
104		ADD	/program returns here after interrupt
.		YSTA Z	
.		.	
200	SRV,	.	/interrupt service here
			/store content of AC
		STA SAC	/move E into AC (15)
		CIR	/store content of E
		STA SE	/check input flag
		SKI	/flag is off, check next flag
		BUN	/flag is on, input character
		NXTINP	/print character
	NXT,	OUT	/store it in input buffer
		STA PT1 I	/increment input pointer
		ISZ PT1	/check output flag
		SKO	/flag is off, exit
		BUN EXT	/load character from output buffer
	EXT,	LDA PT2 I	/output character
		OUT	/increment output pointer
		ISZ PT2	/restore value of AC (15)
		LDASE	/shift it to E
		CIL	/restore content of AC
		LDA SAC	/turn interrupt on
		ION	/return to running program
		BUNZROI	/AC is stored here
	SAC,	-	/E is stored here
	SE,	-	/pointer of input buffer
	PT1,	-	/pointer of output buffer
	PT2,	-	

A typical computer may have many more input and output devices connected to the interrupt facility.

Problems

1. The following program is stored in the memory unit of the basic computer. Show the contents of the registers AC, PC, and IR in hexadecimal, after each instruction is executed. All numbers listed in the program are in hexadecimal.

Location	Instruction
010	CLA
011	ADD 016
012	BUN 014
013	HLT
014	AND 017
015	BUN 013
016	C1A5
017	93C6

2. For the following program:
   a. Explain in words what the following program accomplished when it is executed.
   b. What is the value of location CTR when the computer halts?
   c. List the hexadecimal code of the translated program.

	ORG 100 CLE CLA STA
ROT,	CTR LDA WRD SZA BUN ROT BUN STP
	CIL SZE BUN AGN BUN ROT CLE
AGN,	ISZ CTR
	SZA
	BUN
STP,	ROT
CTR,	HLT
WRD,	HEX 0
	HEX
	62C1
	END

3. The following program is in hexadecimal code. The computer executes the instructions starting from address 100. What are the content of AC and the memory word at address 103 when the computer halts?

Location	Instruction
100	5103
101	7200
102	7001
103	0000
104	7800
105	7020
106	C103

4. Write an assembly program to multiply two positive numbers by a repeated addition method. For example, to multiply 5 ×4, the program evaluates the product by adding 5 four times,5+5+5+5.

5. Write an assembly program to multiply two unsigned positive numbers, each with 16 significant bits, to produce an unsigned double-precision product.

6. Write an assembly program to subtract two double-precision numbers.
7. Write an assembly program that evaluates the logic exclusive-OR of two logic operands.
8. Write a subroutine to circulate E and AC four times to the right. If AC contains hexadecimal 079C and E = 1, what are the contents of AC and E after the subroutine is executed.
9. Write an assembly program to accept input characters. Pack two characters in one word and store them in consecutive locations in a memory buffer. The first address of the buffer is (400)16. The size of the buffer is (512)10 words. If the buffer overflows, the computer should halt.
10. Translate the service routine SRV from table 7.8 to its equivalent hexadecimal code. Assume that the routine is stored staring from location 200.
11. The following program is stored in the memory unit of the basic computer. Show the contents of the registers AC, PC, and IR in hexadecimal, after each instruction is executed. All numbers listed in the program are in hexadecimal.

Location	Instruction
010	CLA
011	ADD 016
012	BUN 014
013	HLT
014	AND 017
015	BUN 013
016	C1A5
017	93C6

Solution

Location	Instruction	Contents of Registers AC, PC and IR after the Execution of the Instruction		
		AC	PC	IR
010	CLA	0000	011	7800
011	ADD 016	C1A5	012	1016
012	BUN 014	C1A5	014	4014
013	HLT	8184	013	7001
014	AND 017	8184	015	0017
015	BUN 013	8184	013	4013
016	C1A5			
017	93C6			

Note: The IR column is taken from the hexadecimal presentation of the given instruction (see table 6.2 in chapter6).

12. The PC column calculated due to the fetch phase PC← PC + 1 as in case of CLA, ADD, and AND instructions or due to the fetch phase with the execution result of the instruction as in the case of BUN instruction as follows (taking the example of BUN014):
    a. In fetch phase AR← IR (0-11), i.e. AR = 014.
    b. After instruction execution PC← AR, i.e. PC =014.
13. For the AC, its contents are changed according to the instruction function.

    For the following program:
    a. Explain in words what the following program accomplished when it is executed.
    b. What is the value of location CTR when the computer halts?
    c. List the hexadecimal code of the translated program.

	ORG 100
	CLE
	CLA
	STA CTR
	LDA WRD
	SZA
	BUN ROT
ROT,	BUN STP
	CIL
	SZE
AGN,	BUN AGN
	BUN ROT
	CLE
STP,	ISZ CTR
CRT,	SZA
WRD,	BUN ROT
	HLT
	HEX 0
	HEX 62C1
	END

Solution

a. The program counts the number of 1's in the number stored in location WORD.

b. The counter CTR = 6, since the number (62C1) contains 6one's.

Assembly Language Program		Corresponding Hexadecimal Program	
Location	**Instruction**	**Location**	**Content**
	ORG100	100	7400
	CLE	101	7800
	CLA	102	3110
	STA CTR	103	2111
	LDA WRD	104	7004
	SZA	105	4107
	BUN ROT	106	410F
ROT,	BUN STP	107	7040
	CIL	108	7002
	SZE	109	410B
	BUN AGN	10A	4107
AGN,	BUN ROT	10B	7400
	CLE	10C	6110
	ISZ CTR	10D	7004

	SZA	10E	4107
STP,	BUN ROT	10F	7001
CTR,	HLT	110	0000
WRD,	HEX 0	111	62C1
	HEX 62C1		
	END		

14. The following program is in hexadecimal code. The computer executes the instructions starting from address 100. What are the content of AC and the memory word at address 103 when the computer halts?

Location	Instruction
100	5103
101	7200
102	7001
103	0000
104	7800
105	7020
106	C103

Solution

To understand the function of the program, we have to translate the binary program to an assembly program.

Hexadecimal Program		Assembly Language Program	
Location	Instruction	Location	Instruction
			ORG 100
100	5103	100	BSA 103
101	7200	101	CMA
102	7001	102	HLT
103	0000	103	HEX 0000
104	7800	104	CLA
105	7020	105	INC

The contents of:

1. The memory location H 103 =0101
2. The content of the accumulator AC = HE X FFFE

Explanation of how the program is executed

1. The execution of instruction BSA 103 do the following:
   a. Stores the address of the next instruction in sequence (HEX 101) which is available in PC into a memory location specified by the effective address (Location HEX 103 in memory) i.e. memory location 103 contains HEX0101.
   b. The effective address plus 1 is then transferred to PC to serve as the address of the first instruction in the subroutine i.e. PC = HE X 104.

2. Executing CLA instruction (Clear Accumulator) i.e. AC =0000.
3. Executing INC instruction (Increment Accumulator) i.e. AC =0001.
4. Executing BUN instruction (Indirect Branch Unconditional) when this instruction is executed, control goes to the indirect phase to read the effective address at location 103, where it finds the previously saved address 101.
5. The instruction at location 101 is executed (i.e. CMA), then AC = FFFE.
6. Then the HLT is executed and the program halts.
7. Then location 103 in the memory will contain 0101, and AC=FFFE

15. Write an assembly program to multiply two positive numbers by a repeated addition method. For example, to multiply 5 × 4, the program evaluates the product by adding 5 four times,5+5+5+5.

Solution

Assembly Prog.			Hexadecimal Prog.	
Loc.	Instruction	Comment	Loc.	Instruction
	ORG 100			
	LDA MUL	/Load multiplier in location MUL in AC	100	210A
	CMA	/Complement AC	101	7200
	INC	/Increment AC	102	7020
	STA MUL	/Store AC in location MUL	103	310A
	CLA	/Clear AC	104	7800
LOP,	ADD MUC	/Add multiplicand MUC to AC	105	110B
	ISZMUL	/Increment the counter MUL	106	610A
	BUN LOP	/Counter not zero; repeat loop	107	4105
	STA PRO	/Store the result in location PRO	108	310C
	HLT	/Halt	109	7001
MUL,	DEC 4	/Multiplier stored here (as counter)	10A	0004
MUC,	DEC 5	/Multiplicand stored here	10B	0005
PRO,	-	/Product stored here	10C	-

16. Write an assembly program to multiply two unsigned positive numbers, each with 16 significant bits, to produce an unsigned double-precision product.

Solution: Assume the Multiplier = Y & the Multiplicand = X, the Product stored in two words of memory PH &PL

	ORG 200	
LOP,	CLE	/Clear E
	LDA Y	/Load multiplier
	CIR	/Shift multiplier bit to E

	STA Y	/Store shifted multiplier
	SZE	/Check if the shifted bit in E is zero
	BUN ONE	/If the bit is one; go to ONE
	BUN ZRO	/If the bit is zero; go to ZRO
ONE,	LDA X ADD	/Load multiplicand
	PL STA PL	/Add low part of partial product PL to the shifted multiplicand
	CLA CIL	/Store in the PL
	ADDPH STA	/Clear AC
	PH CLE LDA	/Shift the carry to AC(0)
	X CIL STAX	/Add the carry to high part of partial product PH
	ISZ CTR	/Store in the PH
	BUNLOP HLT	/Clear E
ZRO,	DEC-16	/Load multiplicand
	HEX 0007	/Shift left d
	HEX000A	/Store shifted multiplicand
	HEX0	/Increment counter
	HEX 0 END	/For counter not zero; repeat loop
CTR, X,		/If counter is zero; Halt
Y,		/This location serves as a counter
PL,		/Location of multiplicand
PH,		/Location of multiplier
		/Location of the low part of product
		/Location of the high part of product

17. Write an assembly program to subtract two double-precision numbers.

Solution

	ORG 100		This part of the program is to obtain the 2's complement of the 2nd number B.
	LDA BL	/load B low	
	CMA	/Complement B low (1's complement)	
	INC	/ obtain 2's complement of B low	
	STA BL	/Store in BL	
	LDABH	/ load B high	
	CMA	/Complement B high (1's complement)	
	SZE	/Skip if E is zero	
	INC	/Increment AC if E = 1	
	STABH	/Store in BH	
AL,	LDA AL	/load A low	
AH,	ADD BL	/Load B low, carry in E	
	STA CL	/Store in C low	

	CLA	/Clear AC
	CIL	/Circulate to bring carry into AC (0)
	ADDAH	/Add A high and carry
	ADDBH	/Add B high
	STA CH	/Store in C high
	HLT	/Halt the computer
	xxxx	/Location of the low significant part of the 1st number
	xxxx	/Location of the high significant part of the 1st number

Assume two positive numbers, the first number A stored in two consecutive memory locations AH & AL, the second number B stored in two consecutive memory locations BH & BL, and the result of subtraction stored in two consecutive memory locations CH & CL.

Loc.	Instruction	Comment
BL,	yyyy	/Location of the low significant part of the 2ndnumber
BH,	yyyy	/Location of the high significant part of the 2ndnumber
CL,	-	/Location of the low significant part of the result
CH,	-	/Location of the high significant part of the result

18. Write an assembly program that evaluates the logic exclusive-OR of two logic operands.

Solution

$X \oplus Y = X\,I\,Y\,Y\,X\,I\,Y$

XIYYXIY

XIYIXIY

Location	Instruction	Comment
	ORG200	
	LDAX	/Load the first operand X
	CMA	/complement to get X
	STRXCO	/Store in the temporary location XCO
	LDA Y	/Load the second operand Y
	CMA	/complement to get Y
	STRYCO	/Store in the temporary location YCO
	AND X	/And with X to get X I Y
	CMA	/Complement to get $\overline{X\,I\,Y}$
	STRTMP	/Store in the temporary location TMP
	LDA Y	/Load the second operand Y
	ANDXCO	/And with X to get X I Y
	CMA	/Complement to get X I Y
	ANDTMP	/And with TMP to get XIYIXIY
	CMA	/Complement to get $\overline{\overline{X\,I\,Y}\;\overline{I\,X\,I\,Y}}$
	STR EOR	/Store in the location EOR = $X \oplus Y$
	HLT	/Halt the program
X,	xxxx	/Location of the first operand
Y,	yyyy	/Location of the second operand

XCO,	-	/Temporary location of X
YCO,	-	/Temporary location of $\overline{Y}$
TMP,	-	/Temporary location of X IY

19. Write a subroutine to circulate E and AC four times to the right. If AC contains hexadecimal 079C and E = 1, what are the contents of AC and E after the subroutine is executed.

Solution

Loc.	Instruction	Comment	Content of AC	Content of E
	ORG 300			
	LDA CAR	/Load AC by carry 1	0001	x
	CIR	/Set the carry to 1	0000 or 8000	1
	LDA NUM	/ Load AC by the number HEX 079C	079C	1
	CIR	/Circulate E & AC to the right	83CE	0
	CIR	/Circulate E & AC to the right	41E7	0
	CIR	/Circulate E & AC to the right	20F3	1
	CIR	/Circulate E & AC to the right	9079	1
	STA NUM	/Store the content of AC in NUM		
	CLA	/Clear AC		
	CIL	/Circulate to the left i.e. AC(0) ← E		
	STA CAR	/Store the content of E in CAR		
	HLT	/Halt the program		
NUM,	HEX 079C	/Location of the number		
CAR,	HEX 0001	/Location of the carry		

Therefore the content of: -

The accumulator AC =9079

The accumulator extension E =1

20. Write an assembly program to accept input characters. Pack two characters in one word and store them in consecutive locations in a memory buffer. The first address of the buffer is $(400)_{16}$. The size of the buffer is $(512)_{10}$ words. If the buffer overflows, the computer should halt.

Solution

Location	Instruction	Comment
	LDA ADS	/Load first address of buffer
	STA PTR	/Initialize the pointer
LOP,	BSA IN2	/Go to subroutine IN2
	STA PTR I	/Store double character word in buffer
	ISZ PTR	/Increment pointer
	ISZ CTR	
	BUN LOP	/Branch to input more characters
IN2, FST,	HLT	/Halt the program
	HEX 0	/Store return address here
	SKI	/Skip if input flag is on
	BUN FST	/Branch unconditional if input flag is 0

SCD	INP	/Input first character
	OUT	/Out first character
	BSA SH4	/Shift left four times
	BSA SH4	/Shift left four more times
SH4,	SKI	/Skip if input flag is on
	BUN	/Branch unconditional if input flag is 0
	SCDINP	/Input second character
	OUT	/Out second character
	BUN IN2I	/Return to the main program
	HEX 0	/Store return address here
	CIL	/Circulate left once
	CIL	/Circulate left second time
	CIL	/Circulate left third time
	CIL	/Circulate left fourth time
MSK,	AND MSK	/Set AC(12-15) to zero
	BUN SH4I	/Return to subroutine IN2
	HEXFFF0	/Mask operand
ADS,	HEX 400	/First address of buffer
PTR,	HEX 0	/Location for pointer
CTR	DEC-512	/Buffer size

21. Translate the service routine SRV from table 7.8 to its equivalent hexadecimal code. Assume that the routine is stored staring from location 200.

Solution

Loc.	Label	Instruction	Comment	Loc.	Inst
0	ZRO,	-	/return address stored here	000	-
1		BUN SRV	/branch to service routine	001	4200
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
100		CLA	/portion of running program	100	7800
101		ION	/turn on interrupt facility	101	F080
102		LDA X		102	2217
103		ADD Y	/interrupt occur here	103	1218
104		STA Z	/Program returns here after interrupt	104	3219
.		.	.	.	
.		.	.	.	
.			/interrupt service here	.	

200 SRV,	STA SAC	/store content of AC	200	3213
	CIR	/move E into AC (15)	201	7080
	STA SE	/store content of E	202	3214
	SKI	/check input flag	203	F200
	BUN NXT	/flag is off, check next flag	204	420E
	INP	/flag is on, input character	205	F800
	OUT	/print character	206	F400
	STA PT1 I	/store it in input buffer	207	B215
	ISZ PT1	/increment input pointer	208	6215
NXT,	SKO	/check output flag	209	F100
	BUN EXT	/flag is off, exit	20A	4209
	LDA PT2 I	/load character from output buffer	20B	A216
	OUT	/output character	20C	F400
	ISZ PT2	/increment output pointer	20D	0216
EXT,	LDA SE	/restore value of AC (15)	20E	2214
	CIL	/shift it to E	20F	7040
	LDA SAC	/restore content of AC	210	2213
	ION	/turn interrupt on	211	F080
	BUN ZRO I	/returntorunningprogram	212	C000
SAC,	-	/AC is stored here	213	-
SE,	-	/E is stored here	214	-
PT1,	-	/pointer of input buffer	215	-
PT2,	-	/pointer of output buffer	216	-
X,	xxxx		217	xxxx
Y,	yyyy		218	yyyy
Z,	-		219	-

Chapter 5

Microprogrammed Control

Objective

This chapter introduces the concept of microprogramming. A specific microprogrammed control unit is developed to show by example how to write microcode for a typical set of instructions. The design of control unit is carried-out in detail including the hardware for the microprogram sequencer.

OUTLINE

5.1 Control Memory
5.2 Address Sequencing
5.3 Microprogram Example
5.4 Design of Control Unit

Chapter 5

Microprogrammed Control

Objective

This chapter introduces the concept of microprogramming. A specific microprogrammed control unit is developed to show by example how to write microcode for a typical set of instructions. The design of control unit is carried-out in detail including the hardware for the microprogram sequencer.

OUTLINE

5.1 Control Memory
5.2 Address Sequencing
5.3 Microprogram Example
5.4 Design of Control Unit

5.1 CENTRAL MEMORY

Hardwired Control Unit

When the control signals are generated by hardware using conventional logic design techniques, the control unit is said to be hardwired.

Micro programmed control unit

A control unit whose binary control variables are stored in memory is called a micro programmed control unit.

Dynamic microprogramming

A more advanced development known as *dynamic* microprogramming permits a microprogram to be loaded initially from an auxiliary memory such as a magnetic disk. Control units that use dynamic microprogramming employ a writable control memory. This type of memory can be used for writing.

Control Memory

Control Memory is the storage in the microprogrammed control unit to store the microprogram.

Writeable Control Memory

Control Storage whose contents can be modified, allow the change in microprogram and Instruction set can be changed or modified is referred as Writeable Control Memory.

Control Word

The control variables at any given time can be represented by a control word string of 1 's and 0's called a control word.

Microoperation, Microinstruction, Micro program, Microcode

Microoperations:

- In computer central processing units, micro-operations (also known as a micro-ops or mops) are detailed low-level instructions used in some designs to implement complex machine instructions (sometimes termed macro-instructions in this context).

Micro instruction:

- A symbolic microprogram can be translated into its binary equivalent by means of an assembler.
- Each line of the assembly language microprogram defines a symbolic microinstruction.
- Each symbolic microinstruction is divided into five fields: label, microoperations, CD, BR, and AD.

Micro program:

- A sequence of microinstructions constitutes a microprogram.

- Since alterations of the microprogram are not needed once the control unit is in operation, the control memory can be a read-only memory (ROM).
- ROM words are made permanent during the hardware production of the unit.
- The use of a micro program involves placing all control variables in words of ROM for use by the control unit through successive read operations.
- The content of the word in ROM at a given address specifies a microinstruction.

Microcode:

- Microinstructions can be saved by employing subroutines that use common sections of microcode.
- For example, the sequence of micro operations needed to generate the effective address of the operand for an instruction is common to all memory reference instructions.
- This sequence could be a subroutine that is called from within many other routines to execute the effective address computation.

Organization of micro programmed control unit

The general configuration of a micro-programmed control unit is demonstrated in the block diagram of Figure 5.1. The control memory is assumed to be a ROM, within which all control information is permanently stored.

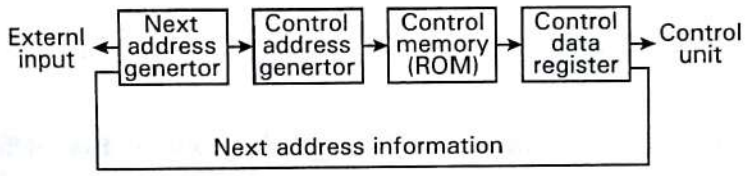


Figure 5.1: *Micro-programmed control organization*

The control memory address register specifies the address of the microinstruction, and the control data register holds the microinstruction read from memory. The microinstruction contains a control word that specifies one or more micro operations for the data processor. Once these operations are executed, the control must determine the next address. The location of the next microinstruction may be the one next in sequence, or it maybe located somewhere else in the control memory. While the microoperations are being executed, the next address is computed in the next address generator circuit and then transferred into the control address register to read the next microinstruction.

Thus, a microinstruction contains bits for initiating microoperations in the data processor part and bits that determine the address sequence for the control memory. The next address generator is sometimes called a ***micro-program sequencer***, as it determines the address sequence that is read from control memory. Typical functions of a micro-program sequencer are incrementing the control address register by one, loading into the control address register an address from control memory, transferring an external address, or loading an initial address to start the control operations. The control data register holds the present microinstruction while the next address is computed and read from memory.

The data register is sometimes called a ***pip line register***. It allows the execution of the microoperations specified by the control word simultaneously with the generation of the next

microinstruction. This configuration requires a two-phase clock, with one clock applied to the address register and the other to the data register. The main advantage of the micro programmed control is the fact that once the hardware configuration is established; there should be no need for further hardware or wiring changes. If we want to establish a different control sequence for the system, all we need to do is specify a different set of microinstructions for control memory.

5.2 ADDRESS SEQUENCING

Microinstructions are stored in control memory in groups, with each group specifying a routine. To appreciate the address sequencing in a micro-program control unit, let us specify the steps that the control must undergo during the execution of a single computer instruction.

Step-1:

- An initial address is loaded into the control address register when power is turned on in the computer.
- This address is usually the address of the first microinstruction that activates the instruction fetch routine.
- The fetch routine may be sequenced by incrementing the control address register through the rest of its microinstructions.
- At the end of the fetch routine, the instruction is in the instruction register of the computer.

Step-2:

- The control memory next must go through the routine that determines the effective address of the operand.
- A machine instruction may have bits that specify various addressing modes, such as indirect address and index registers.
- The effective address computation routine in control memory can be reached through a branch microinstruction, which is conditioned on the status of the mode bits of the instruction.
- When the effective address computation routine is completed, the address of the operand is available in the memory address register.

Step-3:

- The next step is to generate the microoperations that execute the instruction fetched from memory.
- The microoperation steps to be generated in processor registers depend on theoperation code part of the instruction.
- Each instruction has its own micro-program routine stored in a given location of control memory.
- The transformation from the instruction code bits to an address in control memory where the routine is located is referred to as a *mapping* process.
- A mapping procedure is a rule that transforms the instruction code into a control memory address.

Step-4:

- Once the required routine is reached, the microinstructions that execute the instruction may be sequenced by incrementing the control address register.
- Micro-programs that employ subroutines will require an external register for storing the return address.
- Return addresses cannot be stored in ROM because the unit has no writing capability.
- When the execution of the instruction is completed, control must return to the fetch routine.
- This is accomplished by executing an unconditional branch microinstruction to the first address of the fetch routine.

In summary, the address sequencing capabilities required in a control memory are:

1. Incrementing of the control address register.
2. Unconditional branch or conditional branch, depending on status bit conditions.
3. A mapping process from the bits of the instruction to an address for control memory.
4. A facility for subroutine call and return.

Selection of address for control memory

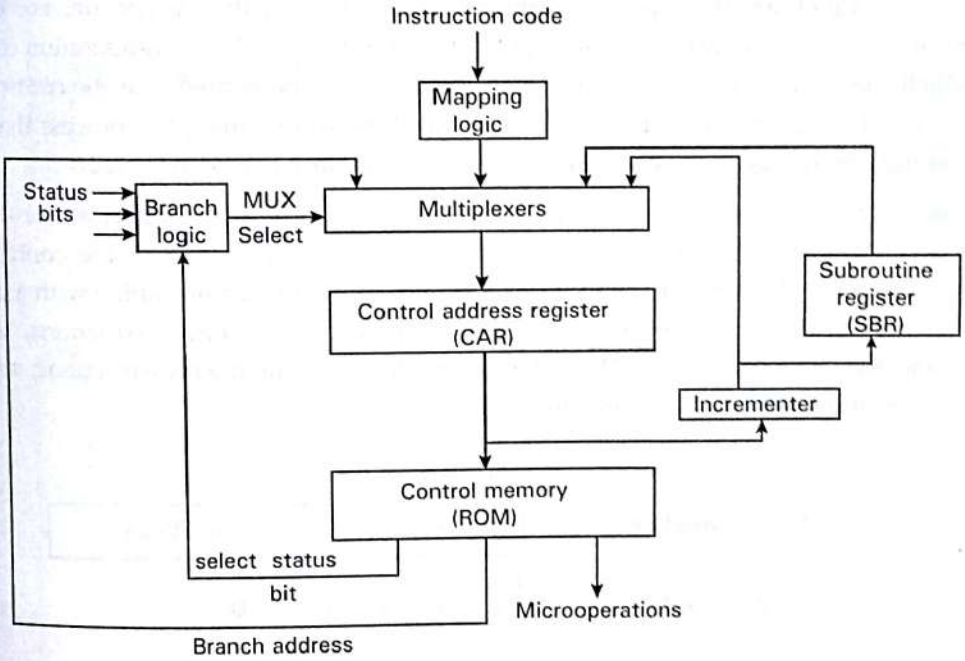


Figure 5.2: *Selection of address for control memory*

Above figure 5.2 shows a block diagram of a control memory and the associated hardware needed for selecting the next microinstruction address. The microinstruction in control memory contains a set of bits to initiate microoperations in computer registers and other bits to specify the method by which the next address is obtained.

The diagram shows four different paths from which the control address register (CAR) receives the address. The incrementer increments the content of the control address register by one, to select the next microinstruction in sequence. Branching is achieved by specifying the branch

address in one of the fields of the microinstruction. Conditional branching is obtained by using part of the microinstruction to select a specific status bit in order to determine its condition. An external address is transferred into control memory via a mapping logic circuit. The return address for a subroutine is stored in a special register whose value is then used when the micro-program wishes to return from the subroutine.

The branch logic of figure 5.2 provides decision-making capabilities in the control unit. The status conditions are special bits in the system that provide parameter information such as the carry-out of an adder, the sign bit of a number, the mode bits of an instruction, and input or output status conditions. The status bits, together with the field in the microinstruction that specifies a branch address, control the conditional branch decisions generated in the branch logic. A 1 output in the multiplexer generates a control signal to transfer the branch address from the microinstruction into the control address register. A 0 output in the multiplexer causes the address register to be incremented.

Mapping of an Instruction

A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a micro program routine for an instruction is located. The status bits for this type of branch are the bits in the operation code part of the instruction. For example, a computer with a simple instruction format as shown in figure 5.3 has an operation code of four bits which can specify up to 16 distinct instructions. Assume further that the control memory has 128 words, requiring an address of seven bits. One simple mapping process that converts the 4-bit operation code to a 7-bit address for control memory is shown in figure 5.3.

This mapping consists of placing a 0 in the most significant bit of the address, transferring the four operation code bits, and clearing the two least significant bits of the control address register. This provides for each computer instruction a microprogram routine with a capacity of four microinstructions. If the routine needs more than four microinstructions, it can use addresses1000000 through 1111111. If it uses fewer than four microinstructions, the unused memory locations would be available for other routines.

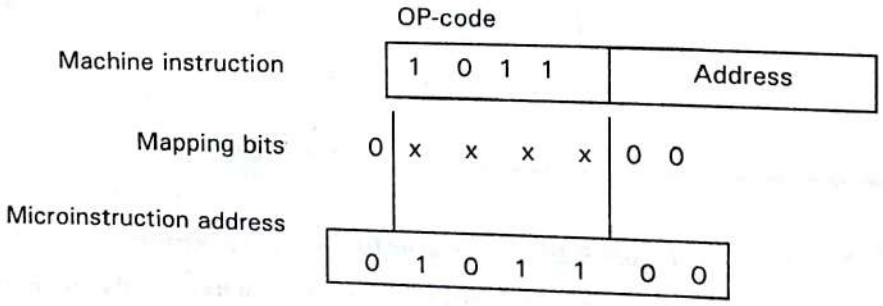


Figure 5.3: *Mapping from instruction code to microinstruction address*

One can extend this concept to a more general mapping rule by using a ROM to specify the mapping function. The contents of the mapping ROM give the bits for the control address register. In this way the microprogram routine that executes the instruction can be placed in any desired location in control memory. The mapping concept provides flexibility for adding instructions for control memory as the need arises.

Subroutines

Subroutines are programs that are used by other routines to accomplish particular task. A subroutine can be called from any point within the main body of the microprogram. Frequently, many microprograms contain identical sections of code. Microinstructions can be saved by employing subroutines that use common sections of microcode. For example, the sequence of microoperations needed to generate the effective address of the operand for n instruction is common to all memory reference instructions. This sequence could be a subroutine that is called from within many other routines to execute the effective address computation.

5.3 COMPUTER HARDWARE CONFIGURATION

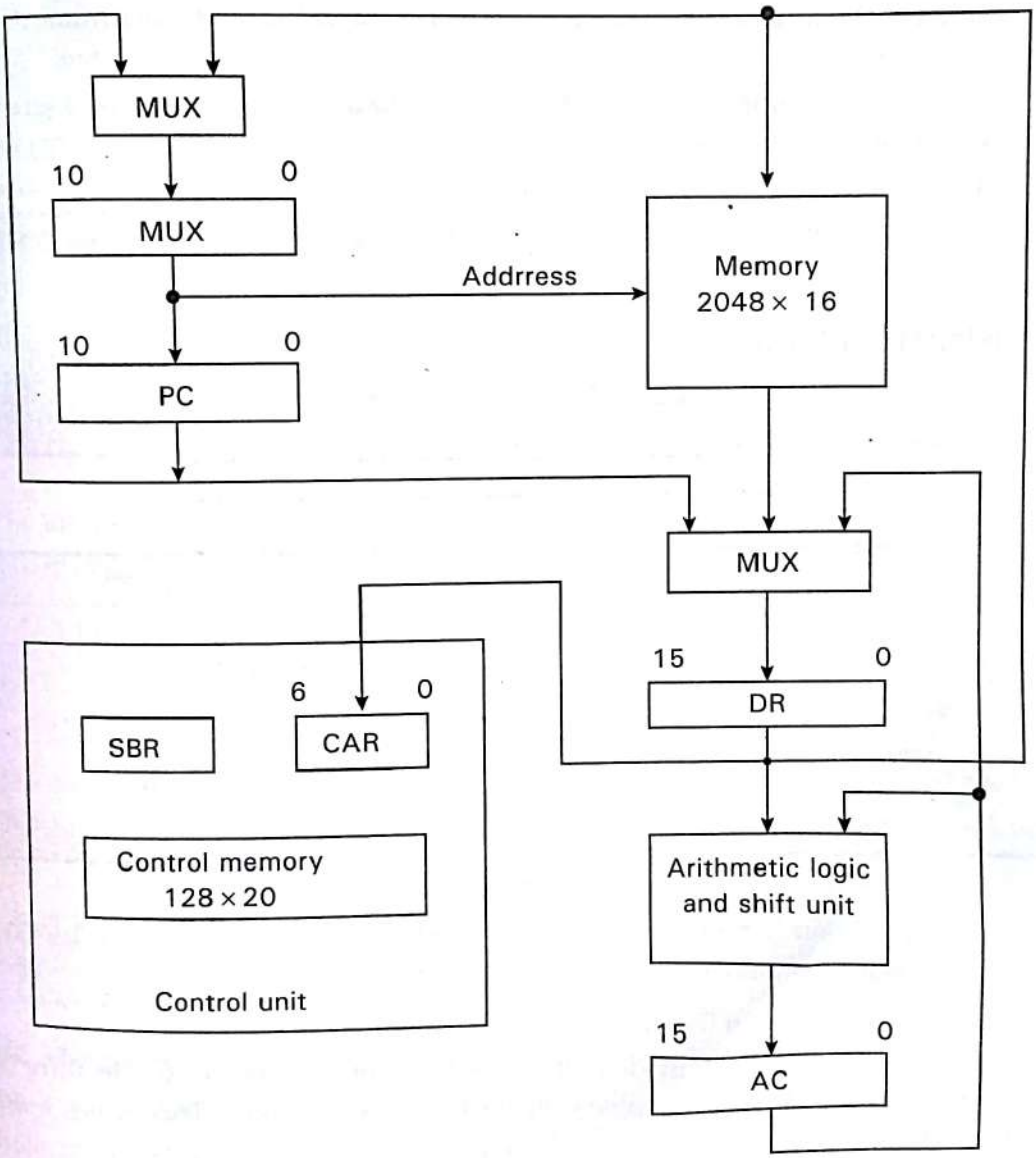


Figure 5.4: *Computer hardware configuration*

The block diagram of the computer is shown in Figure 5.4. It consists of

1. **Two memory units:**

   Main memory -> for storing instructions and data, and Control memory -> for storing the microprogram.

2. **Six Registers:**

   Processor unit register: AC(accumulator),PC(Program Counter), AR(Address Register), DR(Data Register)

   Control unit register: CAR (Control Address Register), SBR(Subroutine Register)

3. **Multiplexers:**

   The transfer of information among the registers in the processor is done through multiplexers rather than a common bus.

4. **ALU:**

   The arithmetic, logic, and shift unit performs microoperations with data from AC and DR and places the result in AC.

   - DR can receive information from AC, PC, or memory.
   - AR can receive information from PC or DR.
   - PC can receive information only from AR.
   - Input data written to memory come from DR, and data read from memory can go only to DR.

Microinstruction Format

Figure 5.5: *Computer instructions*

15	14 11	10 0
I	Opcode	Address

(a) Instruction format

Symbol	Opcode	Description
ADD	0000	AC ← AC + M[EA]
BRANCH	0001	If (AC < 0) then (PC ← EA)
STORE	0010	M[EA] ← AC
EXCHANGE	001	AC ← M[EA], M[EA] ← AC
EA is the effective address		

(b) Four computer instructions

The microinstruction format for the control memory is shown in figure 4.5. The 20 bits of the microinstruction are divided into four functional parts as follows:

1. The three fields F1, F2, and F3 specify microoperations for the computer.

   The microoperations are subdivided into three fields of three bits each. The three bits in each field are encoded to specify seven distinct microoperations. This gives a total of 21 microoperations.

2. The CD field selects status bit conditions.
3. The BR field specifies the type of branch to be used.

4. The AD field contains a branch address. The address field is seven bits wide, since the control memory has 128 = 27words.

3	3	3	2	2	7
F1	F2	F3	CD	BR	AD

F1, F2, F3: Microoperations fields
CD : Condition for branching
BR : Branch field
AD : Address fieldd

Figure 5.6: Microinstruction Format

- As an example, a microinstruction can specify two simultaneous microoperations from F2 and F3 and none from F1.

$DR \leftarrow M[AR]$ with F2 = 100

and $PC \leftarrow PC + 1$ with F3 = 101

- The nine bits of the microoperation fields will then be 000 100101.
- The CD (condition) field consists of two bits which are encoded to specify four status bit conditions as listed in Table 5.1.

CD	Condition	Symbol	Comments
00	Always = 1	U	Unconditional branch
01	DR (15)	I	Indirect address bit
10	AC (15)	S	Sign bit of AC
11	AC = 0	Z	Zero value in Ac

Table 5.1: Condition Field

- The BR (branch) field consists of two bits. It is used, in conjunction with the address field AD, to choose the address of the next microinstruction shown in Table 5.2.

Table 5.2: Branch Field

BR	Symbol	Function
00	JMP	CAR ← AD if condition = 1
		CAR ← CAR + 1 if condition = 0
01	CALL	CAR ← AD, SBR ← CAR + 1 if condition = 1
		CAR ← CAR + 1 if condition = 0
10	RET	CAR ← SR (Return from subroutine)
11	MAP	CAR (2-5) ← DR(11-14), CAR(0,1,6) ← 0

Symbolic Microinstruction

Each line of the assembly language microprogram defines a symbolic microinstruction. Each symbolic microinstruction is divided into five fields: label, microoperations, CD, BR, and AD. The fields specify the following Table 5.3.

1.	Label	The label field may be empty or it may specify a symbolic address. A label is terminated with a colon (:).
2.	Microoperations	It consists of one, two, or three symbols, separated by commas, from those defined in Table 5.3. There may be no more than one symbol from each F field. The NOP symbol is used when the microinstruction has no microoperations. This will be translated by the assembler to nine zeros.
3.	CD	The CD field has one of the letters U, I, S, or Z.
4.	BR	The BR field contains one of the four symbols defined in Table 5.2.
5.	AD	The AD field specifies a value for the address field of the microinstruction in one of three possible ways. With a symbolic address, this must also appear as a label. With the symbol NEXT to designate the next address in sequence. When the BR field contains a RET or MAP symbol, the AD field is left empty and is converted to seven zeros by the assembler.

Table 5.3 Symbolic Microinstruction

5.4 Design of Control Unit

Usually, the bits of the microinstruction are divided into fields, with each field defining a distinct, separate function. The various fields encountered in the instruction formats provide control bits to initiate microoperations in the system, special bits to specify the way that the next address is to be evaluated, and an address field for branching.

Field Decoding

Since, there are three microoperation fields we need 3 decoders. Only some of the outputs of decoders are shown to be connected to their output. Each of the output of the decoders must be connected to the proper circuit to initiate the corresponding microoperation. For example when F1=101 the next clock pulse transition transfers content of DR(0-10) to AR (Symbolized by DRTAC). Similarly other operations are also performed.

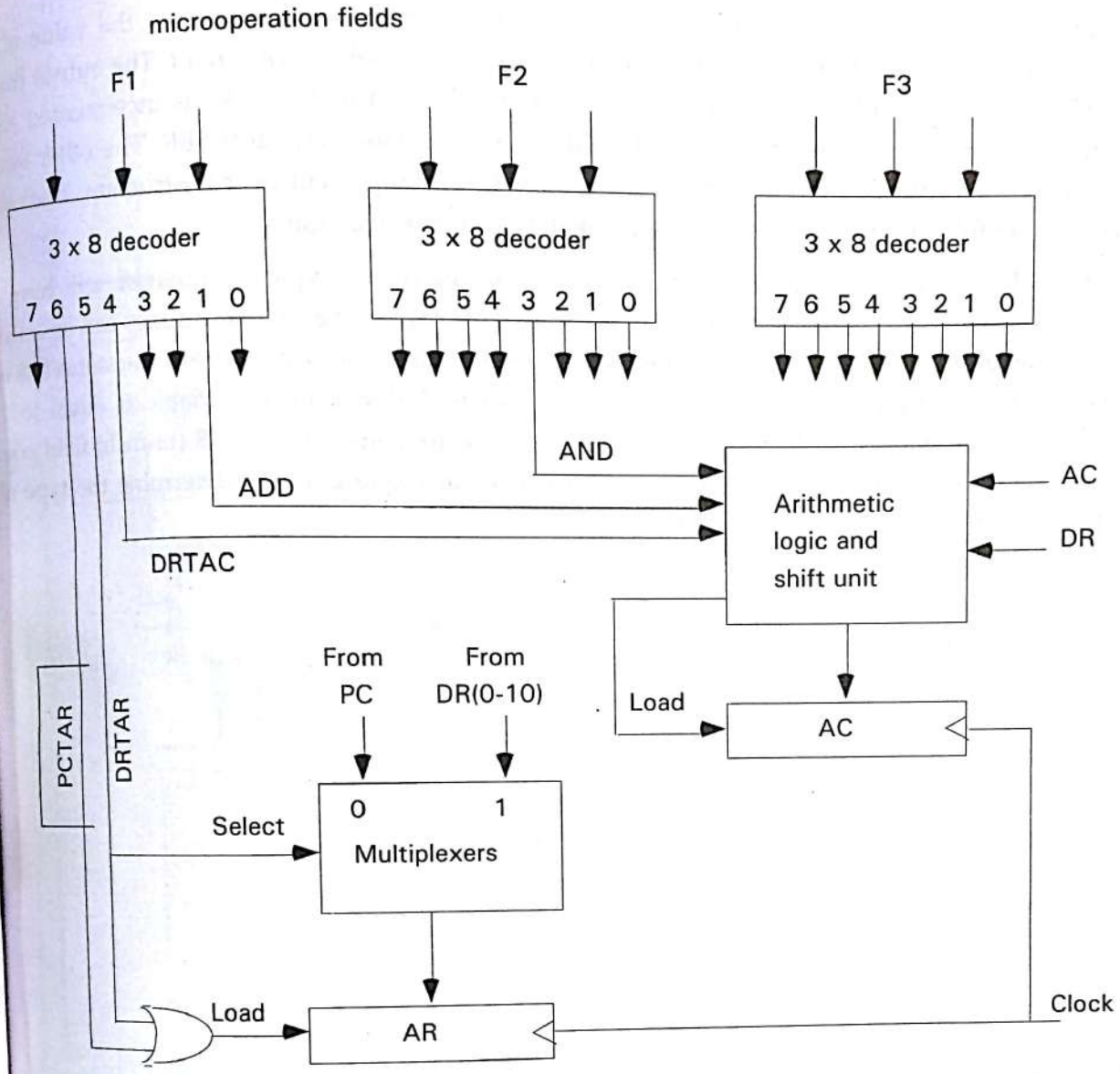


Figure 5.7: *Decoding of Microoperation fields*

Micro programmed sequencer for a control memory

The basic components of a microprogrammed control unit are the control memory and the circuits that select the next address. The address selection part is called a microprogram sequencer. A microprogram sequencer can be constructed with digital functions to suit a particular application. To guarantee a wide range of acceptability, an integrated circuit sequencer must provide an internal organization that can be adapted to a wide range of applications. The purpose of a microprogram sequencer is to present an address to the control memory so that a microinstruction may be read and executed. Commercial sequencers include within the unit an internal register stack used for temporary storage of addresses during microprogram looping and subroutine calls. Some sequencers provide an output register which can function as the address register for the control memory.

The block diagram of the microprogram sequencer is shown in figure 5.8. There are two multiplexers in the circuit. The first multiplexer selects an address from one of four sources and

routes it into a control address register CAR. The second multiplexer tests the value of a selected status bit and the result of the testis applied to an input logic circuit. The output from CAR provides the address for the control memory. The content of CAR is incremented and applied to one of the multiplexer inputs and to the subroutine registers SBR. The other three inputs to multiplexer 1 come from the address field of the present microinstruction, from the output of SBR, and from an external source that maps the instruction.

Although the figure 5.8 shows a single subroutine register, a typical sequencer will have a register stack about four to eight levels deep. In this way, a number of subroutines can be active at the same time. The CD (condition) field of the microinstruction selects one of the status bits in the second multiplexer. If the bit selected is equal to 1, the T (test) variable is equal to 1; otherwise, it is equal to 0. The T value together with the two bits from the BR (branch) field goes to an input logic circuit. The input logic in a particular sequencer will determine the type of operations that are available in the unit.

Figure 5.8: *Microprogram Sequencer for a control memory*

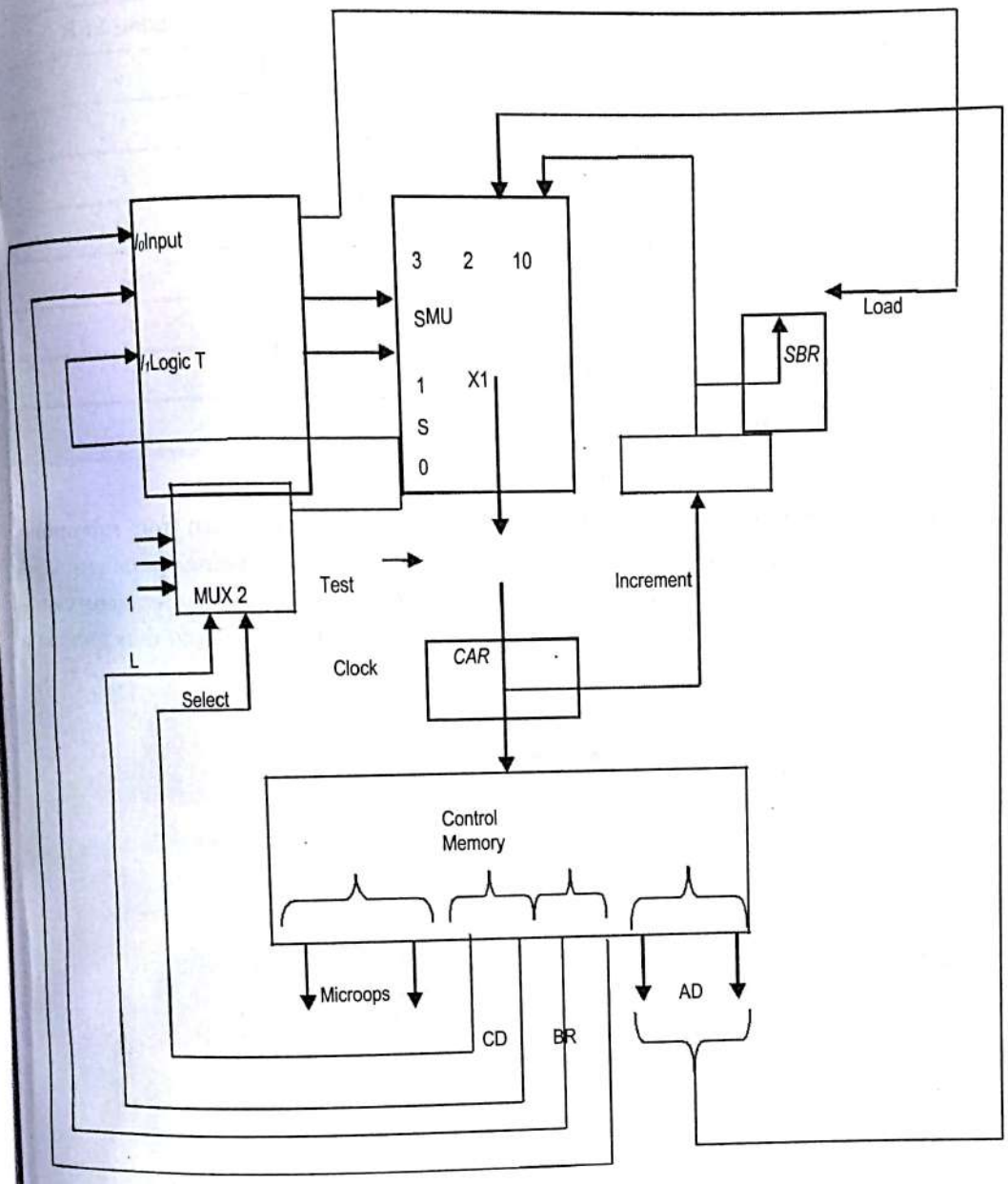


Table 5.2: Input *Logic : Truth Table for microprogram sequencer*

BR	Input			MUX 1		Load SBR
	I1	I0	T	S1	S0	L
0 0	0	0	0	0	0	0
0 0	0	0	1	0	1	0
0 1	0	1	0	0	0	0
0 1	0	1	1	0	1	1
1 0	1	0	X	1	0	0
1 1	1	1	X	1	1	0

Boolean Function: S0 = I0

= I0I1 + I0'T L = I0'I1T

Typical sequencer operations are: increment, branch or jump, call and return from subroutine, load an external address, push or pop the stack, and other address sequencing operations. With three inputs, the sequencer can provide up to eight address sequencing operations. Some commercial sequencers have three or four inputs in addition to the T input and thus provide a wider range of operations.

Chapter 6

Central Processing Unit

Objective

This chapter deals with the central processing unit(CPU). An execution unit with common buses and an arithmetic logic unit is developed to show the general register organization of a typical CPU. The operation of a memory stack is explained and some of its applications are demonstrated. Various instruction format are illustrated together with a variety of addressing modes. The most common instructions found in computers are enumerated with an explanation of their function. The last section introduces the reduced instruction set computer (RISC) concept and discuss its characteristics and advantages.

OUTLINE

6.1 Introduction
6.2 General Register Organization
6.3 Strack Organization
6.4 Instruction Formats
6.5 Addressing Modes
6.6 Reduced Instruction Set Computer

6.1 INTRODUCTION

The Central Processing Unit (CPU) is called the brain of the computer that performs data-processing operations. Figure 6.1 shows the three major parts of CPU.

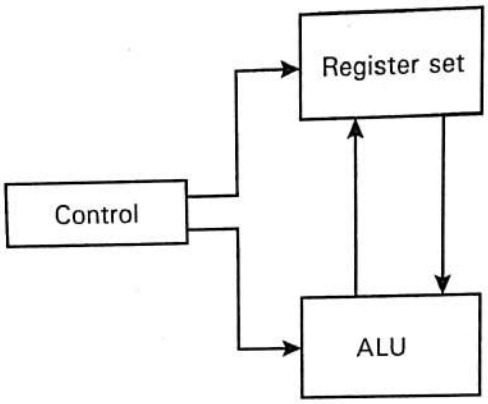


Figure 6.1: *Major components of CPU*

Intermediate data is stored in the register set during the execution of the instructions. The microoperations required for executing the instructions are performed by the arithmetic logic unit whereas the control unit takes care of transfer of information among the registers and guides the ALU. The control unit services the transfer of information among the registers and instructs the ALU about which operation is to be performed. The computer instruction set is meant for providing the specifications for the design of the CPU. The design of the CPU largely, involves choosing the hardware for implementing the machine instructions.

The need for memory locations arises for storing pointers, counters, return address, temporary results and partial products. Memory access consumes the most of the time off an operation in a computer. It is more convenient and more efficient to store these intermediate values in processor registers.

6.2 GENERAL REGISTER ORGANIZATION

A common bus system is employed to contact registers that are included in the CPU in a large number. Communications between registers is not only for direct data transfer but also for performing various micro-operations. A bus organization for such CPU register shown in Figure 6.2, is connected to two multiplexers (MUX) to form two buses A and B. The selected lines in each multiplexers select one register of the input data for the particular bus.

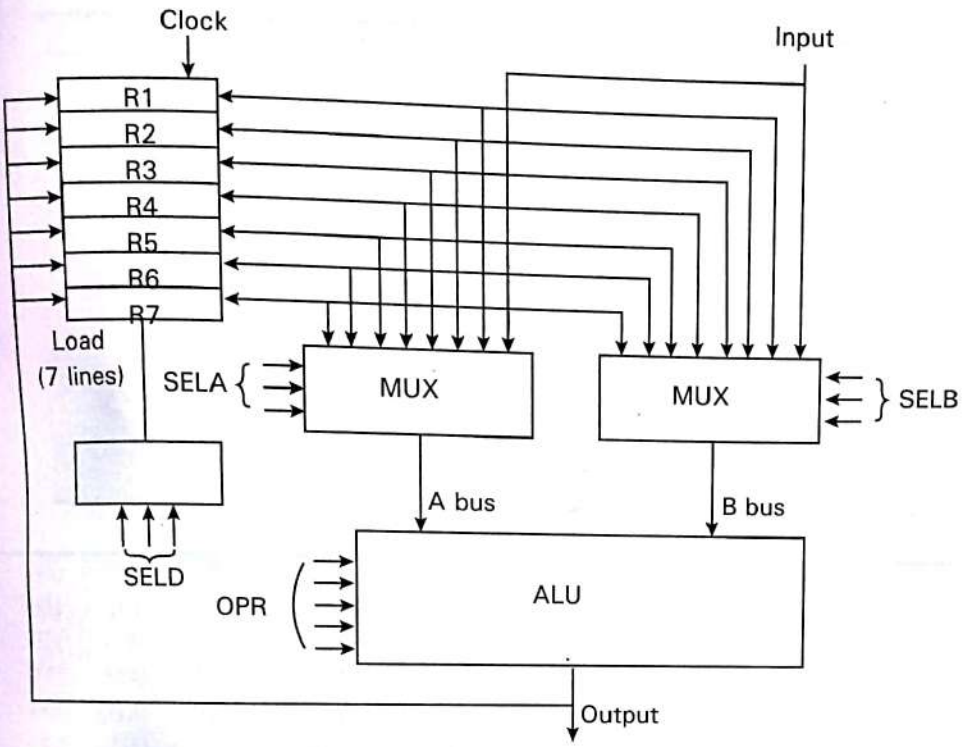


Figure 6.2: *Register set with common ALU*

Operation of Control Unit:

The control unit directs the information flow through ALU by:

- Selecting various *Components* in the system
- Selecting the *Function* of ALU

Example: R1 <- R2 + R3

MUX A selector (SELA): BUS A← R2

MUX B selector (SELB): BUS B ←R3

ALU operation selector (OPR): ALU to ADD

Decoder destination selector (SELD): R1 ← Out Bus

Control Word

3	3	W	5
SELA	SELB	SELD	OPR

Table 6.1: *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

Table 6.2: *Encoding of ALU operations*

OPR Select	Operation	Symbol
00000	Transfer A	TSFA
00001	Increment A	INCA
00010	ADD A + B	ADD
00101	Subtract A – B	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

Table 6.3: *Examples of microoperations for the CPU*

R 1 ← R2 -R3	R2	R3	R1	SUB	010	011	001	00101
R4 ← R4 ∨ R5	R4	R4	R4	OR	100	101	100	01010
R6 ← R6 + 1	R6	–	R6	INCA	110	000	110	00001
R7 ← R1	R1	–	R7	TSFA	001	000	111	00000
Output ← R2	R2	–	None	TSFA	010	000	000	00000
Output ← Input	Input	–	None	TSFA	000	000	000	00000
R4 ← shl R4	R4	-	R4	SHLA	100	000	100	11000
R5 ← 0	R5	R5	R5	XOR	101	101	101	01100

6.3 STACK ORGANIZATION

A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved. The stack in digital computers is essentially a memory unit with an address register that can count only. The register that holds the address for the stack is called a stack pointer (SP) because its value always points at the top item in the stack. The physical registers of a stack are always available for reading or writing. It is the content of the word that is inserted or deleted.

Register stack

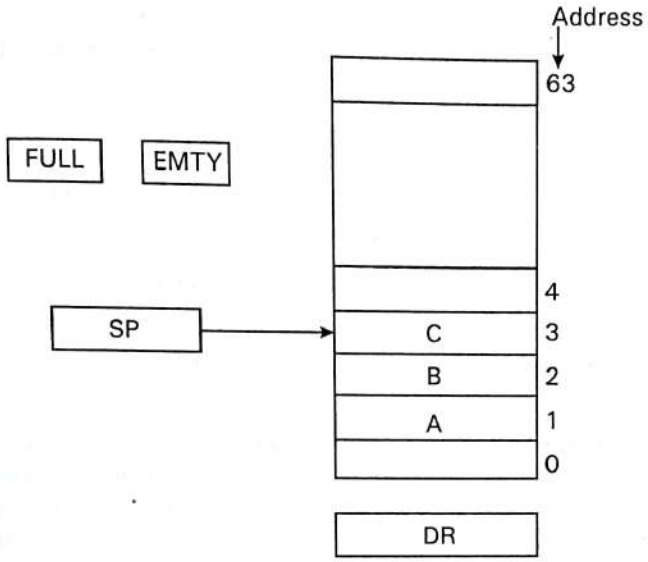


Figure 6.1: *Block diagram of a 64-word stack*

A stack can be placed in a portion of a large memory or it can be organized as a collection of a finite number of memory words or registers. Figure shows the organization of a 64- word register stack. The stack pointer register SP contains a binary number whose value is equal to the address of the word that is currently on top of the stack. Three items are placed in the stack: A, B, and C, in that order. Item C is on top of the stack so that the content of SP is now 3. To remove the top item, the stack is popped by reading the memory word at address 3 and decrementing the content of SP. Item B is now on top of the stack since SP holds address 2. To insert a new item, the stack is pushed by incrementing SP and writing a word in the next-higher location in the stack.

In a 64-word stack, the stack pointer contains 6 bits because 2^6 =64. Since SP has only six bits, it cannot exceed a number greater than 63 (111111 in binary). When 63 are incremented by 1, the result is 0 since 111111 + 1 = 1000000 in binary, but SP can accommodate only the six least significant bits. Similarly, when 000000 is decremented by 1, the result is 111111. The one-bit register FULL is set to 1 when the stack is full, and the one-bit register EMTY is set to 1 when the stack is empty of items. DR is the data register that holds the binary data to be written into or read out of the stack.

PUSH

If the stack is not full (FULL =0), a new item is inserted with a push operation. The push operation consists of the following sequences of microoperations:

SP ← SP+1	Increment stackpointer
M [SP]←DR	WRITE ITEM ON TOP OF THE STACK
IF (SP = 0) then (FULL ← 1)	Check is stack is full EMTY← 0 Mark the stack not empty

The stack pointer is incremented so that it points to the address of next-higher word. A memory write operation inserts the word from DR into the top of the stack. SP holds the address of the top of the stack and that M[SP] denotes the memory word specified by the address presently

available in SP. The first item stored in the stack is at address 1. The last item is stored at address 0. If SP reaches 0, the stack is full of items, so FULL is set to 1. This condition is reached if the top item prior to the last push was in location 63 and, after incrementing SP, the last item is stored in location 0. Once an item is stored in location 0, there are no more empty registers in the stack. If an item is written in the stack, obviously the stack cannot be empty, so EMTY is cleared to 0.

POP:

A new item is deleted from the stack if the stack is not empty (if EMTY = 0). The pop operation consists of the following sequences of microoperations:

DR ←M[SP]	Read item on top of the stack
SP ← SP -1	Decrement stack pointer
IF (SP = 0) then (EMTY ← 1)	Check if stack is empty
FULL←0	Mark the stack notfull

The top item is read from the stack into DR. The stack pointer is then decremented. If its value reaches zero, the stack is empty, so EMTY is set to 1. This condition is reached if the item read was in location 1. Once this item is read out, SP is decremented and reaches the value 0, which is the initial value of SP. If a pop operation reads the item from location 0 and then SP is decremented, SP is changes to 111111, which is equivalent to decimal 63. In this configuration, the word in address 0 receives the last item in the stack. Note also that an erroneous operation will result if the stack is pushed when FULL = 1 or popped when EMTY =1.

Memory Stack

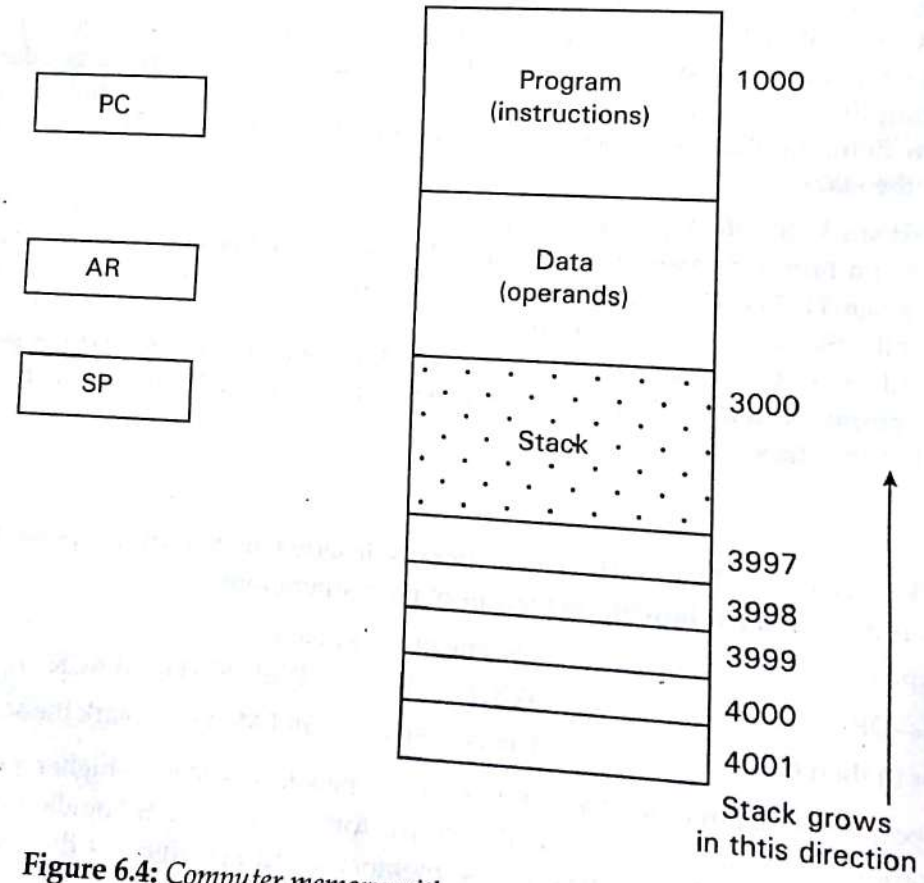


Figure 6.4: *Computer memory with program, data, and stack segments*

The implementation of a stack in the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer. Figure 6.4 shows a portion of computer memory partitioned into three segments: program, data, and stack. The program counter PC points at the address of the next instruction in the program which is used during the fetch phase to read an instruction. The address registers AR points at an array of data which is used during the execute phase to read an operand.

The stack pointer SP points at the top of the stack which is used to push or pop items into or from the stack. The three registers are connected to a common address bus, and either one can provide an address for memory. As shown in Figure 6.4, the initial value of SP is 4001 and the stack grows with decreasing addresses. Thus the first item stored in the stack is at address 4000, the second item is stored at address 3999, and the last address that can be used for the stack is 3000. We assume that the items in the stack communicate with a data register DR.

PUSH

A new item is inserted with the push operation as follows:

$$SP \leftarrow SP - 1 \quad M[SP] \leftarrow DR$$

The stack pointer is decremented so that it points at the address of the next word. A memory write operation inserts the word from DR into the top of the stack.

POP

A new item is deleted with a pop operation as follows:

$$DR \leftarrow M[SP] \qquad SP \leftarrow SP + 1$$

The top item is read from the stack into DR. The stack pointer is then incremented to point at the next item in the stack. The two microoperations needed for either the push or pop are (1) an access to memory through SP, and (2) updating SP. Which of the two microoperations is done first and whether SP is updated by incrementing or decrementing depends on the organization of the stack.

In figure. 6.4 the stack grows by decreasing the memory address. The stack maybe constructed to grow by increasing the memory also. The advantage of a memory stack is that the CPU can refer to it without having to specify an address, since the address is always available and automatically updated in the stack pointer.

6.4 INSTRUCTION FORMATS

Instruction fields

OP-code field - specifies the operation to be performed

Address field - designates memory address(s) or a processor register(s)

Modefield - specifies the way the operand or the effective address is determined.

The number of address fields in the instruction format depends on the internal organization of CPU -

The three most common CPU organizations:

Single accumulator organization

```
ADD   X                  /*AC ← AC + M[X] */
```

General register organization

```
ADD   R1, R2, R3         /* R1 ← R2 + R3 */
ADD   R1, R2             /* R1 ← R1 + R2 */
MOV   R1, R2             /* R1 ← R2 */
ADD   R1, X              /* R1 ← R1 + M[X] */
```

Stack organization

```
PUSH  X                  /* TOS ← M[X] */
```

Three-Address Instructions:

Program to evaluate X = (A + B) * (C + D):

```
ADD   R1, A, B           /* R1 ← M[A] + M[B] */
ADD   R2, C, D           /* R2 ← M[C] + M[D] */
MUL   X, R1, R2          /* M[X] ← R1 * R2 */
```

- Result in short programs
- Instruction becomes long (many bits)

Two-Address Instructions:

Program to evaluate X = (A + B) * (C + D):

```
MOV   R1, A              /* R1 ← M[A] */
ADD   R1, B              /* R1 ← R1 + M[B] */
MOV   R2, C              /* R2 ← M[C] */
ADD   R2, D              /* R2 ← R2 + M[D] */
MUL   R1, R2             /* R1 ← R1 * R2 */
MOV   X, R1              /* M[X] ← R1 */
```

ONE, and ZERO-ADDDRESS INSTRUCTIONS

ONE-ADDRESS INSTRUCTIONS:

- Use an implied AC register for all data manipulation
- Program to evaluate X = (A + B) * (C + D):

```
LOAD   A                 /* AC ← M[A] */
ADD    B                 /* AC ← AC + M[B] */
STORE  T                 /* M[T] ← AC */
LOAD   C                 /* AC ← M[C] */
ADD    D                 /* AC ← AC + M[D] */
MUL    T                 /* AC ← AC + M[T] */
STORE  X                 /* M[X] ← AC */
```

6.5 ADDRESSING MODES

Specifies a rule for interpreting or modifying the address field of the instruction (before the operand is actually referenced) Variety of addressing modes

- to give programming flexibility to the user
- to use the bits in the address field of the instruction efficiently

TYPES OF ADDRESSING MODES

Implied mode

Address of the operands are specified implicitly, in the definition of the instruction

- No need to specify address in the instruction
- EA = AC, or EA = Stack [SP]

Immediate mode

Instead of specifying the address of the operand, operand itself is specified

- No need to specify address in the instruction
  - However, operand itself needs to be specified
  - Sometimes, require more bits than the address
  - Fast to acquire an operand

Register mode

Address specified in the instruction is the register address

- Designated operand need to be in a register
- Shorter address than the memory address
- Saving address field in the instruction
- Faster to acquire an operand than the memory addressing
- EAA = IR (R) (IR(R): Register field of IR)

Register Indirect Mode

Instruction specifies a register which contains the memory address of the operand

- Saving instruction bits since register address is shorter than the memory address
- Slower to acquire an operand than both the register addressing or memory addressing
- EA = [IR(R)] ([x]: Content of x)

Auto-increment or Auto-decrement features:

Same as the Register Indirect, but:

- When the address in the register is used to access memory, the value in the register is incremented or decremented by 1 (after or before the execution of the instruction)

Direct Address Mode

Instruction specifies the memory address which can be used directly to the physical memory

- Faster than the other memory addressing modes
- Too many bits are needed to specify the address for a large physical memory space
- EA = IR(address), (IR(address): address field of IR)

Indirect Addressing Mode

The address field of an instruction specifies the address of a memory location that contains the address of the operand

- When the abbreviated address is used, large physical memory can be addressed with a relatively small number of bits
- Slow to acquire an operand because of an additional memory access
- EA =M[IR(address)]

Relative Addressing Modes

The Address fields of an instruction specifies the part of the address (abbreviated address) which can be used along with a designated register to calculate the address of theoperand

- PC Relative Addressing Mode(R = PC)
  - EA = PC + IR(address)
- Address field of the instruction is short
- Large physical memory can be accessed with a small number of address bits

Indexed Addressing Mode

XR: Index Register:

- EA = XR + IR(address)

Base Register Addressing Mode

BAR: Base Address Register:

- EA = BAR + IR(address)

Addressing Modes -Examples

The figure given below demonstrates the difference between different addressing modes.

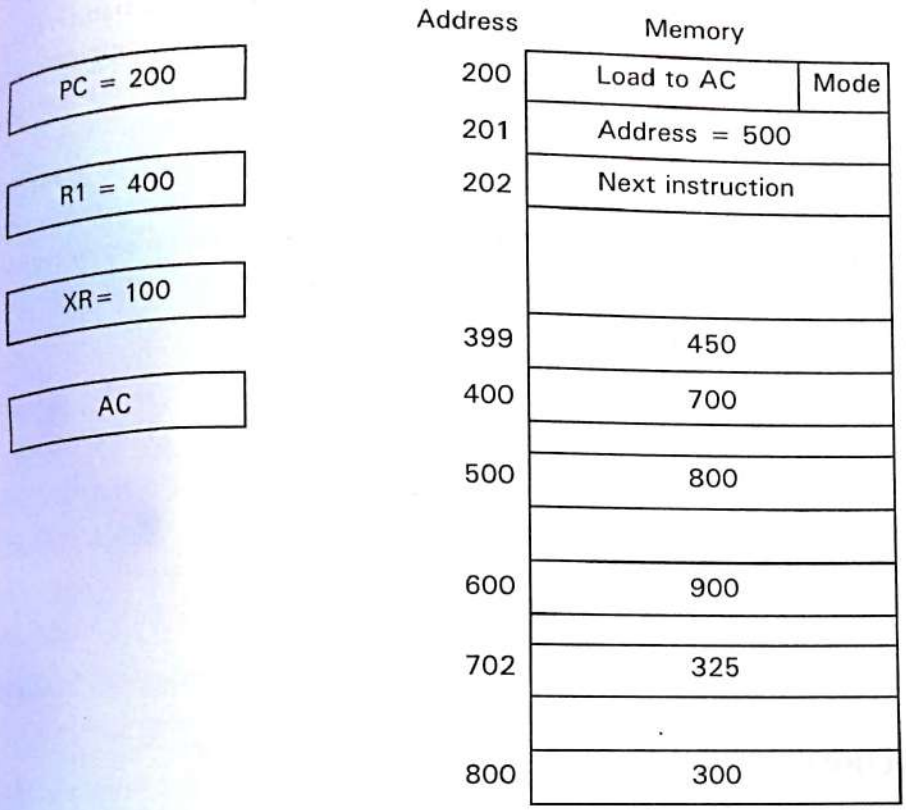


Figure 6.5: *Numerical example for addressing modes*

Addressing Mode	Effective Address		Content of AC
Direct address	500	AC ←(500)	800
Immediate operand	-	AC ←500	500
Indirect address	800	AC ←((500))	300
Relative address	702	AC ←(PC+500)	325
Indexed address	600	AC ←(XR+500)	900
Register	-	AC ←R1	400
Register indirect	400	AC ←(R1)	700
Autoincrement	400	AC ←(R1)+	700
Autodecrement	399	AC ←-(R)	450

6.6 Data Transfer Instructions

Data transfer instructions move data from one place in the computer to another without changing the data content. The most common transfers are between memory and processor registers, between processor registers and input or output, and between the processor registers themselves. The *load* instruction has been used mostly to designate a transfer from memory to a

processor register, usually an accumulator and the *store* instruction designates a transfer from a processor register into memory. The *move* instruction has been used in computers with multiple CPU registers to designate a transfer from one register to another. It has also been used for data transfers between CPU registers and memory or between two memory words. The *exchange* instruction swaps information between two registers or a register and a memory word. The *input and output* instructions transfer data among processor registers and input or output terminals and the *push and pop* instructions transfer data between processor registers and a memory stack.

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

Data Transfer Instructions

Typical data transfer instruction

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

Data transfer instructions with different addressing modes

Mode	Assembly Convention		Register Transfer
Direct address	LD	ADR	AC ← M[ADR]
Indirect address	LD	@ADR	AC ← M[M[ADR]]
Relative address	LD	$ADR	AC ← M[PC + ADR]
Immediate operand	LD	#NBR	AC ←NBR
Index addressing	LD	ADR(X)	AC ← M[ADR + XR]
Register	LD	R1	AC ← R1
Register indirect	LD	(R1)	AC ← M[R1]
Auto increment	LD	(R1) +	AC ← M[R1], R1 ← R1 + 1
Auto decrement	LD	(R1) –	R1 ← R1 - 1 AC ← M[R1]

Data Manipulation Instructions

Three Basic Types:

- Arithmetic instructions
- Logical and
- Bit manipulation instructions

Shift instructions

Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right thru carry	RORC
Rotate left thru carry	ROLC

Arithmetic Instructions

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with Carry	ADDC
Subtract with Borrow	SUBB
Negate(2's Complement)	NEG

Logical and Bit Manipulation Instructions

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI

Program Control Instructions

It is sometimes convenient to supplement the ALU circuit in the CPU with a status register where status bit conditions be stored for further analysis. Status bits are also called condition-code bits or flag bits.

Figure 6.6 shows the block diagram of an 8-bit ALU with a 4-bit status register. The four status bits are symbolized by C, S, Z, and V. The bits are set or cleared as a result of an operation performed in the ALU.

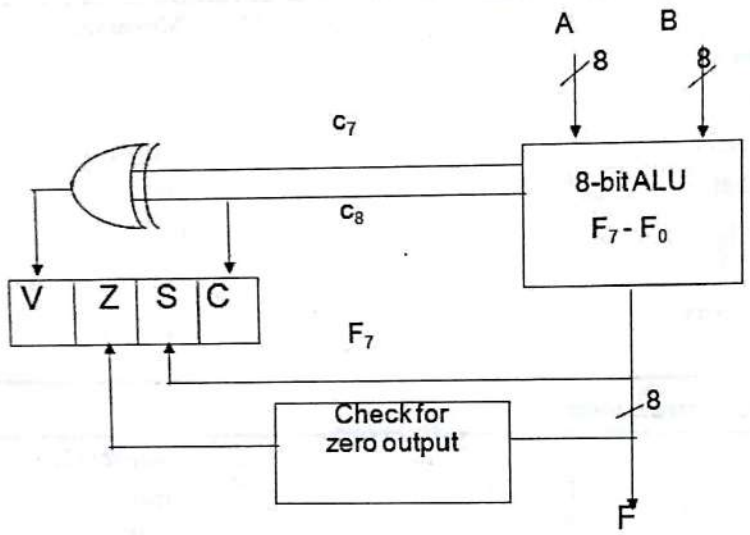


Figure 6.6: *Status Register Bits*

- Bit C (carry) is set to 1 if the end carry C8 is 1. It is cleared to 0 if the carry is0.
- Bit S (sign) is set to 1 if the highest-order bit F7 is 1. It is set to 0 if set to 0 if the bit is 0.
- Bit Z (zero) is set to 1 if the output of the ALU contains all 0's. it is cleared to 0 otherwise. In other words, Z = 1 if the output is zero and Z = 0 if the output is not zero.
- Bit V (overflow) is set to 1 if the exclusives-OR of the last two carries is equal to 1, and cleared to 0 otherwise. This is the condition for an overflow when negative numbers are in 2's complement. For the 8-bit ALU, V = 1 if the output is greater than + 127 or less than-128.

The status bits can be checked after an ALU operation to determine certain relationships that exist between the vales of A and B. If bit V is set after the addition of two signed numbers, it indicates an overflow condition. If Z is set after an exclusive-OR operation, it indicates that A =B. A single bit in A can be checked to determine if it is 0 or 1 by masking all bits except the bit in question and then checking the Z status bit.

Unconditional Branching Instructions

Name	Mnemonic
Branch	BR
Jump	JMP

Skip	SKP
Call	CALL
Return	RTN
Compare (by -)	CMP
Test (by AND)	TST

Program Control Instructions

CMP and TST instructions do not retain their results of operations(- and AND, respectively). They only set or clear certain Flags.

Conditional branch Instructions

Mnemonic	Branch condition	Tested condition
BZ	Branch if zero	Z =1
BNZ	Branch if not zero	Z =0
BC	Branch if carry	C =1
BNC	Branch if no carry	C =0
BP	Branch if plus	S =0
BM	Branch if minus	S =1
BV	Branch if over flow	V =1
BNV	Branch if no over flow	V =0
***Unsigned* compare conditions (A - B)**		
BHI	Branch if higher	A >B
BHE	Branch if higher or equal	A ≥B
BLO	Branch if lower	A <B
BLOE	Branch if lower or equal	A ≤B
BE	Branch if equal	A =B
BNE	Branch if not equal	A ≠B
***Signed* compare conditions (A - B)**		
BGT	Branch if greater than	A >B
BGE	Branch if greater or equal	A ≥B
BLT	Branch if less than	A <B
BLE	Branch if less or equal	A ≤B
BE	Branch if equal	A =B
BNE	Branch if not equal	A ≠B

Subroutine Call and Return

Subroutine Call

Call subroutine Jump to subroutine Branch to subroutine

Branch and save return address

Two Most Important Operations are implied;

- Branch to the beginning of the Subroutine
  - Same as the Branch or Conditional Branch

- Save the Return Address to get the address of the location in the Calling Program upon exit from the Subroutine
  - Locations for storing Return address:
    - Fixed Location in the subroutine(Memory)
    - Fixed Location in memory
    - In a process or Register
    - In a memory stack
    - Most efficient way

```
CALL
SP←SP -1, M[SP] ← PC
PC←EA

RTN
PC ←M[SP]
SP ←SP + 1
```

Program interrupt

The concept of program interrupt is used to handle a variety of problems that arise out of normal program sequence. Program interrupt refers to the transfer of program control from a currently running program to another service program as a result of an external or internal generated request. Control returns to the original program after the service program is executed. After a program has been interrupted and the service routine been executed, the CPU must return to exactly the same state that it was when the interrupt occurred. Only if this happens will the interrupted program be able to resume exactly as if nothing had happened. The state of the CPU at the end of the execute cycle (when the interrupt is recognized)is determined from:

1. The content of the program counter
2. The content of all processor registers
3. The content of certain status conditions

The interrupt facility allows the running program to proceed until the input or output device sets its ready flag. Whenever a flag is set to 1, the computer completes the execution of the instruction in progress and then acknowledges the interrupt. The result of this action is that the retune address is stared in location 0. The instruction in location 1 is then performed; this initiates a service routine for the input or output transfer. The service routine can be stored in location 1. The service routine must have instructions to perform the following tasks:

1. Save contents of processor registers.
2. Check which flag is set.
3. Service the device whose flag is set.
4. Restore contents of processor registers.
5. Turn the interrupt facility on.
6. Return to the running program.

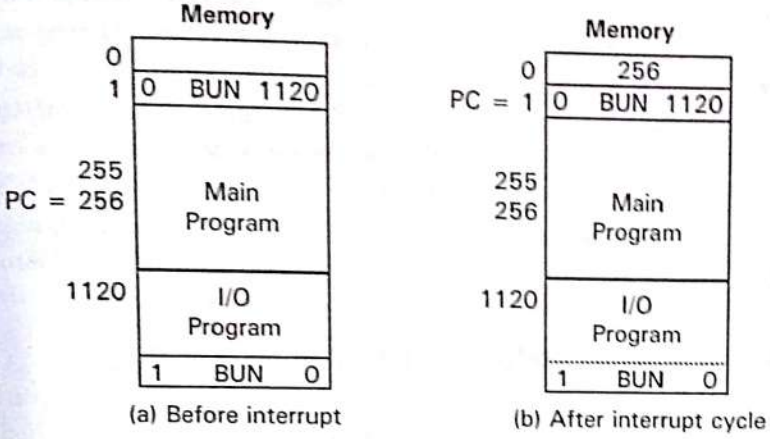


Figure 6.7: *Interrupt cycle*

Types of interrupts

There are three major types of interrupts that cause a break in the normal execution of a program. They can be classified as:

1. External interrupts
2. Internal interrupts
3. Software interrupts

1. External interrupts:

External interrupts come from input-output (I/0) devices, from a timing device, from a circuit monitoring the power supply, or from any other external source. Examples that cause external interrupts are I/0 device requesting transfer of data, I/o device finished transfer of data, elapsed time of an event, or power failure. Timeout interrupt may result from a program that is in an endless loop and thus exceeded its time allocation. Power failure interrupt may have as its service routine a program that transfers the complete state of the CPU into a nondestructive memory in the few milliseconds before power ceases. External interrupts are asynchronous. External interrupts depend on external conditions that are independent of the program being executed at the time.

2. Internal interrupts:

Internal interrupts arise from illegal or erroneous use of an instruction or data. Internal interrupts are also called traps. Examples of interrupts caused by internal error conditions are register overflow, attempt to divide by zero, an invalid operation code, stack overflow, and protection violation. These error conditions usually occur as a result of a premature termination of the instruction execution. The service program that processes the internal interrupt determines the corrective measure to betaken. Internal interrupts are synchronous with the program. If the program is rerun, the internal interrupts will occur in the same place each time.

3. Software interrupts:

A software interrupt is a special call instruction that behaves like an interrupt rather than a subroutine call. It can be used by the programmer to initiate an interrupt procedure at any

desired point in the program. The most common use of software interrupt is associated with a supervisor call instruction. This instruction provides means for switching from a CPU user mode to the supervisor mode. Certain operations in the computer may be assigned to the supervisor mode only, as for example, a complex input or output transfer procedure. A program written by a user must run in the user mode. When an input or output transfer is required, the supervisor mode is requested by means of a supervisor call instruction. This instruction causes a software interrupt that stores the old CPU state and brings in a new PSW that belongs to the supervisor mode. The calling program must pass information to the operating system in order to specify the particular task requested.

Reverse Polish Notation (RPN) with appropriate example.

The postfix RPN notation, referred to as Reverse Polish Notation (RPN), places the operator after the operands. The following examples demonstrate the three representations:

A+B Infix notation

+AB Prefix or Polish notation

AB+ Postfix or reverse Polishnotation

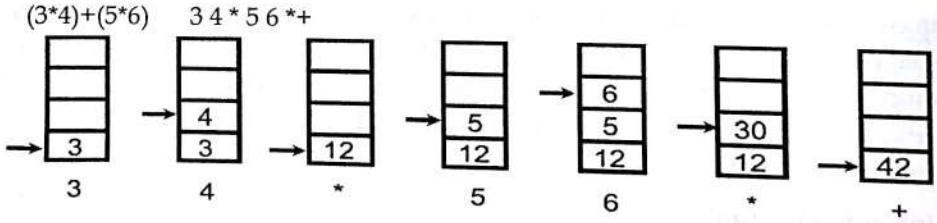
The reverse Polish notation is in a form suitable for stack manipulation. The expression

A * B + C * D is written in reverse Polish notation as A B * C D * +

The conversion from infix notation to reverse Polish notation must take into consideration the operational hierarchy adopted for in fix notation. This hierarchy dictates that we first perform all arithmetic inside inner parentheses, then inside outer parentheses, and do multiplication and division operations before addition and subtraction operations.

Evaluation of Arithmetic Expressions

- Any arithmetic expression can be expressed in parenthesis-free Polish notation, including reverse Polish notation
- (3*4)+(5*6) 3 4 * 5 6 *+



6.7 RISC AND CISC

Complex Instruction Set Computer (CISC)

Computers with many instructions and addressing modes came to be known as Complex Instruction Set Computers (CISC).One goal for CISC machines was to have a machine language instruction to match each high-level language statement type so that job of compiler writer becomes easy. Characteristics of CISC computers are:

- The large number of instructions and addressing modes led CISC machines to have variable length instruction formats

- Multiple operand instructions could specify different addressing modes for each operand
- Variable length instructions greatly complicate the fetch and decode problem for a processor
- They have instructions that act directly on memory addresses due to which multiple memory cycle are needed for executing instructions.
- Microprogrammed control is used rather than hardwired

Reduced Instruction Set Computer (RISC)

Reduced Instruction Set Computers (RISC) were proposed as an alternative. The underlying idea behind RISC processors is to simplify the instruction set and reduce instruction execution time. Characteristics of RISC are:

- Few instructions
- Few addressing modes
- Only load and store instructions access memory
- All other operations are done using on-processor registers
- Fixed length instructions
- Single cycle execution of instructions
- The control unit is hardwired, not microprogrammed

Register Overlapped Windows

The procedure (function) call/return is the most time-consuming operations in typical HLL programs. The depth of procedure activation is within a relatively narrow range. If we use multiple small sets of registers (windows), each assigned to a different procedure, a procedure call automatically switches the CPU to use a different window of registers, rather than saving registers in memory. Windows for adjacent procedures are overlapped to allow parameter passing.

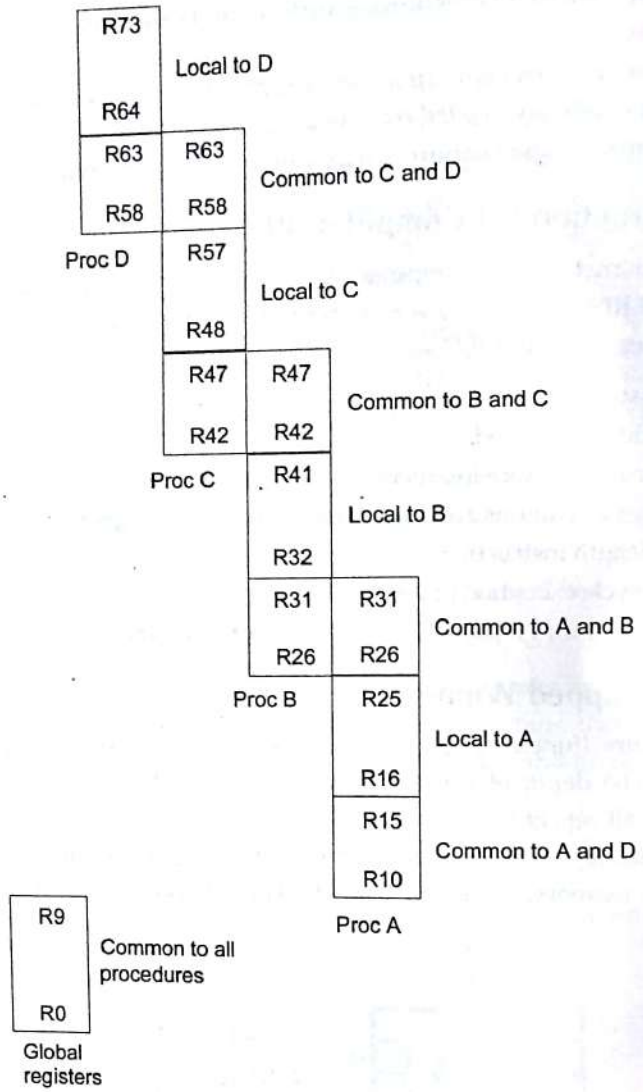


Figure 6.8: *Overlapped register window*

There are three classes of registers:

- Global Registers, G
  - Available to all functions
- Window local registers, L
  - Variables local to the function
- Window shared registers, C
  - Permit data to be shared without actually needing to copy it

Window size, $W = L + 2C + G$

Total number of registers = $(L + C)\,W + G$

Only one register window is active at a time. The active register window is indicated by a pointer. When a function is called, a new register window is activated. This is done by incrementing the pointer. When a function calls a new function, the high numbered registers of the calling function window are shared with the called function as the low numbered registers in its register window. This way the caller's high and the called function's low registers overlap and can be used to pass parameters and results

The advantage of overlapped register windows is that the processor does not have to push registers on a stack to save values and to pass parameters when there is a function call.

This saves

- Accesses to memory to access the stack.
- The cost of copying the register contents at all

And, since function calls and returns are so common, this results in a significant savings relative to a stack-based approach

Chapter 7

Pipelining and Vector Processing

Objective

In this chapter the concept of pipelining is explained and the way it can be speed-up is illustrated with several examples. Both arithmetic and instruction pipeline is considered. It is shown how RISC processors can achieve single-cycle instruction execution by using an efficient instruction pipeline together with the delayed load and delayed branch techniques.

OUTLINE

7.1 Parallel Processing
7.2 Pipelining
7.3 Arithmetic Pipeline
7.4 Instruction Pipeline
7.5 RISC Pipeline
7.6 Vector Processing
7.7 Array Processors

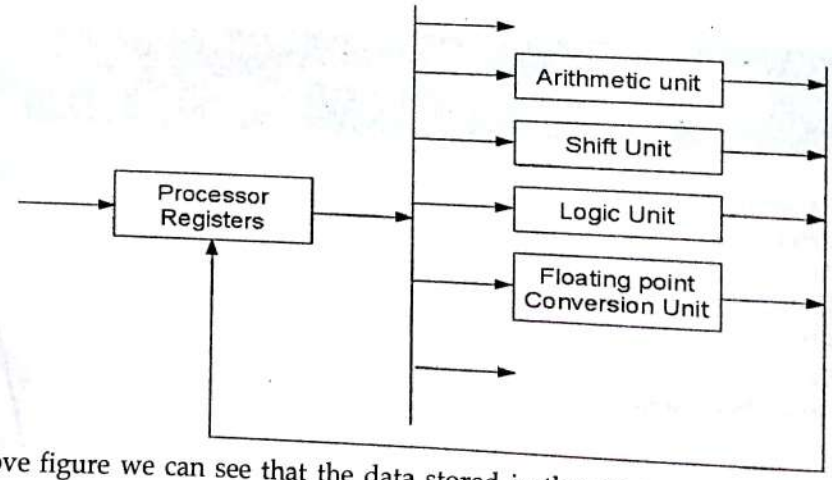
7.1 PARALLEL PROCESSING

The term parallel processing indicates that the system is able to perform several operations in a single time. Now we will elaborate the scenario, in a CPU we will be having only one Accumulator which will be storing the results obtained from the current operation. Now if we are giving only one command such that "a+b" then the CPU performs the operation and stores the result in the accumulator. Now we are talking about parallel processing, therefore we will be issuing two instructions "a+b" and "c-d" in the same time, now if the result of "a+b" operation is stored in the accumulator, then "c-d" result cannot be stored in the accumulator in the same time. Therefore the term parallel processing in not only based on the Arithmetic, logic or shift operations. The above problem can be solved in the following manner. Consider the registers R1 and R2 which will be storing the operands before operation and R3 is the register which will be storing the results after the operations. Now the above two instructions "a+b" and "c-d" will be done in parallel as follows.

- Values of "a" and "b" are fetched in to the registers R1 andR2
- The values of R1 and R2 will be sent into the ALU unit to perform the addition
- The result will be stored in the Accumulator
- When the ALU unit is performing the calculation, the next data "c" and "d" are brought into R1 and R2.
- Finally the value of Accumulator obtained from "a+b" will be transferred into theR3
- Next the values of C and D from R1 and R2 will be brought into the ALU to perform the "c-d" operation.
- Since the accumulator value of the previous operation is present in R3, the result of "c-d" can be safely stored in the Accumulator.

This is the process of parallel processing of only one CPU. Consider several such CPU performing the calculations separately. This is the concept of parallel processing.

Fig 7.1: *Concept of parallel processing*



In the above figure we can see that the data stored in the processor registers is being sent to separate devices basing on the operation needed on the data. If the data inside the processor registers is requesting for an arithmetic operation, then the data will be sent to the arithmetic unit and if in the same time another data is requested in the logic unit, then the data will be sent

to logic unit for logical operations. Now, in the same time, both arithmetic operations and logical operations are executing in parallel. This is called as parallel processing.

Instruction Stream: The sequence of instructions read from the memory is called as an Instruction Stream

Data Stream: The operations performed on the data in the processor is called as a Data Stream.

The computers are classified into 4 types based on the Instruction Stream and Data Stream. They are called as the Flynn's Classification of computers.

Flynn's Classification of Computers:

- Single Instruction Stream and Single Data Stream(SISD)
- Single Instruction Stream and Multiple Data Stream(SIMD)
- Multiple Instruction Stream and Single Data Stream(MISD)
- Multiple Instruction Stream and Multiple Data Stream(MIMD)

SISD 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 capabilities. Parallel processing in this case may be achieved by means of multiple functional units or by pipeline processing.

SIMD represents an organization that includes many processing units under the supervision of a common control unit. All processors receive the same instruction from the control unit but operate on different items of data. The shared memory unit must contain multiple modules so that it can communicate with all the processors simultaneously.

MISD structure is only of theoretical interest since no practical system has been constructed using this organization because Multiple instruction streams means more no of instructions, therefore we have to perform multiple instructions on same data at a time. This is practically impossible.

MIMD structure refers to a computer system capable of processing several programs at the same time operating on different data.

7.2 PIPELINING

Pipelining is a technique of decomposing a sequential process into sub operations, with each sub process being executed in a special dedicated segment that operates concurrently with all other segments. We can consider the pipelining concept as a collection of several segments of data processing programs which will be processing the data and sending the results to the next segment until the end of the processing is reached. We can visualize the concept of pipelining in the example below.

Consider the following operation: Result=(A+B)*C

- First the A and B values are Fetched which is nothing but a "Fetch Operation".
- The result of the Fetch operations is given as input to the Addition operation, which is an Arithmetic operation.

- The result of the Arithmetic operation is again given to the Data operand C which is fetched from the memory and using another arithmetic operation which is Multiplication in this scenario is executed.
- Finally the Result is again stored in the "Result" variable.

In this process we are using up-to 5 pipelines which are the

→ Fetch Operation (A) | FetchOperation(B) |

Addition of (A & B) | Fetch Operation(C) | Multiplication of ((A+B), C) |

Load ((A+B)*C),Result);

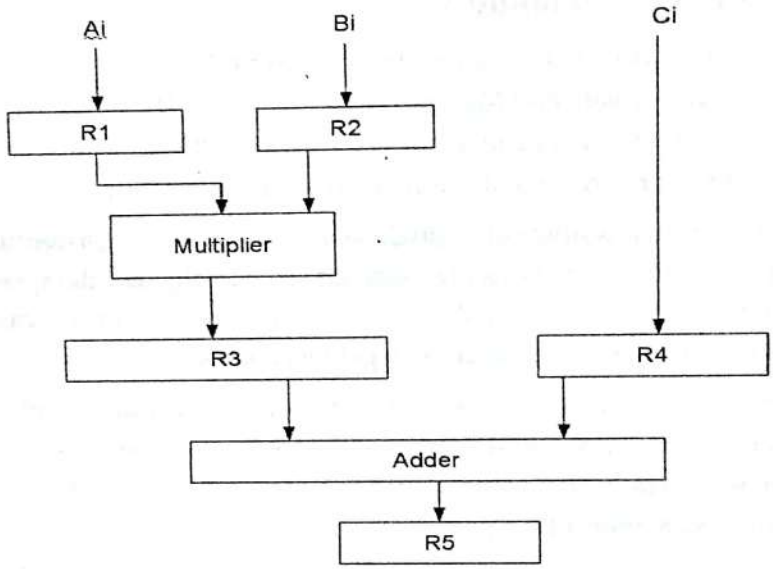


Fig. 7.2: *Example of pipeline processing*

The contents of the Registers in the above pipeline concept are given below. We are considering the implementation of A[7] array with B[7] array.

Table 7.1: *Content of registers in pipeline example*

Clock Pulse Number	Segment1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A1	B1	-	-	-
2	A2	B2	A1*B1	C1	-
3	A3	B3	A2*B2	C2	A1*B1+C1
4	A4	B4	A3*B3	C3	A2*B2+C2
5	A5	B5	A4*B4	C4	A3*B3+C3
6	A6	B6	A5*B5	C5	A4*B4+C4
7	A7	B7	A6*B6	C6	A5*B5+C5
8			A7*B7	C7	A6*B6+C6
9					A7*B7+C7

If the above concept is executed with out the pipelining, then each data operation will be tak 5 cycles, totally they are 35 cycles of CPU are needed to perform the operation. But if are us

the concept of pipeline, we will be cutting off many cycles. Like given in the table below when the values of A1 and B1 are coming into the registers R1 and R2, the registers R3, R4 and R5 are empty. Now in the second cycle the multiplication of A1 and B1 is transferred to register R3, now in this point the contents of the register R1 and R2 are empty. Therefore the next two values A2 and B2 can be brought into the registers. Again in the third cycle after fetching the C1 value the operation (A1*B1)+C1 will be performed. So in this way we can achieve the total concept in only 9 cycles. Here we are assuming that the clock cycle timing is fixed. This is the concept of pipelining.

Below is the diagram of 4 segment pipeline.

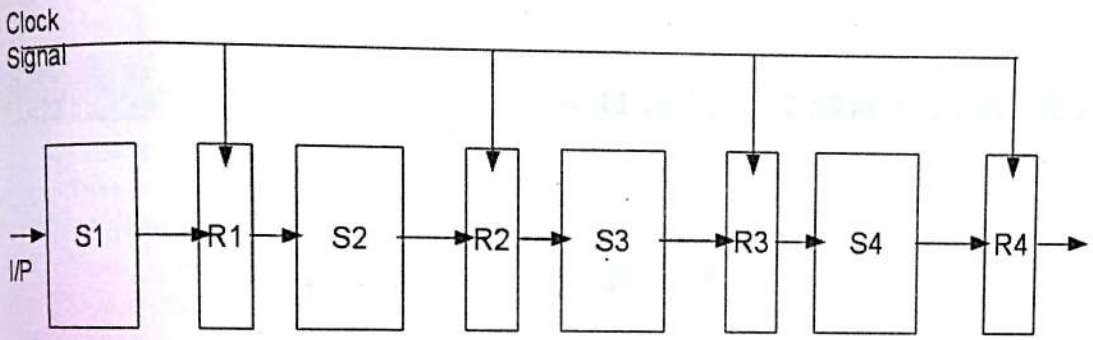


Figure 7.3: *Segment pipeline*

The below table is the space time diagram for the execution of 6 tasks in the 4 segment pipeline.

Seg/ clock	C1	C2	C3	C4	C5	C6	C7	C8	C9
S1	T1	T2	T3	T4	T5	T6			
S2		T1	T2	T3	T4	T5	T6		
S3			T1	T2	T3	T4	T5	T6	
S4				T1	T2	T3	T4	T5	T6

Figure 7.4: *Space and time diagram*

Speedup

Now, consider the case where -k segment pipeline with clock cycle time t_p is used to execute n tasks. The first task T_1 requires a time equal to kt_p to complete its operation since there re k segments in the pipe. The remaining n-1 tasks emerge from the pipe t the rate of one task per clock cycle and they will be completed after a time equal to $(n - 1)t_p$. Therefore, to complete n-tasks using a k-segment pipeline requires k + (n – 1) clock cycles. For example, the diagram of Fig. 7.4. shows four segments and six tasks. The time required for n tasks is nt_n. The speedup of pipeline processing over an equivalent non-pipeline processing is defined by the ratio.

$$S = \frac{nt_n}{(k + n - 1)t_p}$$

As the number of tasks increases, n becomes much larger than k - 1, nd k + n – 1 approaches the value of n. Under this condition, the speedup becomes,

$$S = \frac{t_n}{t_p}$$

If we assume that the time it takes to process a task is the same in the pipeline and non-pipeline circuits, we will have $t_n = kt_p$. Including this assumption, the speedup reduces to

$$S = \frac{kt_p}{t_p} = k$$

7.3 ARITHMETIC PIPELINE

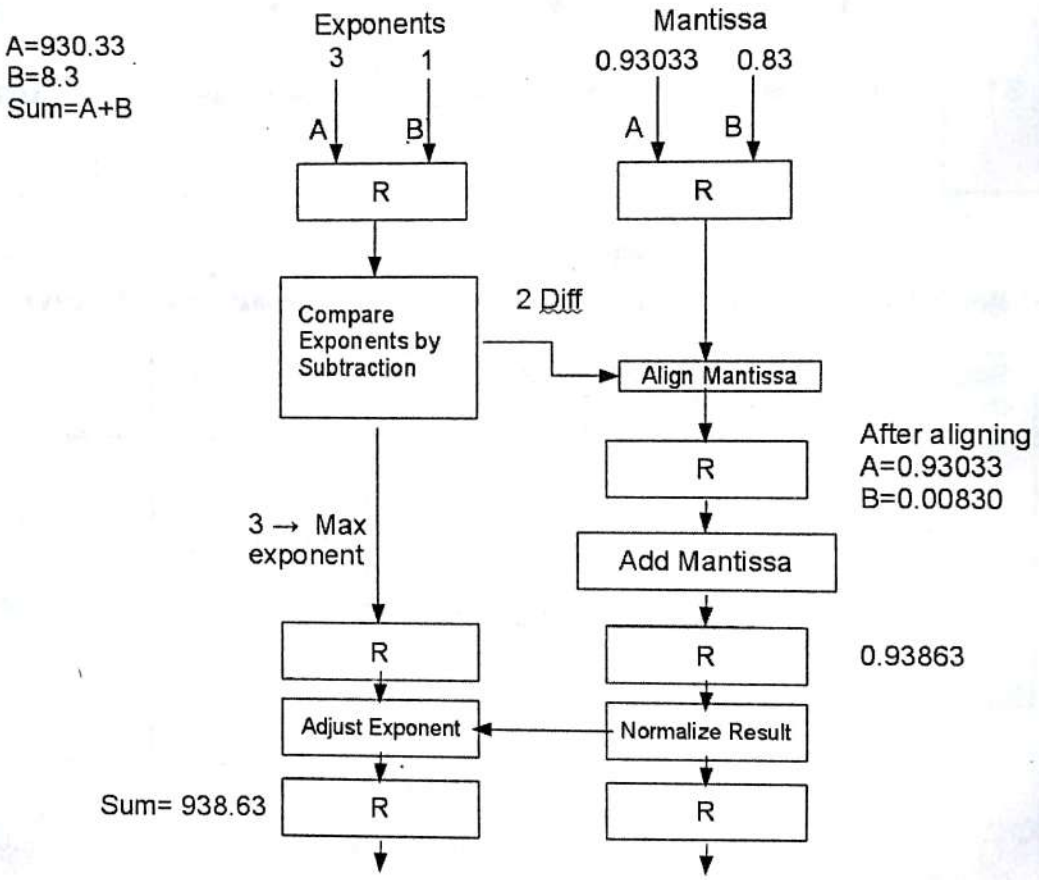


Figure 7.5: *Arithmetic pipelining*

The above diagram represents the implementation of arithmetic pipeline in the area of floating point arithmetic operations. In the diagram, we can see that two numbers A and B are added together. Now the values of A and B are not normalized, therefore we must normalize them before start to do any operations. The first thing is we have to fetch the values of A and B into the registers. Here R denote a set of registers. After that the values of A and B are normalized, therefore the values of the exponents will be compared in the comparator. After that the alignment of mantissa will be taking place. Finally, we will be performing addition, since an addition is happening in the adder circuit. The source registers will be free and the second set of

values can be brought. Like wise when the normalizing of the result is taking place, addition of the new values will be added in the adder circuit and when addition is going on, the new data values will be brought into the registers in the start of the implementation. We can see how the addition is being performed in the diagram.

7.4 INSTRUCTION PIPELINE

Pipelining concept is not only limited to the data stream, but can also be applied on the instruction stream. The instruction pipeline execution will be like the queue execution. In the queue the data that is entered first, will be the data first retrieved. Therefore when an instruction is first coming, the instruction will be placed in the queue and will be executed in the system. Finally the results will be passing on to the next instruction in the queue. This scenario is called as Instruction pipelining. The instruction cycle is given below

- Fetch the instruction from the memory
- Decode the instruction
- calculate the effective address
- Fetch the operands from the memory
- Execute the instruction
- Store the result in the proper place.

In a computer system each and every instruction need not necessary to execute all the above phases. In a Register addressing mode, there is no need of the effective address calculation. Below is the example of the four segment instruction pipeline.

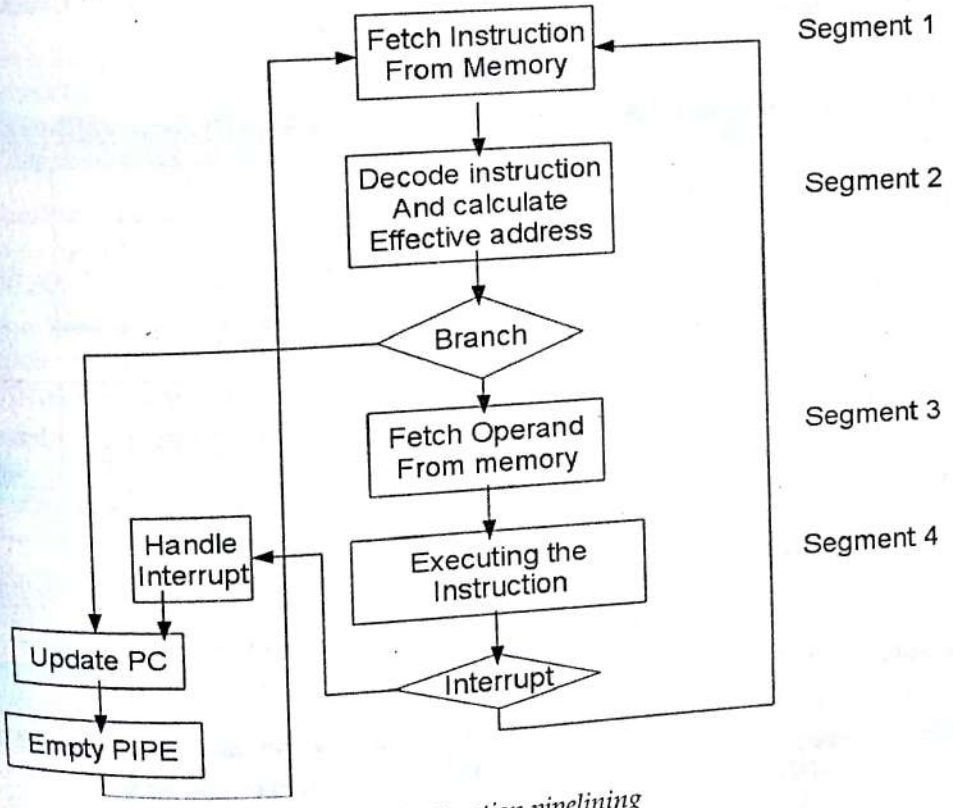


Figure 7.6: *Instruction pipelining*

In the above diagram we can see that the instruction which is first executing has to be fetched from the memory, there after we are decoding the instruction and we are calculating the effective address. Now we have two ways to execute the instruction. Suppose we are using a normal instruction like ADD, then the operands for that instruction will be fetched and the instruction will be executed. Suppose we are executing an instruction such as Fetch command. The fetch command itself has internally three more commands which are like ACTDR, ARTDR etc.., therefore we have to jump to that particular location to execute the command, so we are using the branch operation. So in a branch operation, again other instructions will be executed. That means we will be updating the PC value such that the instruction can be executed. Suppose we are fetching the operands to perform the original operation such as ADD, we need to fetch the data. The data can be fetched in two ways, either from the main memory or else from an input output devices. Therefore in order to use the input output devices, the devices must generate the interrupts which should be handled by the CPU. Therefore the handling of interrupts is also a kind of program execution. Therefore we again have to start from the starting of the program and execute the interrupt cycle.

The different instruction cycles are given below:

- FI → FI is a segment that fetches an instruction
- DA → DA is a segment that decodes the instruction and identifies the effective address.
- FO → FO is a segment that fetches the operand.
- EX → EX is a segment that executes the instruction with the operand.

Figure 7.7: *Timing of instruction pipeline*

FI → Fetch Instruction
FO → Fetch Operand
DA → Decode instruction and Fetch Effective Address
EX → Execute the Instruction

Step	1	2	3	4	5	6	7	8	9	10	11	12	13
1	FI	DA	FO	EX									
2		FI	DA	FO	EX								
3			FI	DA	FO	EX							
4				FI	-	-	FI	DA	FO	EX			
5					-	-	-	FI	DA	FO	EX		
6									FI	DA	FO	EX	
7										FI	DA	FO	EX

Pipelining Conflicts

There are different conflicts that are caused by using the pipeline concept. They are

Resource conflicts

These are caused by access to memory by two or more segments at the same time. Most of these conflicts can be resolved by using separate instruction and data memories

Data dependency

These conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available.

Branch difficulties

These difficulties arise from branch and other instructions that change the value of PC.

Data Dependency Conflict

The data dependency conflict can be solved by using the following methods.

- **Hardware Interlocks:** The most straight forward method is to insert hardware interlocks. An interlock is a circuit that detects instructions whose source operands are destination of instructions farther up in the pipeline. Detection of this situation causes the instruction whose source is not available to be delayed by enough clock cycles to resolve the conflict. This approach maintains the program sequence by using hardware to insert the required delay.
- **Operand Forwarding:** Another technique called operand forwarding uses special hardware to detect a conflict and avoid the conflict path by using a special path to forward the values between the pipeline segments.
- **Delayed Load:** The delayed load operation is nothing but when executing an instruction in the pipeline, simply delay the execution starting of the instruction such that all the data that is needed for the instruction can be successfully updated before execution.

Branch Conflicts

The following are the solutions for solving the branch conflicts that are obtained in the pipelining concept.

- **Pre-fetch Target Instruction:** In this the branch instructions which are to be executed are pre-fetched to detect if any errors are present in the branch before execution.
- **Branch Target Buffer:** BTB is the associative memory implementation of the branch conditions.
- **Loopbuffer:**Theloopbufferisaveryhighspeedmemorydevice.Wheneveraloopistobeexecuted in the computer. The complete loop will be transferred in to the loop buffer memory and will be executed as in the cache memory.
- **Branch Prediction:** The use of branch prediction is such that, before a branch is to be executed, the instructions along with the error checking conditions are checked. Therefore we will not be going into any unnecessary branch loops.
- **Delayed Branch:** The delayed branch concept is same as the delayed load process in which we are delaying the execution of a branch process, before all the data is fetched by the system for beginning the CPU.

7.5 RISC PIPELINE

The ability to use the instruction pipelining concept in the RISC architecture is very efficient. The simplicity of the instruction set can be utilized to implement an instruction pipeline using a small number of sub operations, with each being executed in one clock cycle. Due to fixed length instruction format, the decoding of the operation can occur at the same time as the register selection. Since the arithmetic, logic and shift operations are done on register basis,

there is no need for extra fetching or effective address decoding steps to perform the operation. So pipelining concept can be effectively used in this scenario. Therefore the total operations can be categorized as one segment will be fetching the instruction from program memory, the other segment executes the instruction in the ALU and the third segment may be used to store the result of the ALU operation in a destination register. The data transfer instructions in RISC are limited to only Load and Store instructions. To prevent conflicts in data transfer, we will be using two separate buses one for storing the instructions and other for storing thedata.

Example of three segment instruction pipeline:

We want to perform a operation in which there is some arithmetic, logic or shift operations. Therefore as per the instruction cycle, we will be having the following steps:

- I: Instruction Fetch
- A: ALU Operation
- E: Execute Instruction.

The I segment will be fetching the instruction from program memory. The instruction is decoded and an ALU operation is performed in the A segment. In the A segment the ALU operation instruction will be fetched and the effective address will be retrieved and finally in the E segment the instruction will be executed.

Delayed Load

Consider the following instructions:

1. LOAD: R1 ← M[address1]
2. LOAD: R2 ← M[address2]
3. ADD: R3 ← R1 +R2
4. STORE: M[address 3] ←R3

The below tables will be showing the pipelining concept with the data conflict and without data conflict.

Clock Cycles	1	2	3	4	5	6
1.Load R1	I	A	E			
2.Load R2		I	A	E		
3. Add R1+R2			I	A	E	
4. Store R3				I	A	E

Clock Cycles	1	2	3	4	5	6	7
1.Load R1	I	A	E				
2.Load R2		I	A	E			
3. No Operation			I	A	E		
4. Add R1+R2				I	A	E	
5. Store R3					I	A	E

Figure 7.8: *Three-segment pipeline timing*

7.6 VECTOR PROCESSING

Normal computational systems are not enough in some special processing requirements. Such as, in special processing systems like artificial intelligence systems and some weather forecasting systems, terrain analysis, the normal systems are not sufficient. In such systems the data processing will be involving on very high amount of data, we can classify the large data as a very big arrays. Now if we want to process this data, naturally we will need new methods of data processing. The vectors are considered as the large one dimensional array of data. The term vector processing involves the data processing on the vectors of such large data.

The vector processing system can be understand by the example below. Consider a program which is adding two arrays A and B of length 100;

Machine level program

```
Initialize I=0
20   Read A(I) ReadB(I)
Store C(I)=A(I)+B(I)
Increment I=I+1
If I<=100 go to 20 continue
```

So in this above program we can see that the two arrays are being added in a loop format. First we are starting from the value of 0 and then we are continuing the loop with the addition operation until the I value has reached to 100. In the above program there are 5 loop statements which will be executing 100 times. Therefore the total cycles of the CPU taken is 500 cycles. But if we use the concept of vector processing then we can reduce the unnecessary fetch cycles, since the fetch cycles are used in the creation of the vector. The same program written in the vector processing statement is given below.

Operation Code	Base Address SRC 1	Base Address SRC 2	Base Address DST	Vector length

C(1:100)=A(1:100)+B(1:100)

In the above statement, when the system is creating a vector like this the original source values are fetched from the memory into the vector, therefore the data is readily available in the vector. So when a operation is initiated on the data, naturally the operation will be performed directly on the data and will not wait for the fetch cycle. So the total no of CPU Cycles taken by the above instruction is only 100.

Instruction format of Vector Instruction

Below we can see the implementation of the vector processing concept on the following matrix multiplication. In the matrix multiplication, we will be multiplying the row of A matrix with the column of the B matrix elements individually finally we will be adding the results.

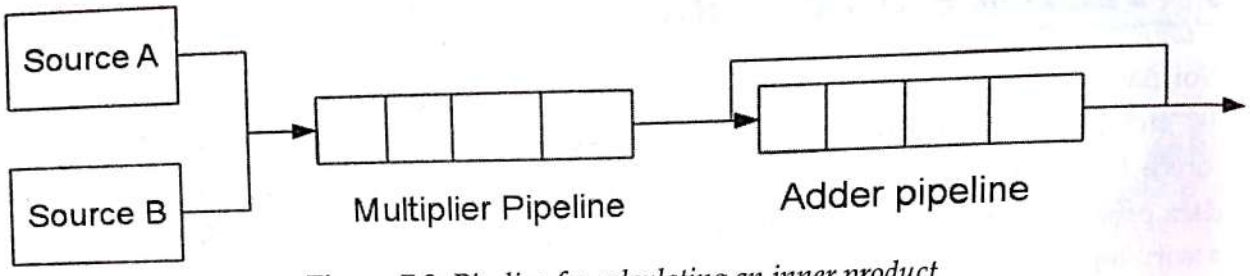


Figure 7.9: *Pipeline for calculating an inner product*

In the above diagram we can see that how the values of A vector and B Vector which represents the matrix are being multiplied. Here we will be considering a 4x4 matrix A and B. Now the from the source A vector we will be taking the first 4 values and will be sending to the multiplier pipeline along with the 4 values from the vector B. The resultant 1 value is stored in the adder pipeline. Like wise remaining values from a row and column multiplication will be brought into the adder pipeline, which will be performing the addition of all the things finally we will have the result of one row to column multiplication. When addition operation is taking place in the adder pipeline the next set of values will be brought into the multiplier pipeline, so that all the operations can be performed simultaneously using the parallel processing concepts by the implementation of pipeline.

Memory Interleaving

Pipelining and vector processing naturally requires the several data elements for processing. So instead of using the same memory and selecting one at a time, we will be using several modules of the memory such that we can have separate data for each processing unit. As we can see in the above in the diagram each memory array is designed independently of the next memory array. Such that when the data needed for a operation is stored in the first memory array, another data for another operation can be safely stored in the next memory array, so that the operations can be performed concurrently. This process is called as memory interleaving.

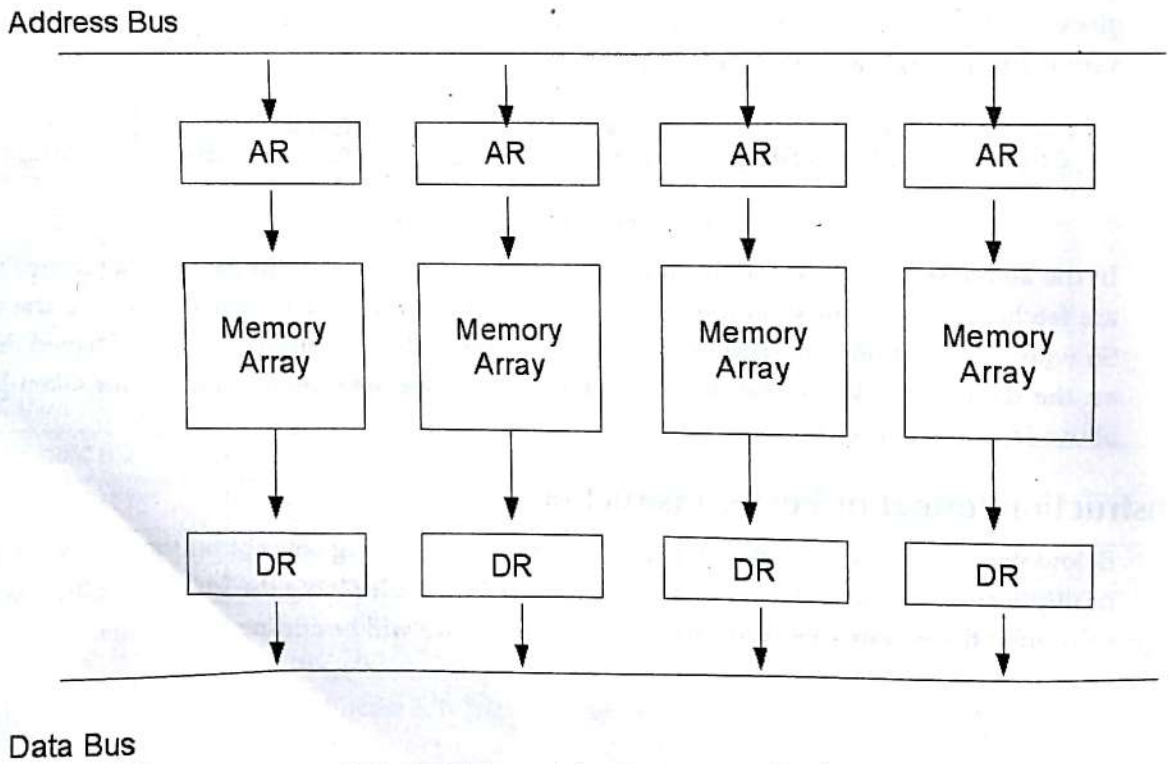


Figure 7.10: *Multiple modules memory organization*

7.7 ARRAY PROCESSORS

In a distributed computing we will be having several computers working on the same task such that their processing power will be shared among all the systems so that they can perform the task fast. But the disadvantage of the distributed computing is that we have to give separate resources for each system and every system need to be controlled by a task initiating system or can be called as a central control unit. The management of this kind of systems is very hard. In order to perform a specific operation involving a large processing there is no need of distributed computing. The alternate for this kind of scenarios is array processors or attached array processors. The simplest is the SIMD Attached array processor.

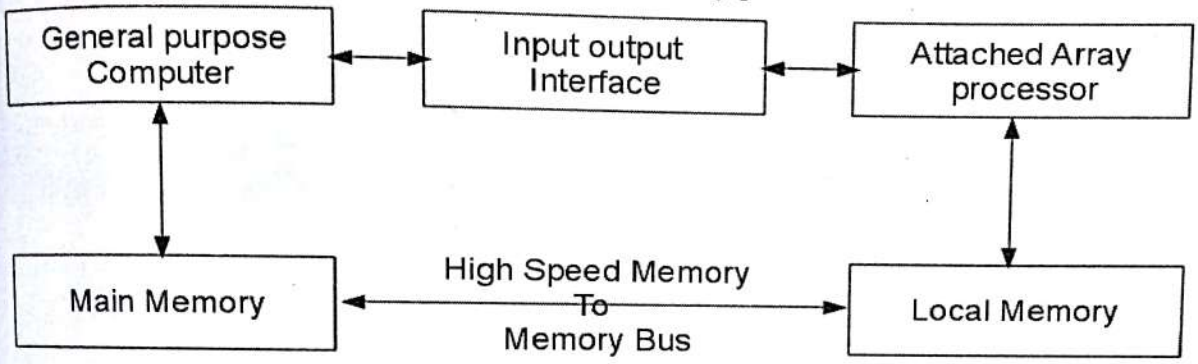


Figure 7.11: *Attached Array processor with host computer*

The above diagram shows that the system is attached a separate processor which will be used for operation specific purpose. If the array processor is designed for solving floating point arithmetic, then it will only perform that operations. The detailed figure of the attached array processor is given in the diagram below. This will be having the SIMD architecture. In this we will be having a master control unit which will be coordinating all the process in the array processor. Each processing unit in the array processor is having a local memory unit as in the memory interleaving concept on which it performs the operations. Finally we will be having a main memory in which the original source data and the results that are obtained from the array processor will be stored. This is the working principle of the SIMD array processor technology.

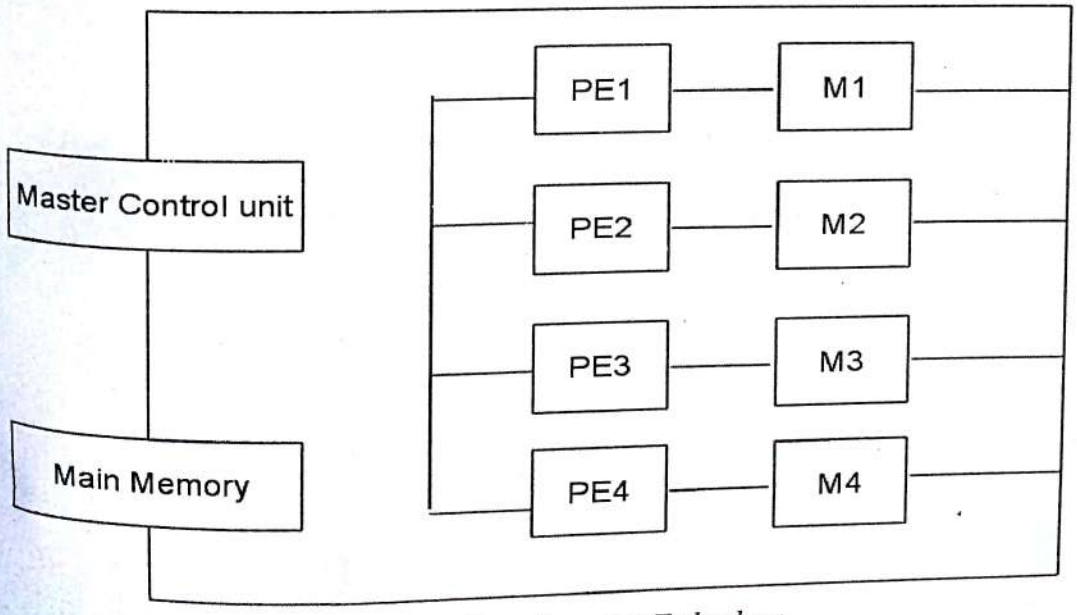


Figure 7.12: *SIMD Array Processor Technology*

Chapter 8

COMPUTER ARITHMETIC

Objective

This chapter presents arithmetic algorithms for addition, subtraction, multiplication, and division and shows the procedure for implementing them with digital hardware. Procedures are developed for signed-magnitude and signed 2's complement fixed point numbers.

OUTLINE

8.1 Introduction
8.2 Addition and Subtraction
8.3 Multiplication Algorithms

8.1 INTRODUCTION

Data is manipulated by using the arithmetic instructions in digital computers. Data is manipulated to produce results necessary to give solution for the computation problems. The Addition, subtraction, multiplication and division are the four basic arithmetic operations. If we want then we can derive other operations by using these four operations.

To execute arithmetic operations there is a separate section called arithmetic processing unit in central processing unit. The arithmetic instructions are performed generally on binary or decimal data. Fixed-point numbers are used to represent integers or fractions. We can have signed or unsigned negative numbers. Fixed-point addition is the simplest arithmetic operation. If we want to solve a problem then we use a sequence of well-defined steps. These steps are collectively called algorithm. To solve various problems we give algorithms.

In order to solve the computational problems, arithmetic instructions are used in digital computers that manipulate data. These instructions perform arithmetic calculations.

And these instructions perform a great activity in processing data in a digital computer. As we already stated that with the four basic arithmetic operations addition, subtraction, multiplication and division, it is possible to derive other arithmetic operations and solve scientific problems by means of numerical analysis methods.

A processor has an arithmetic processor (as a sub part of it) that executes arithmetic operations. The data type, assumed to reside in processor, registers during the execution of an arithmetic instruction. Negative numbers may be in a signed magnitude or signed complement representation. There are three ways of representing negative fixed point - binary numbers signed magnitude, signed 1's complement or signed 2's complement. Most computers use the signed magnitude representation for the mantissa.

8.2 ADDITION AND SUBTRACTION

Addition and Subtraction with Signed –Magnitude Data

We designate the magnitude of the two numbers by A and B. Where the signed numbers are added or subtracted, we find that there are eight different conditions to consider, depending on the sign of the numbers and the operation performed. These conditions are listed in the first column of Table 4.1. The other columns in the table show the actual operation to be performed with the magnitude of the numbers. The last column is needed to present a negative zero. In other words, when two equal numbers are subtracted, the result should be +0 not -0.

The algorithms for addition and subtraction are derived from the table and can be stated as follows (the words parentheses should be used for the subtraction algorithm)

Signed Magnitude Addition and Subtraction

Table 8.1: *Addition and subtraction of signed magnitude numbers*

Operation	Add Magnitudes	Subtract Magnitudes		
		When $A > B$	When $A < B$	When $A = B$
$(+A) + (+B)$	$+(A + B)$			
$(+A) + (-B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(-A) + (+B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$
$(-A) + (-B)$	$-(A + B)$			
$(+A) - (+B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(+A) - (-B)$	$+(A + B)$			
$(-A) - (+B)$	$-(A + B)$			
$(-A) - (-B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$

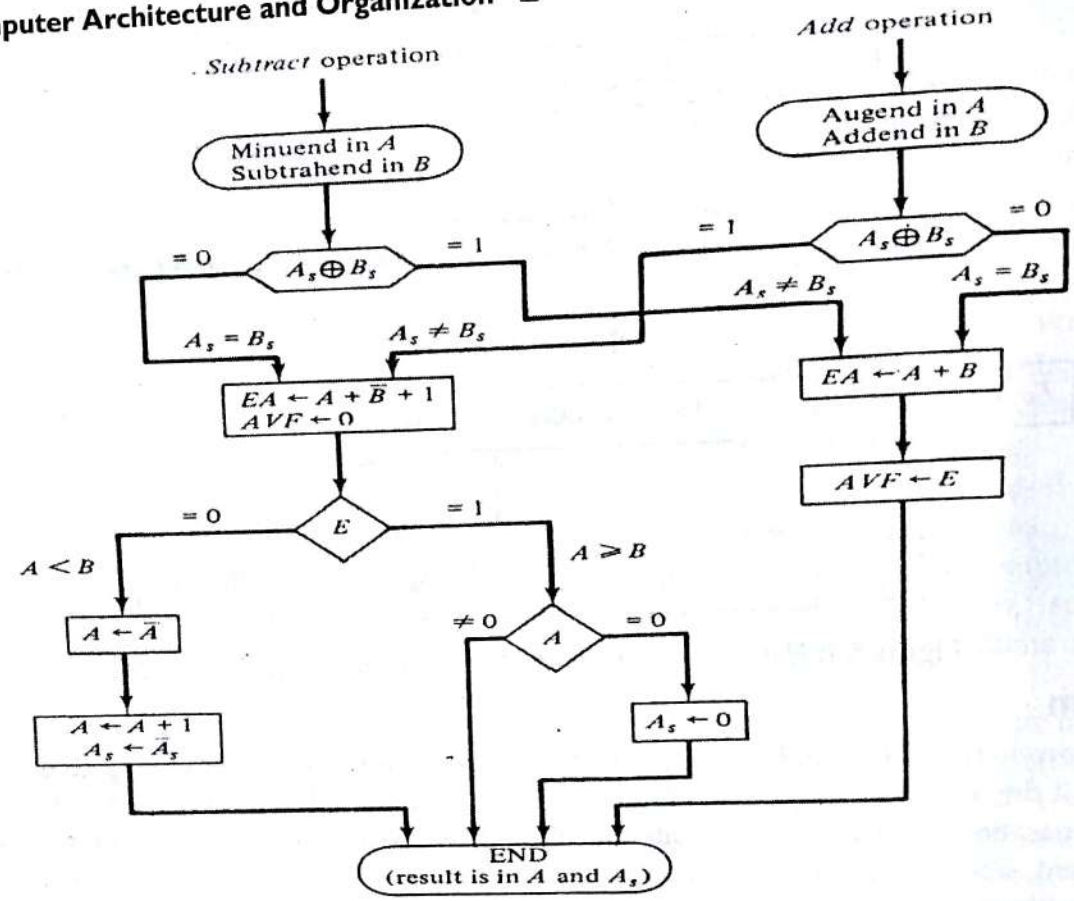


Figure 8.2: *Flowchart for add and subtract operations*

Signed 2's Complement Addition and Subtraction

The algorithm for adding and subtracting two binary numbers in signed-2's complement representation is shown in the flowchart. The sum is obtained by adding the contents of AC and BR (including their sign bits). The overflow bit V is set to 1 if the exclusive-OR of the last two carries is 1, and it is cleared to 0 otherwise. The subtraction operation is accomplished by adding the content of AC to the 2's complement of BR. Taking the 2's complement of BR has the effect of changing a positive number to negative, and vice versa. An overflow must be checked during this operation because the two numbers added could have the same sign. The programmer must realize that if an overflow occurs, there will be an erroneous result int he AC register.

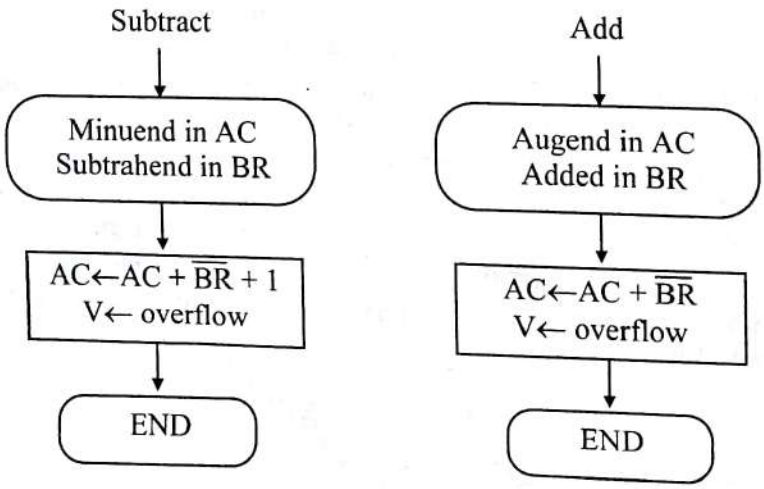


Table 8.3: *Algorithm for adding and subtracting numbers in signed-2's complement representation*

8.3 MULTIPLICATION ALGORITHM

In the beginning, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in Bs and Qs respectively. We compare the signs of both A and Q and set to corresponding sign of the product since a double-length product will be stored in registers A and Q. Registers A and E are cleared and the sequence counter SC is set to the number of bits of the multiplier. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consist of n-1 bits.

Now, the low order bit of the multiplier in Qn is tested. If it is 1, the multiplicand (B) is added to present partial product (A), 0 otherwise. Register EAQ is then shifted once to the right to form the new partial product. The sequence counter is decremented by 1 and its new value checked. If it is not equal to zero, the process is repeated and a new partial product is formed. When SC = 0 we stops the process.

Figure 8.4: *Hardware for multiply operation*

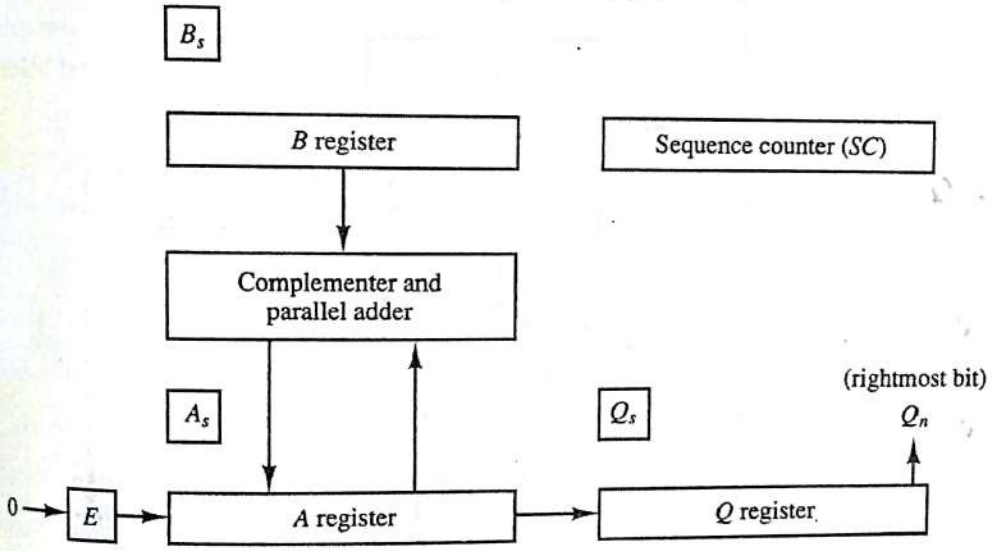


Figure 8.5: *Flowchart for multiply operation*

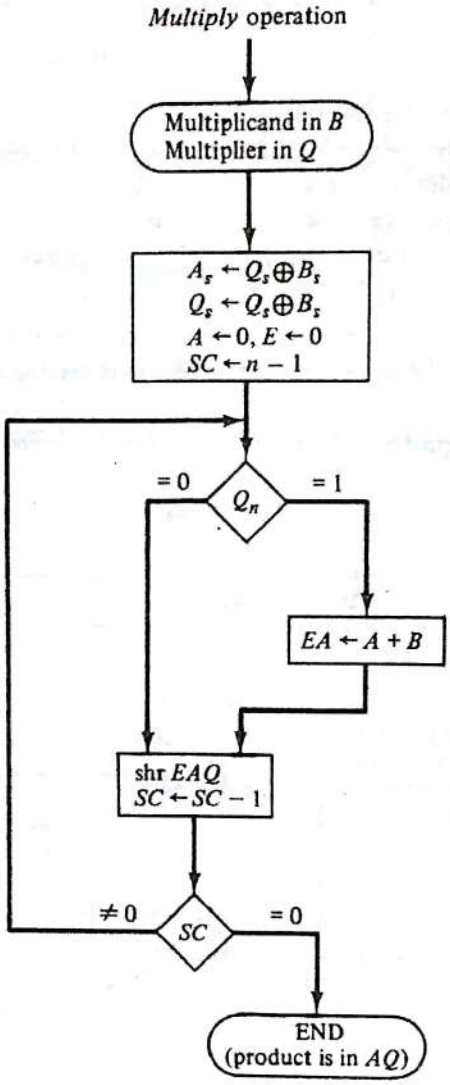


Table 8.2: *Numerical example for binary multiplier*

Multiplic and B = 10111	E	A	Q	SC
Multiplier in Q	0	00000	10011	101
Q_n = 1; add B		10111		
First partial product	0	10111		
Shift right EAQ	0	01011	11001	100
Q_n = 1; add B		10111		
Second partial product	1	00010		
Shift right EAQ	0	10001	01100	011
Q_n = 0; shift right EAQ	0	01000	10110	010
Q_n = 0; shift right EAQ	0	00100	01011	001
Q_n = 1; add B		1011		
Fifth partial product	0	11011		
Shift right EAQ	0	01101	10101	000
Final product in AQ = 0110110101				

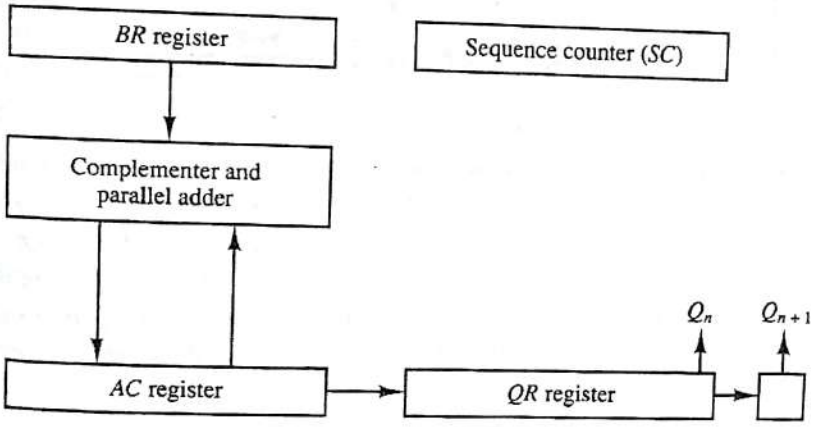
Booth's algorithm

Booth algorithm gives a procedure for multiplying binary integers in signed- 2's complement representation. It operates on the fact that strings of 0's in the multiplier require no addition but just shifting, and a string of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as $2^{k+1} - 2^m$. For example, the binary number 001110 (+14) has a string 1's from 2^3 to 2^1 (k=3, m=1). The number can be represented as $2^{k+1} - 2^m$. $= 2^4 - 2^1 = 16 - 2 = 14$. Therefore, the multiplication M X 14, where M is the multiplicand and 14 the multiplier, can be done as $M \times 2^4 - M \times 2^1$. Thus the product can be obtained by shifting the binary multiplicand M four times to the left and subtracting M shifted left once.

Hardware for Booth Algorithm

In this algorithm, sign bit are not separated from the rest of the registers. Here, registers A, B and Q are renamed as AC, BR and QR respectively. Also, Q_n designates the least significant bit of the multiplier in the reregister QQR and flip-flop Q_{n+1} is appended to QR to facilitate a doubled bit inspection of the multiplier.

Figure 8.7: *Hardware for booth algorithm*



As in all multiplication schemes, booth algorithm requires examination of the multiplier bits and shifting of partial product. Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial, or left unchanged according to the following rules:

- The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1's in the multiplier.
- The multiplicand is added to the partial product upon encountering the first 0 in a string of 0's in the multiplier.
- The partial product does not change when multiplier bit is identical to the previous multiplier bit.

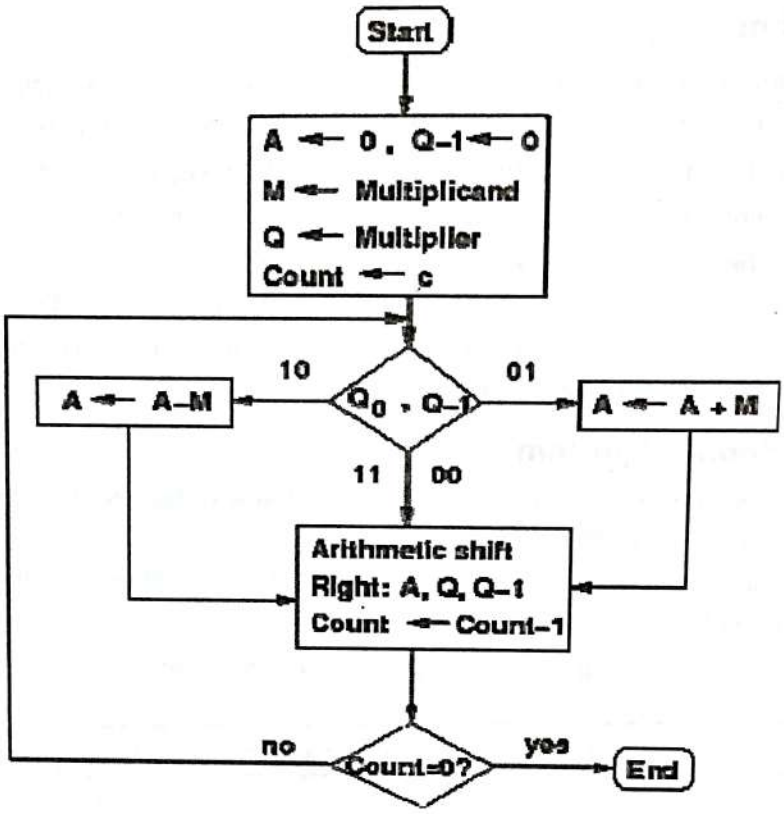


Figure 8.8: *Flowchart of booth algorithm*

The algorithm works for positive or negative multipliers in 2's complement representation. This is because a negative multiplier ends with a string of 1's and the last operation will be a subtraction of the appropriate weight. The two bits of the multiplier in Qn and Qn+1 are inspected. If the two bits are equal to 10, it means that the first 1 in a string of 1 's has been encountered. This requires a subtraction of the multiplicand from the partial product in AC. If the two bits are equal to 01, it means that the first 0 in a string of 0's has been encountered. This requires the addition of the multiplicand to the partial product in AC. When the two bits are equal, the partial product does not change.

A numerical example of Booth algorithm is shown in table 8.3 for n = 5. It shows the step-by-step multiplication of (–9) × (–13) = + 117. Note that the multiplier in QR is negative and that the multiplicand in BR is also negative. The 10-bit product appears in AC and QR and is positive. The final value of Q_{n+1} is the original sign bit of the multiplier and should not be taken as part of the product.

$Q_n Q_{n+1}$	$BR = 10111$ $\overline{BR} + 1 = 01001$	AC	QR	Q_{n+1}	SC
	Initial	00000	10011	0	101
1 0	Subtract BR	01001			
		01001			
	ashr	00100	11001	1	100
1 1	ashr	00010	01100	1	011
0 1	Add BR	10111			
		11001			
	ashr	11100	10110	0	010
0 0	ashr	11110	01011	0	001
1 0	Subtract BR	01001			
		00111			
	ashr	00011	10101	1	000

Table 8.3: *Example of Multiplication with Booth Algorithm*

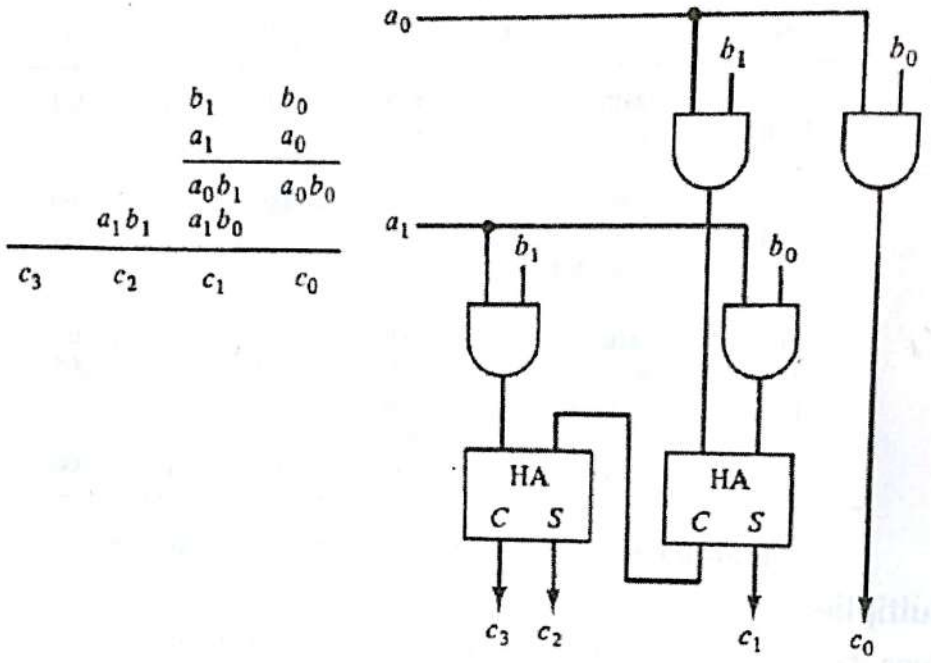
Array Multiplier

Checking the bits of the multiplier one at a time and forming partial products is a sequential operation that requires a sequence of add and shift microoperations. The multiplication of two binary numbers can be done with one micro-operation by means of a combinational circuit that forms the product bits all at once.

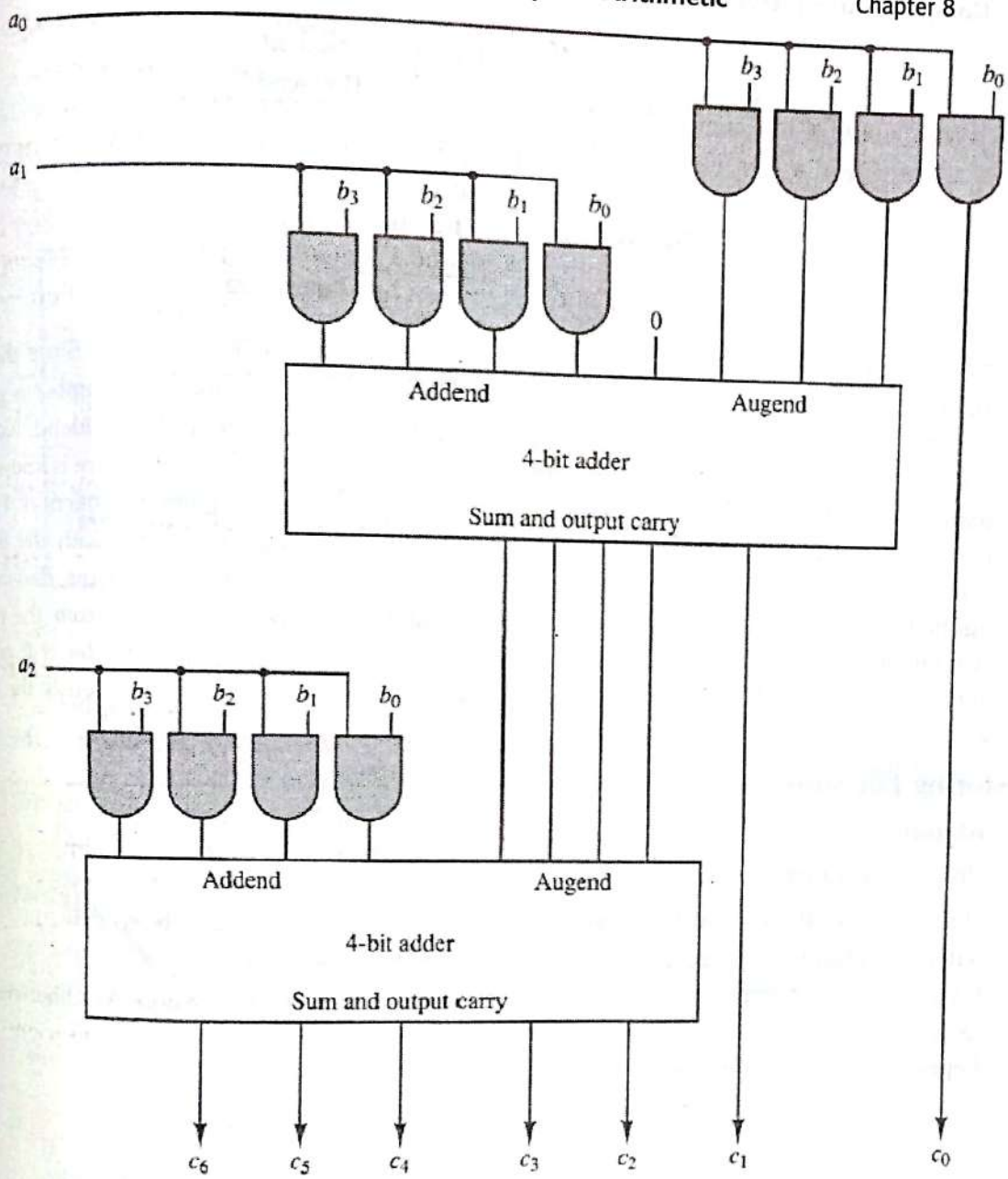
This is a fast way of multiplying two numbers since all it takes is the time for the signals to propagate through the gates that form the multiplication array. However, an array multiplier requires a large number of gates, and for this reason it was not economical until the development of integrated circuits.

To see how an array multiplier can be implemented with a combinational circuit, consider the multiplication of two 2-bit numbers as shown in Fig. 10-9. The multiplicand bits are b1 and bo, the multiplier bits are al and ao, and the product is c3 c2 c1 co. The first partial product is formed by multiplying ao by bi bo. The multiplication of two bits such as ao and bo produces a 1 if both bits are 1; otherwise, it produces a 0. This is identical to an AND operation and can be implemented with an AND gate. As shown in the diagram, the first partial product is formed by means of two AND gates. The second partial product is formed by multiplying a1 by b, bo and is shifted one position to the left. The two partial products are added with two half-adder (HA) circuits. Usually, there are more bits in the partial products and it will be necessary to use full-adders to produce the sum. Note that the least significant bit of the product does not have to go through an adder since it is formed by the output of the first AND gate.

Figure 8.9: *2-bit by 2-bit array multiplier*



A combinational circuit binary multiplier with more bits can be constructed in a similar fashion. A bit of the multiplier is ANDed with each bit of the multiplicand in as many levels as there are bits in the multiplier. The binary output in each level of AND gates is added in parallel with the partial product of the previous level to form a new partial product. The last level produces the product. For j multiplier bits and k multiplicand bits we need j x k AND gates and (j − 1) k-bit adders to produce a product of j + k bits. As a second example, consider a multiplier circuit that multiplies a binary number of four bits with a number of three bits. Let the multiplicand be represented by b_3 b_2 b_i b_o and the multiplier by a_2 a_I a_0. Since k = 4 and j = 3, we need 12 AND gates and two 4-bit adders to produce a product of seven bits. The logic diagram of the multiplier is shown in Figure above.



Figurer 8.10: *4-bit by 3-bit array multiplier*

8.4 Division Algorithms

Division of two fixed-point binary numbers in signed magnitude representation is performed with paper and pencil by a process of successive compare, shift and subtract operations. Binary division is much simpler than decimal division because here the quotient digits are either 0 or 1 and there is no need to estimate how many times the dividend or partial remainder fits into the divisor. The division process is described in Figure

```
                 000010101   Quotient
Divisor  1101 ) 100010010   Dividend
                -1101
                  10000
                 -1101
                    1110
                   -1101
                       1   Remainder
```

The devisor is compared with the five most significant bits of the dividend. Since the 5-bit number is smaller than B, we again repeat the same process. Now the 6-bit number is greater than B, so we place a 1 for the quotient bit in the sixth position above the dividend. Now we shift the divisor once to the right and subtract it from the dividend. The difference is known as a partial remainder because the division could have stopped here to obtain a quotient of 1 and a remainder equal to the partial remainder. Comparing a partial remainder with the divisor continues the process. If the partial remainder is greater than or equal to the divisor, the quotient bit is equal to 1. The divisor is then shifted right and subtracted from the partial remainder. If the partial remainder is smaller than the divisor, the quotient bit is 0 and no subtraction is needed. The divisor is shifted once to the right in any case. Obviously the result gives both a quotient and a remainder.

Restoring Division

Algorithm

Division Operation Steps:

1. Shift A and Q left one binary position.
2. Subtract divisor from A and place answer back in A(A← A – B).
3. If the sign bit of A is 1, set Q_0 to 0 and add divisor back to A (that is, restore A); Otherwise, set Q_0 to 1.
4. Repeat steps 1, 2 and 3 n times.

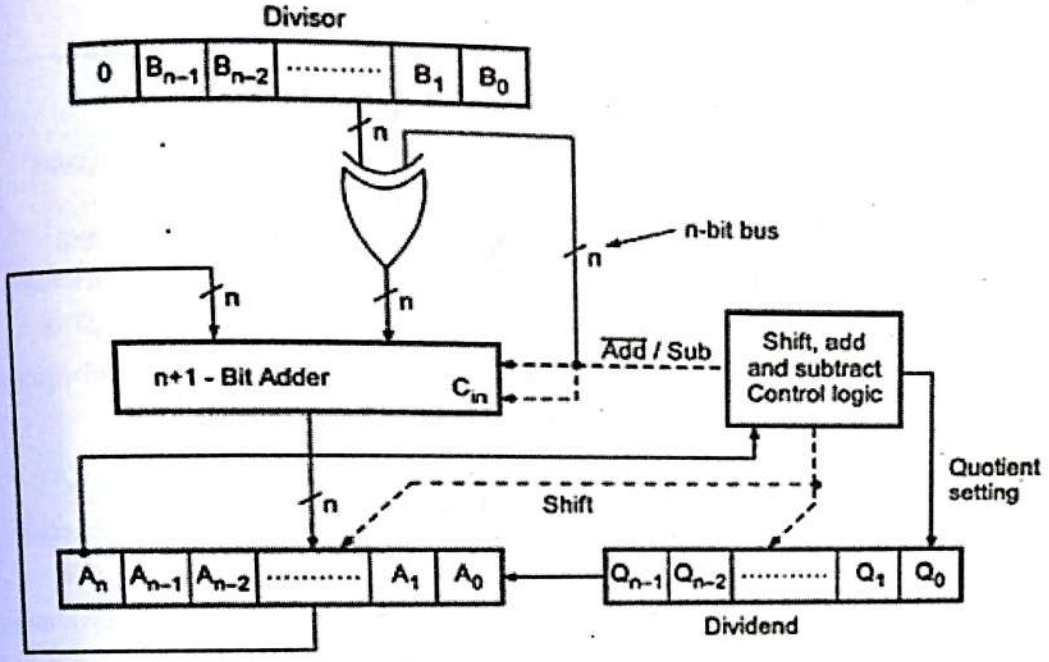


Table 8.11: *Hardware to implement binary division*

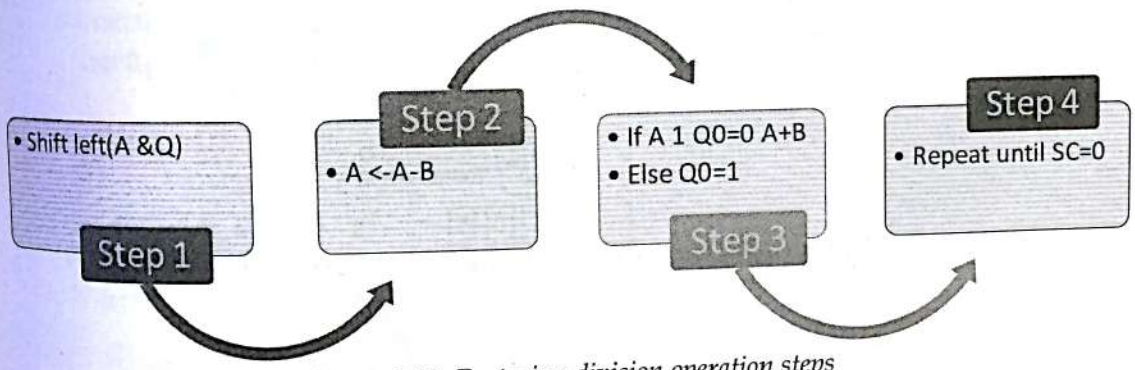


Figure 8.12: *Restoring division operation steps*

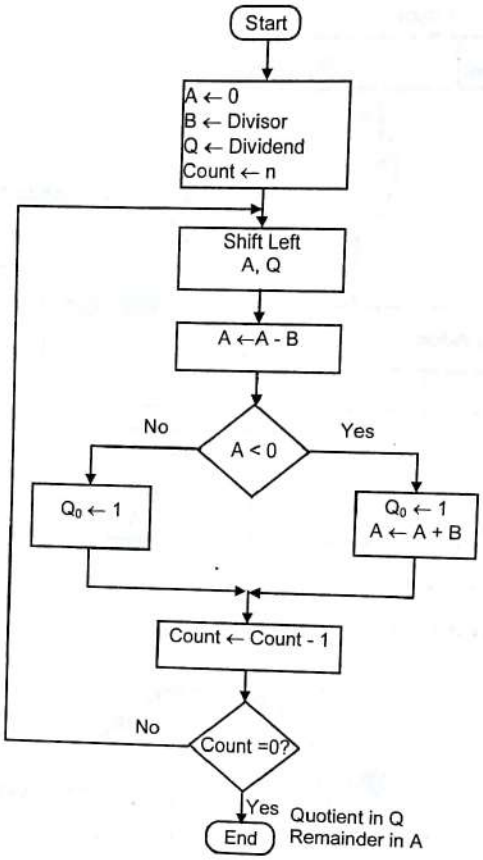


Figure 8.13: *Flowchart for restoring division operation*

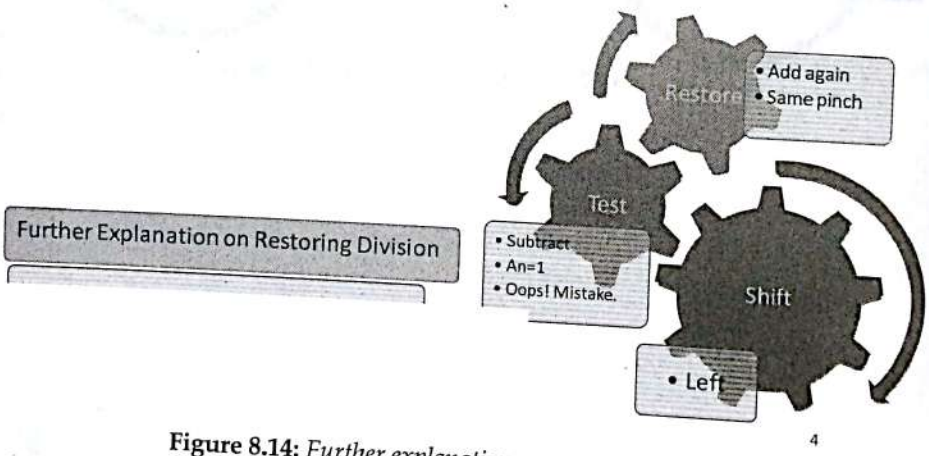


Figure 8.14: *Further explanation on restoring division*

Table 8.4: *Restoring division example*

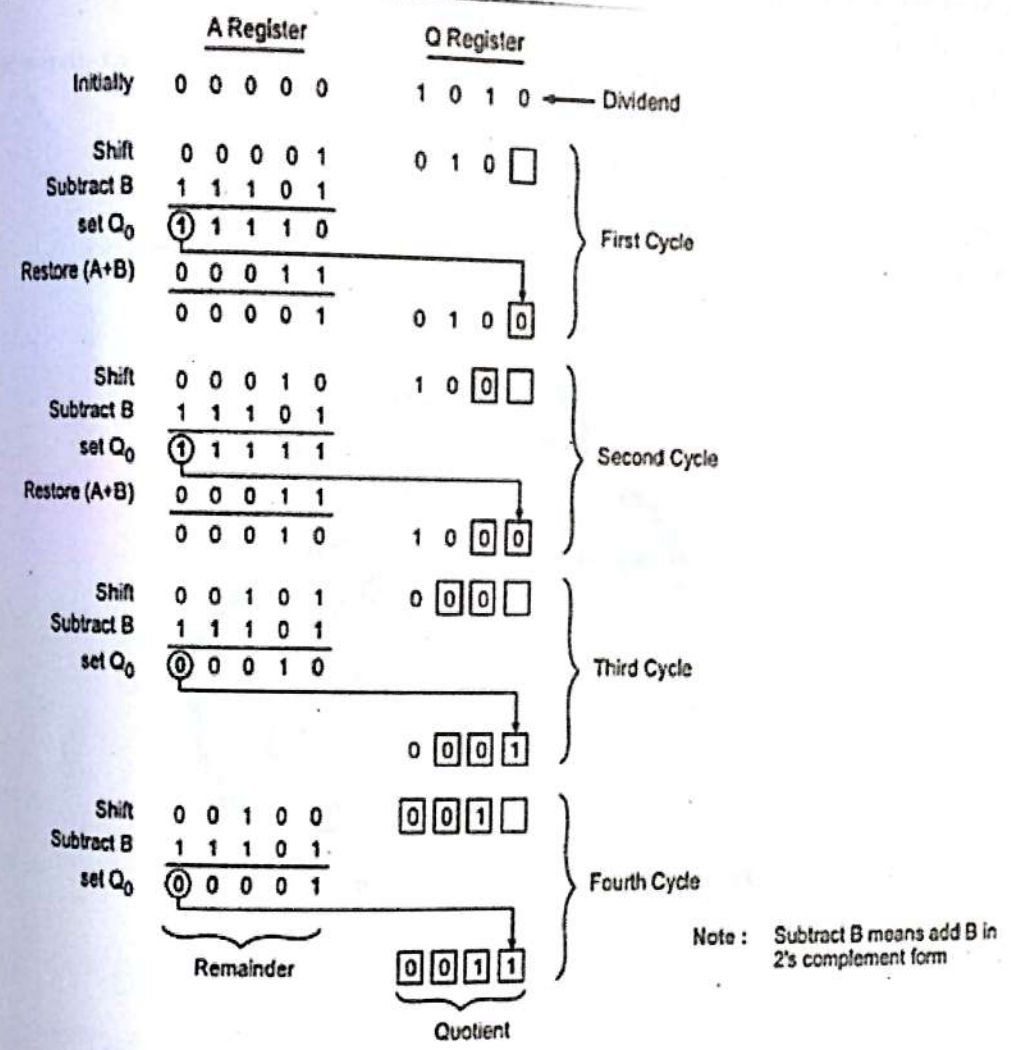


Figure: 8.15: *A restoring division example*

If A is positive	If A is negative
Shift left and subtract divisor→ 2A – B	Restore → A + B Shift left and subtract divisor → 2(A + B) – B = 2A + B

Non-Restoring Division

Algorithm

Step 1: If the sign of A is 0, shift A and Q left one bit position and subtract divisor from A; otherwise, shift A and Q left and add divisor to A.

Step 2: Repeat steps 1 and 2 for n times.

Step 3: If the sign of A is 1, add divisor to A.

Note: *Step 3 is required to leave the proper positive remainder in A at the end of n cycles.*

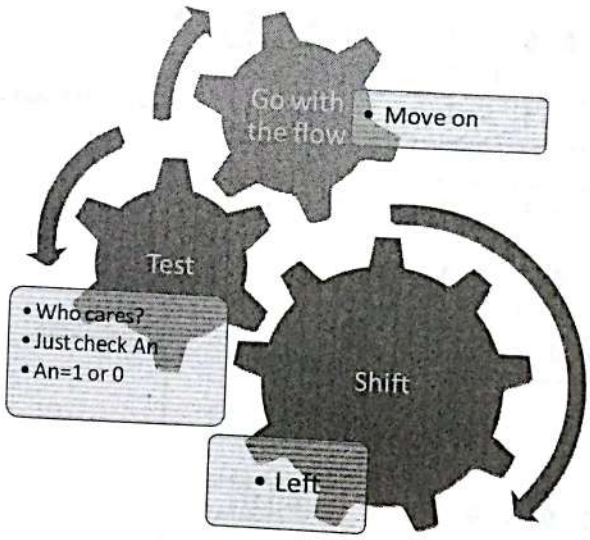


Figure 8.16: *Non-restoring division operation steps*

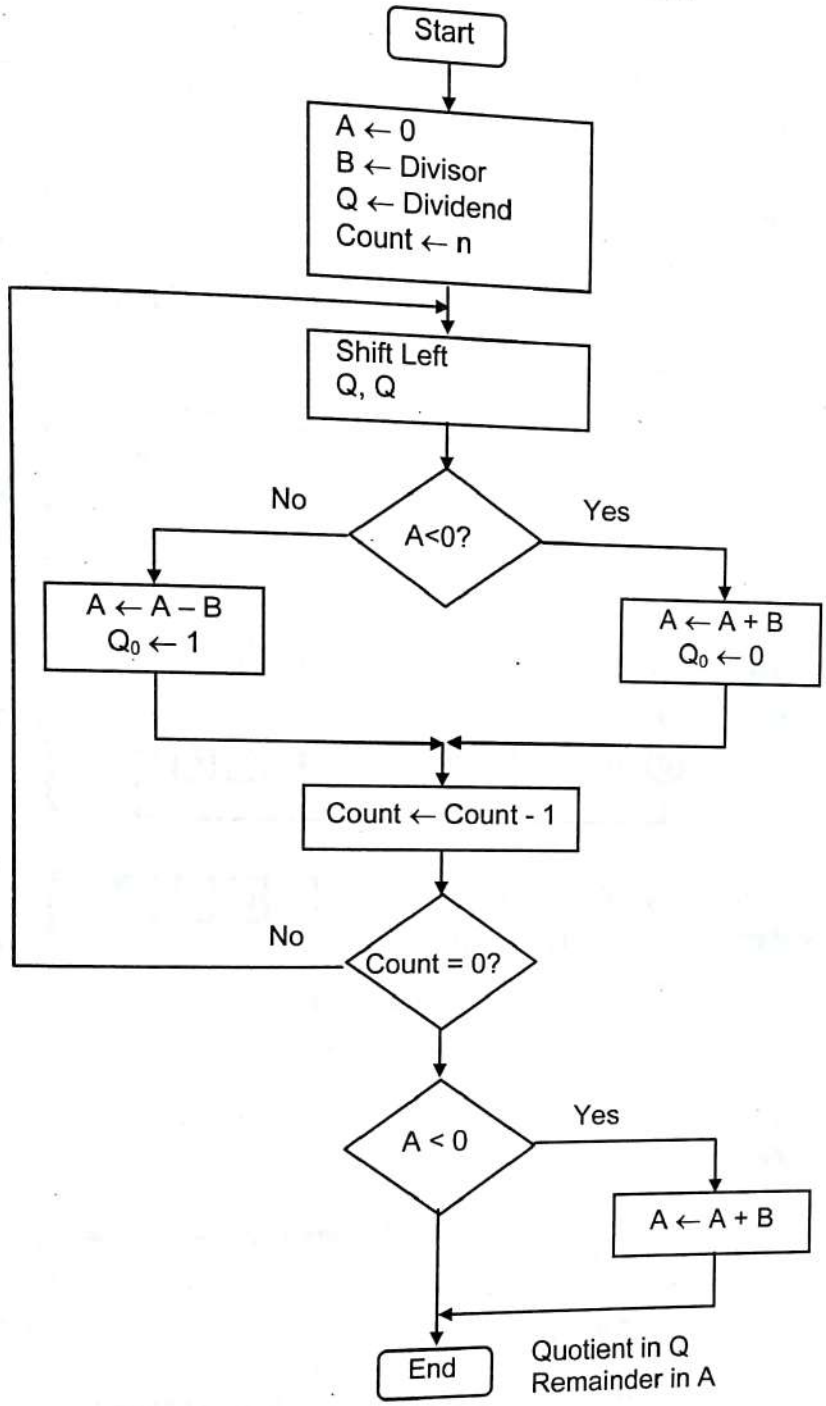


Figure 8.17: *Flowchart for non-restoring division operation*

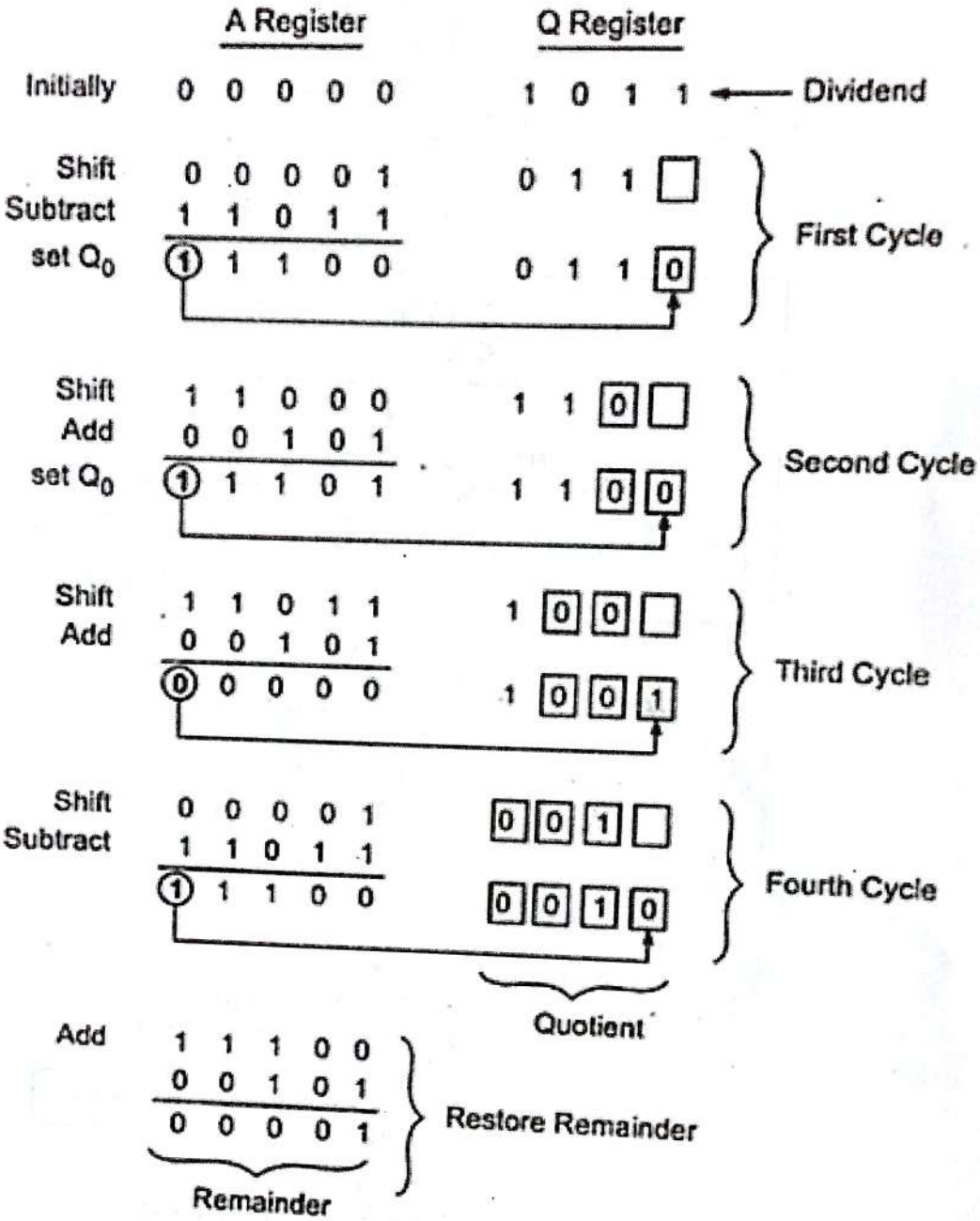


Figure 8.18: *A non-restoring division example*

Chapter 9

Input/Output Organization

Objective

This chapter discusses the techniques that computers use to communicate with input and output devices. Interface units are presented to show the way that the processor interacts with external peripherals. Four modes of transfer are discussed: programmed I/O, interrupt initiated transfer, direct memory access, and the use of input-output processors.

OUTLINE

9.1 Peripheral Devices
9.2 Input-Output Interface
9.3 Modes of Transfer
9.4 Direct Memory Access
9.5 Input-Output Processor
9.6 Serial Communication

9.1 Introduction to Peripheral Devices

The input-output subsystem of a computer, referred to a I/O, provides an efficient mode of communication between the central system and the outside environment. Programs and data must be entered into computer memory for processing and results obtained from computations must be recorded or displayed for the user. A computer services no useful purpose without the ability to received information from an outside source and to transmit results in a meaningful form. Any input/output devices connected to the computer are called peripheral devices.

9.2 Input-Output Interface

Provides a method for transferring information between internal storage (such as memory and CPU registers) and external I/O devices, resolves the differences between the computer and peripheral devices

- Peripherals - Electromechanical Devices, CPU or Memory - Electronic Device
- Data Transfer Rate
  - Peripherals - Usually slower
  - CPU or Memory - Usually faster than peripherals
    - Some kinds of Synchronization mechanism may be needed
- Unit of Information
  - Peripherals – Byte, Block, …
  - CPU or Memory – Word
- Data representations may differ

I/O Bus and Interface Modules

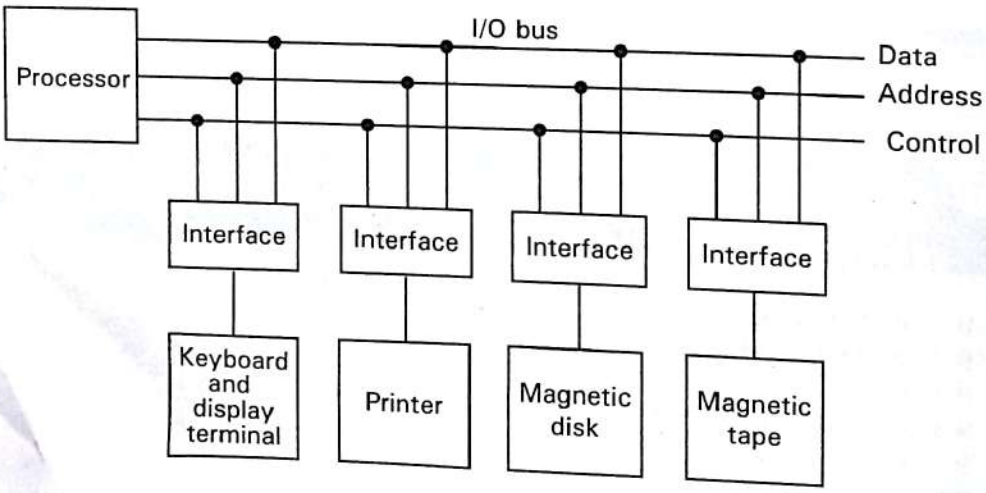


Fig 9.1: *Connection of I/O bus to input-output devices.*

I/O bus from the processor is connected to all peripheral interfaces. To communicate with a particular device, the processor places a device address on the address lines. Each peripheral has an interface module associated with its interface. Functions of an interface are as below:

- Decodes the device address (device code)
- Decodes the I/O commands (operation or function code)
- Provides signals for the peripheral controller
- Synchronizes the data flow and
- Supervises the transfer rate between peripheral and CPU or Memory

Types of I/O command

Control command:-Issued to activate the peripheral and to inform it what to do?

Status command:-Used to check the various status conditions of the interface before a transfer is initiated

Data input command:-Cause the interface to read the data from the peripheral and place it into the interface buffer.

Data output command:-Causes the interface to read the data from the bus and save it into the interface buffer

I/O bus and memory bus

Memory bus is used for information transfers between CPU and the MM (main memory). I/O bus is for information transfers between CPU and I/O devices through their I/O interface

Physical Organizations

Many computers use a common single bus system for both memory and I/O interface units

- Use one common bus but separate control lines for each function
- Use one common bus with common control lines for both functions
- Some computer systems use two separate buses,
- One to communicate with memory and the other with I/O interfaces

I/O Bus

Communication between CPU and all interface units is via a common I/O bus. An interface connected to a peripheral device may have a number of data registers, a control register, and a status register. A command is passed to the peripheral by sending to the appropriate interface register

Isolated vs. Memory mapped I/O

Isolated I/O

- Separate I/O read/write control lines in addition to memory read/write control lines
- Separate (isolated) memory and I/O address spaces
- Distinct input and output instructions

9.3 MODES OF TRANSFER

- Program-Controlled I/O
- Interrupt-Initiated I/O
- Direct Memory Access (DMA)

1. Program-Controlled I/O (Input Device to CPU)

- Continuous CPU involvement
- CPU slowed down to I/O speed
- Simple
- Least hardware

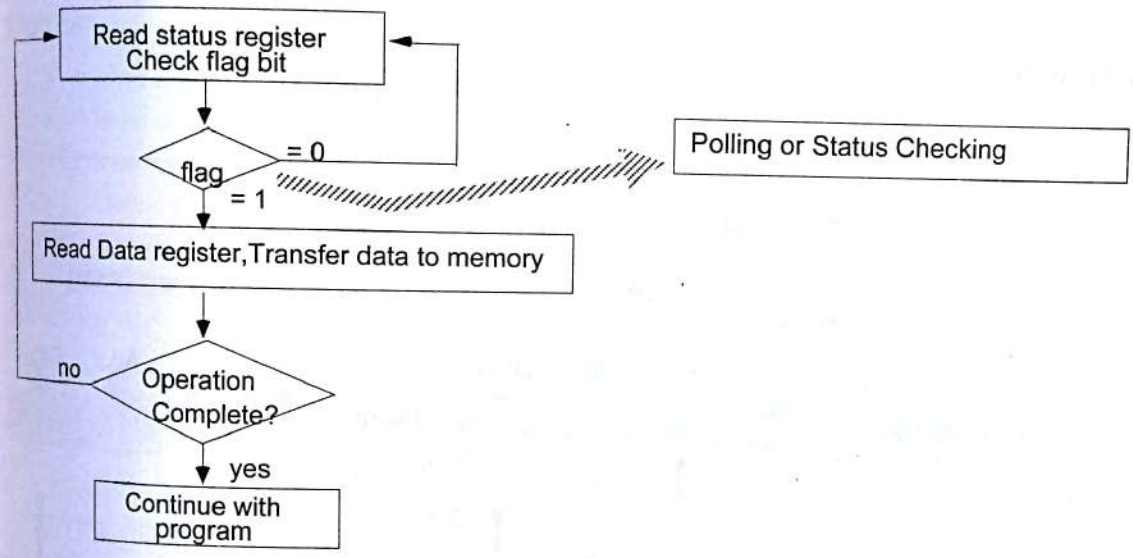


Figure 9.3: *Flowchart of program-controlled IO Communication*

2. Interrupt Initiated I/O

- Polling takes valuable CPU time
- Open communication only when some data has to be passed -> *Interrupt.*
- I/O interface, instead of the CPU, monitors the I/O device
- When the interface determines that the I/O device is ready for data transfer, it generates an *Interrupt Request* to the CPU
- Upon detecting an interrupt, CPU stops momentarily the task it is doing, branches to the service routine to process the data transfer, and then returns to the task it was performing

3. DMA (Direct Memory Access)

Large blocks of data transferred at a high speed to or from high speed devices, magnetic drums, disks, tapes, etc. DMA controller (Interface) provides I/O transfer of data directly to and from the memory and the I/O device. CPU initializes the DMA controller by sending a memory address and the number of words to be transferred. Actual transfer of data is done directly

between the device and memory through DMA controller freeing CPU for other tasks. DMA works by stealing the CPU cycles.

Cycle Stealing

While DMA I/O takes place, CPU is also executing instructions. DMA Controller and CPU both access Memory which causes memory Access Conflict. Memory Bus Controller is responsible for coordinating the activities of all devices requesting memory access by using priority schemes. Memory accesses by CPU and DMA Controller are interwoven; with the top priority given to DMA Controller which is called cycle Stealing. CPU is usually much faster than I/O(DMA), thus CPU uses the most of the memory cycles. DMA Controller steals the memory cycles from CPU for those stolen cycles, CPU remains idle. For those slow CPU, DMA Controller may steal most of the memory cycles which may cause CPU remain idle long time.

DMA Transfer

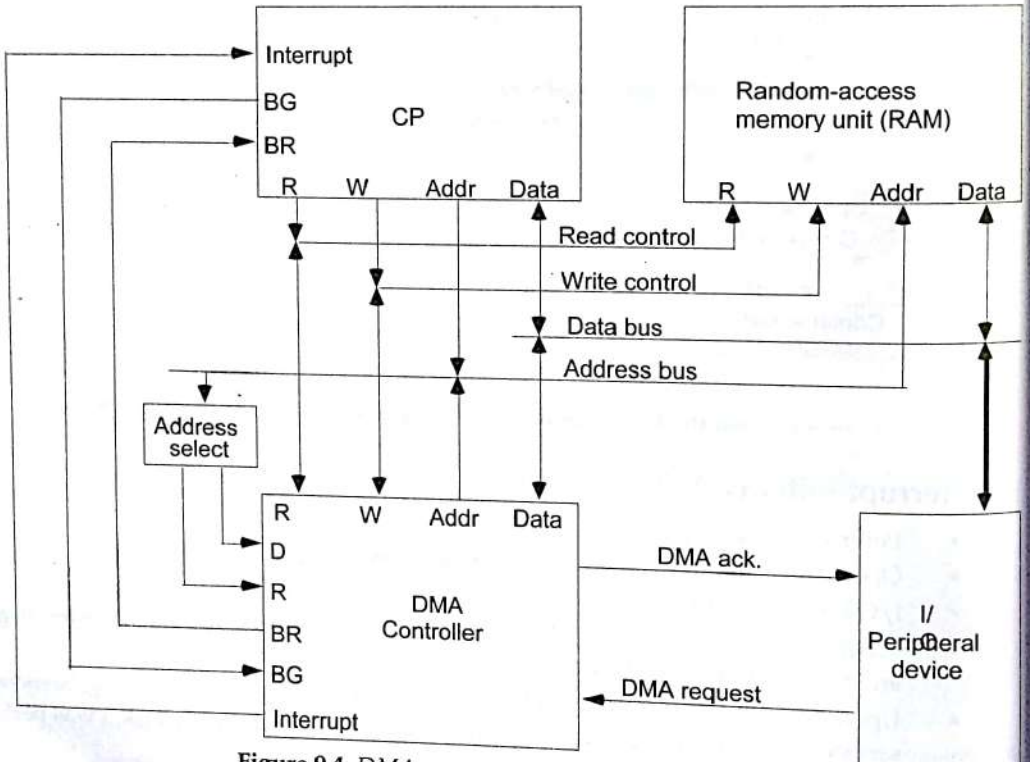


Figure 9.4: *DMA communication block diagram.*

CPU executes instruction to

- Load Memory Address Register
- Load Word Counter
- Load Function(Read or Write) to be performed
- Issue a GO command

Upon receiving a GO Command DMA performs I/O operation as follows independently from CPU

Input

- Send read control signal to Input Device
- DMA controller collects the input from input device byte by bye and assembles the byte into a word until word is full
- Send write control signal to memory
- Increment address register (Address Reg<- Address Reg +1)
- Decrement word count (WC <- WC – 1)
- If WC = 0, then Interrupt to acknowledge done, else repeat same process

Output

- Send read control signal to memory
- Read data from memory
- Increment address register (Address Reg<- Address Reg +1)
- Decrement word count (WC <- WC – 1)
- Disassemble the word
- Transfer data to the output device byte by byte
- If WC = 0, then Interrupt to acknowledge done, else repeat same process

9.4 Input Output Processor (I/O Channel)

Processor with direct memory access capability that communicates with I/O devices is called I/O processor (channel). Channel accesses memory by cycle stealing. Channel can execute a channel program stored in the main memory. CPU initiates the channel by executing a channel I/O class instruction and once initiated, channel operates independently of the CPU.

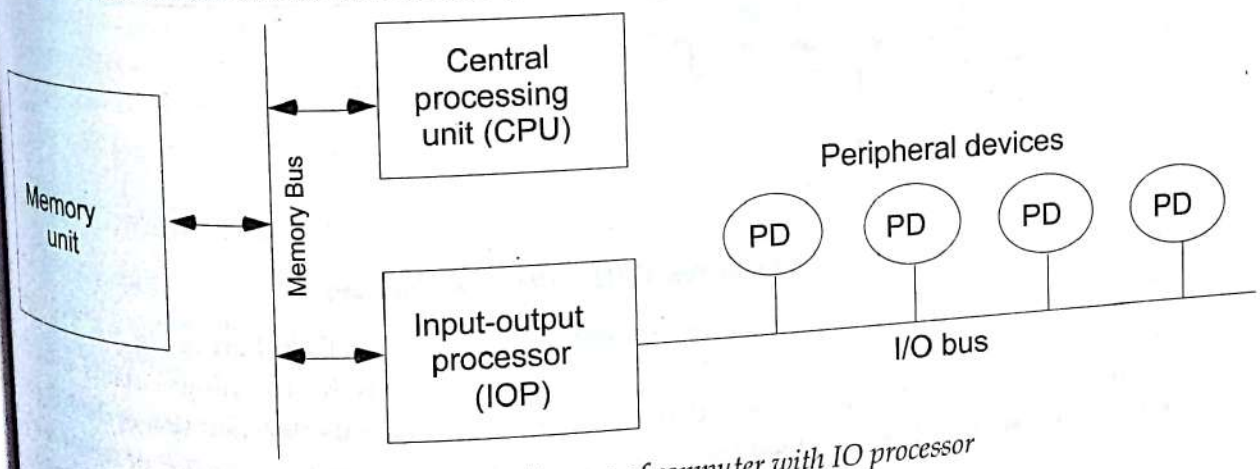


Figure 9.5: *Block diagram of computer with IO processor*

CPU-IOP Communication

The communication between CPU and IOP may take different forms, depending on the particular computer considered. In most cases the memory unit acts as a message center where

Data Communication Processor

A data communication Processor is an I/O processor that distributes and collects data from many remote terminals connected through telephone and other communication lines. It is a specialized I/O processor designed to communicate directly with data communication networks. A communication network may consist of any of a wide variety of devices, such as printers, interactive display devices, digital sensors, or a remote computing facility. With the use of a data communication processor, the computer can service fragments of each network demand in an interspersed manner and thus have the apparent behavior of serving many users at once. In this way the computer is able to operate efficiently in a time-sharing environment.

The most striking difference between an I/O processor and a data communication processor is in the way the processor communicates with the I/O devices. An I/O processor communicates with the Peripherals through a common I/O bus that is comprised of many data and control lines. All peripherals share the common bus and use it to transfer information to and from the I/O processor. A data communication processor communicates with each terminal through a single pair of wires. Both data and control information are trans-character received. Another procedure used in asynchronous terminals involving a human operator is to *echo* the character. The character transmitted from the keyboard to the computer is recognized by the processor and retransmitted to the terminal printer. The operator would realize that an error occurred during transmission if the character printed is not the same as the character whose key he has struck.

Modes of Data Transfer

Data can be transmitted between two points in three different modes: simplex, half-duplex, or full-duplex.

Simplex

A simplex line carries information in one direction only. This mode is seldom used in data communication because the receiver cannot communicate with the transmitter to indicate the occurrence of errors. Examples of simplex transmission are radio and television broadcasting.

Half-Duplex

A half-duplex transmission system is one that is capable of transmitting in both directions but data can be transmitted in only one direction at a time. A pair of wires is needed for this mode. A common situation is for one modem to ad as the transmitter and the other as the receiver. When transmission in one direction is completed, the role of the modems is reversed to enable transmission in the reverse direction. The time required to switch a half-duplex line from one direction to the other is called the turnaround time.

Full-Duplex

A full-duplex transmission can send and receive data in both directions simultaneously. This can be achieved by means of a four-wire link, with a different pair of wires dedicated to each direction of transmission. Alternatively, a two-wire circuit can support full-duplex communication if the frequency spectrum is subdivided into two non-overlapping frequency bands to create separate receive and transmit channels in the same physical pair of wires.

Protocol

The communication lines, modems, and other equipment used in the transmission of information between two or more stations are called a data link. The orderly transfer of information in a data link is accomplished by means of a protocol. A data link control protocol is a set of rules that are followed by interconnecting computers and terminals to ensure the orderly transfer offered in a serial fashion with the result that the transfer rate is much slower. The task of the data communication processor is to transmit and collect digital information to and from each terminal, determine if the information is data or control and respond to all requests according to predetermined established procedures. The processor, obviously, must also communicate with the CPU and memory in the same manner as any I/O processor.

Character-Oriented Protocols

In a character-oriented protocol, data to be carded are 8-bit characters from a coding system. The header, which normally carries the source and destination addresses and other control information, and the trailer, which carries error detection or error correction redundant bits, are also multiples of 8 bits. To separate one frame from the next, an 8-bit (1-byte) flag is added at the beginning and the end of a frame. The flag, composed of protocol-dependent special characters, signals the start or end of a frame. Figure 1 shows the format of a frame in a character-oriented protocol.

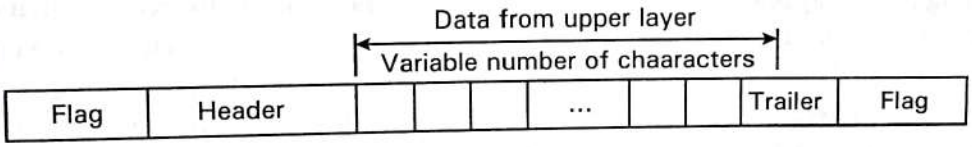


Figure 9.7: *A frame in a character-oriented protocol*

Character-oriented framing was popular when only text was exchanged by the data link layers. The flag could be selected to be any character not used for text communication. Now, however, we send other types of information such as graphs, audio, and video. Any pattern used for the flag could also be part of the information. If this happens, the receiver, when it encounters this pattern in the middle of the data, thinks it has reached the end of the frame. To fix this problem, a byte-stuffing strategy was added to character-oriented framing. In byte stuffing (or character stuffing), a special byte is added to the data section of the frame when there is a character with the same pattern as the flag. The data section is stuffed with an extra byte. This byte is usually called the escape character (ESC), which has a predefined bit pattern. Whenever the receiver encounters the ESC character, it removes it from the data section and treats the next character as data, not a delimiting flag.

Byte stuffing by the escape character allows the presence of the flag in the data section of the frame, but it creates another problem. What happens if the text contains one or more escape characters followed by a flag? The receiver removes the escape character, but keeps the flag, which is incorrectly interpreted as the end of the frame. To solve this problem, the escape characters that are part of the text must also be marked by another escape character. In other words, if the escape character is part of the text, an extra one is added to show that the second one is part of the text.

Character-oriented protocols present another problem in data communications. The universal coding systems in use today have 16-bit and 32-bit characters that conflict with 8-bit characters. We can say that in general, the target is moving toward the bit-oriented protocols that we discuss next.

Bit-Oriented Protocols

In a bit-oriented protocol, the data section of a frame is a sequence of bits to be interpreted by the upper layer as text, graphic, audio, video, and so on. However, in addition to headers (and possible trailers), we still need a delimiter to separate one frame from the other. Most protocols use a special 8-bit pattern flag 01111110 as the delimiter to define the beginning and the end of the frame, as shown in Figure 2.

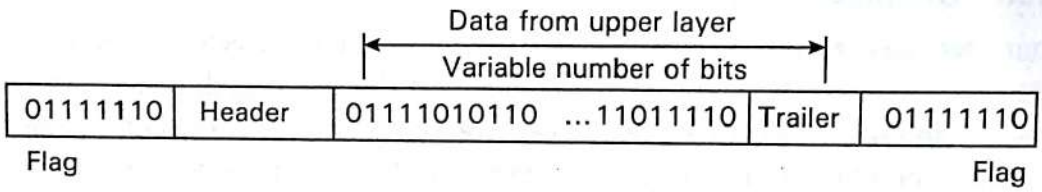


Figure 9.8: *A frame in a bit-oriented protocol*

This flag can create the same type of problem we saw in the byte-oriented protocols. That is, if the flag pattern appears in the data, we need to somehow inform the receiver that this is not the end of the frame. We do this by stuffing 1 single bit (instead of 1 byte) to prevent the pattern from looking like a flag. The strategy is called bit stuffing. In bit stuffing, if a 0 and five consecutive 1 bits are encountered, an extra 0 is added. This extra stuffed bit is eventually removed from the data by the receiver. Note that the extra bit is added after one 0 followed by five ls regardless of the value of the next bit. This guarantees that the flag field sequence does not inadvertently appear in the frame.

Bit stuffing is the process of adding one extra 0 whenever five consecutive Is follow a 0 in the data, so that the receiver does not mistake the pattern 0111110 for a flag.

Figure 3 shows bit stuffing at the sender and bit removal at the receiver. Note that even if we have a 0 after five 1 s, we still stuff a 0. The 0 will be removed by the receiver.

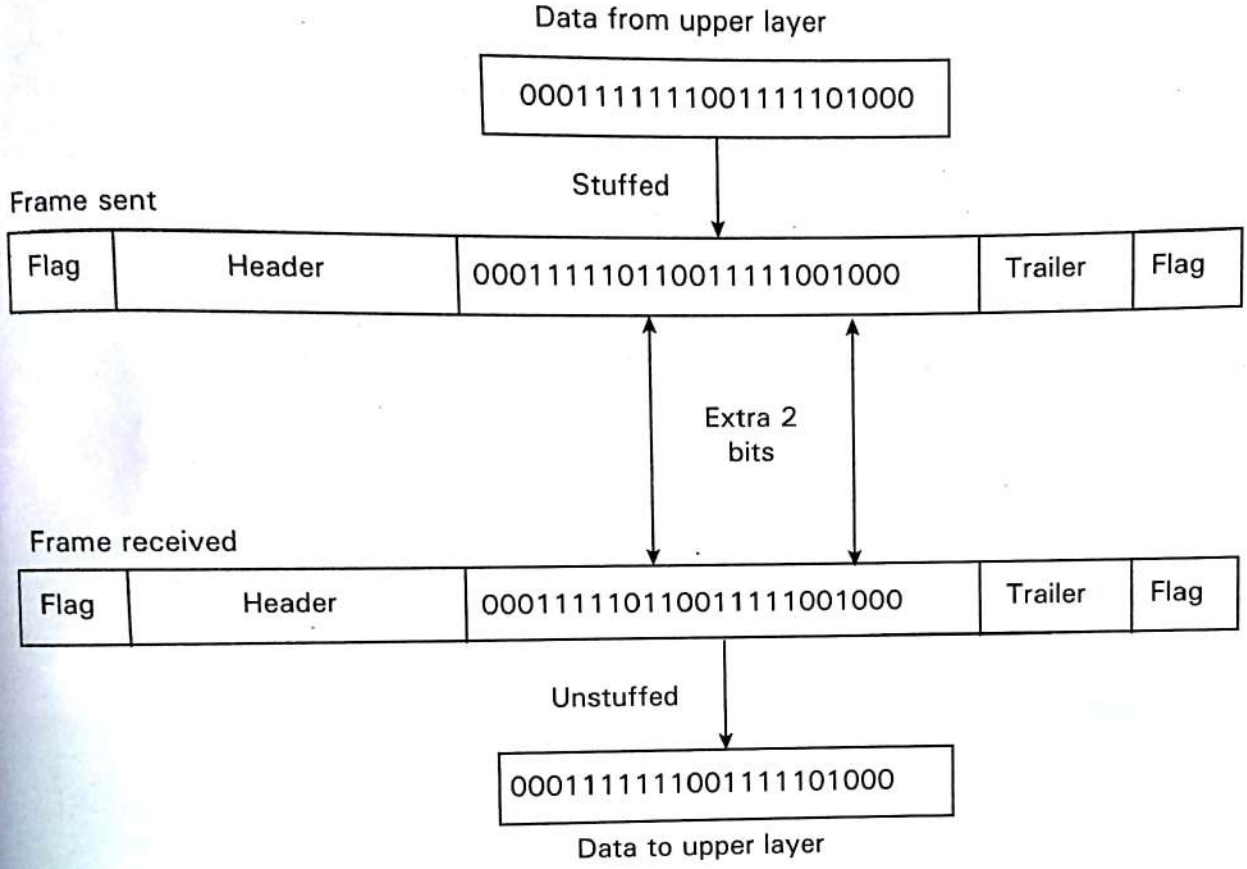


Figure 9.9: *Bit stuffing and unstuffing*

This means that if the flag like pattern 01111110 appears in the data, it will change to 011111010 (stuffed) and is not mistaken as a flag by the receiver. The real flag 01111110 is not stuffed by the sender and is recognized by the receiver.

Chapter 10

MEMORY ORGANIZATION

Objective

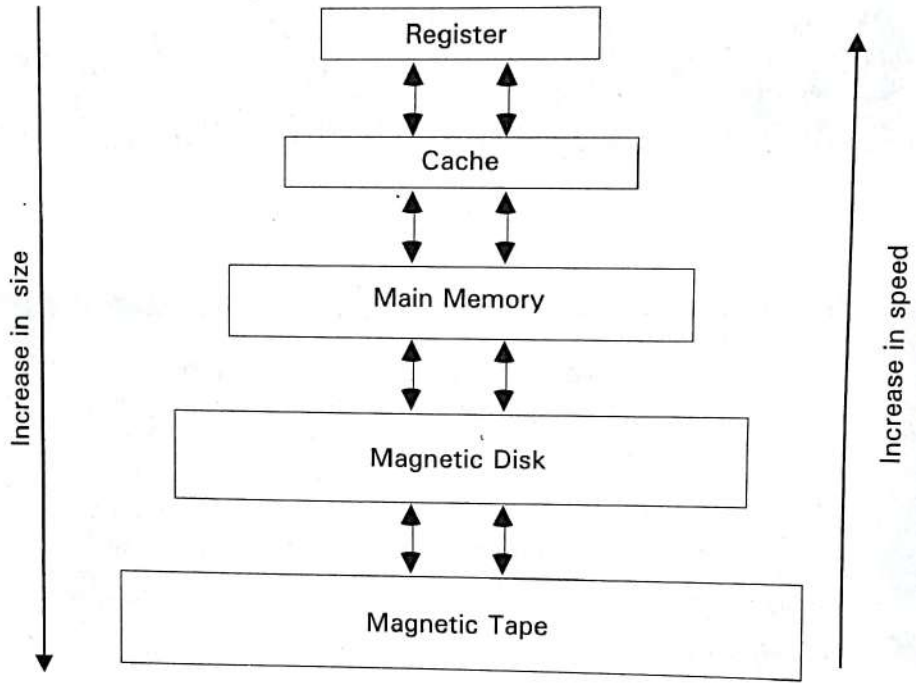
This chapter introduces the concept of memory hierarchy, composed of cache memory, main memory, and auxilary memory such as magnetic disks. The organization and operation of associative memory is explained. The concept of memory management is introduced through the presentation of hardware requirements for a cache memory and a virtual memory system.

OUTLINE

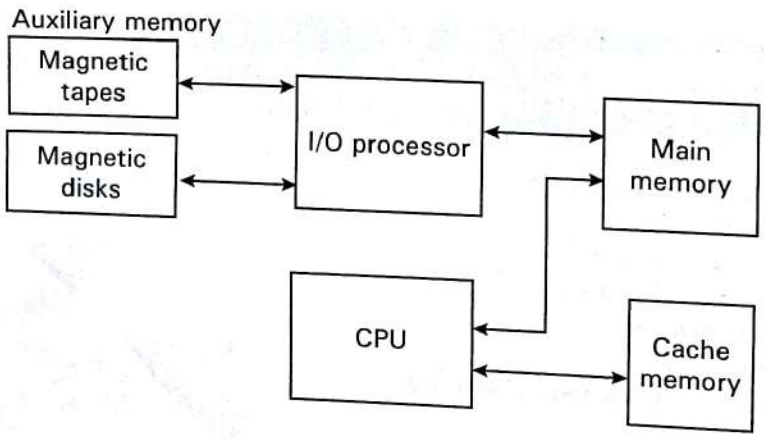
10.1 Memory Hierarchy
10.2 Main Memory
10.3 Auxiliary Memory
10.4 Associative Memory
10.5 Cache Memory
10.6 Virtual Memory
10.7 Memory Management Hardware

10.1 MEMORY HIERARCHY

Memory unit is essential component of digital computer since it is needed for storing programs and data. Memory unit that communicates directly with CPU is called Main memory. Devices that provide backup storage is called auxiliary memory. Only programs and data currently needed by processor reside in the main memory. All other information is stored in auxiliary memory and transferred to main memory when needed. The purpose of memory hierarchy is to obtain the highest possible access speed while minimizing the total cost of the memory system. The Figure 10.1 (a) and 10.1 (b) shows the organization of different kinds of memory and its' hierarchy.



(a)



(b)

Fig 10.1 : *Memory hierarchy in a computer system*

10.2 Main Memory

The main memory is the central storage unit in a computer system. Primary memory holds only those data and instructions on which computer is currently working. It has limited capacity and data is lost when power is switched off. It is generally made up of semiconductor device. These memories are not as fast as registers. The data and instruction required to be processed reside in main memory. It is divided into two subcategories RAM and ROM.

Memory Address Map of RAM and ROM

The designer of a computer system must calculate the amount of memory required for the particular application and assign it to either RAM or ROM. The interconnection between memory and processor is then established from knowledge of the size of memory needed and the type of RAM and ROM chips available. The addressing of memory can be established by means of a table that specifies the memory address assigned to each chip.

The table, called a **memory address map**, is a pictorial representation of assigned address space for each chip in the system, shown in table 10.1. To demonstrate with a particular example, assume that a computer system needs 512 bytes of RAM and 512 bytes of ROM. The RAM and ROM chips to be used are specified in figure 10.1 and figure 10.2.

Memory Address Map of RAM and ROM

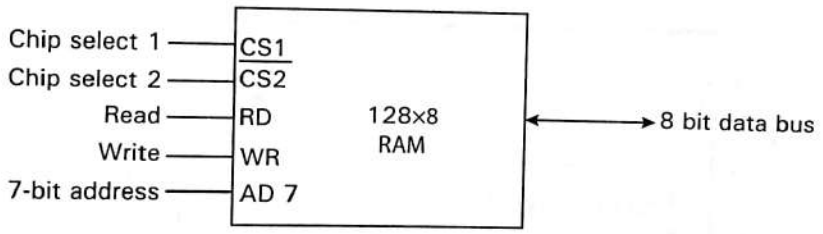


Figure 10.2: *Typical RAM chip*

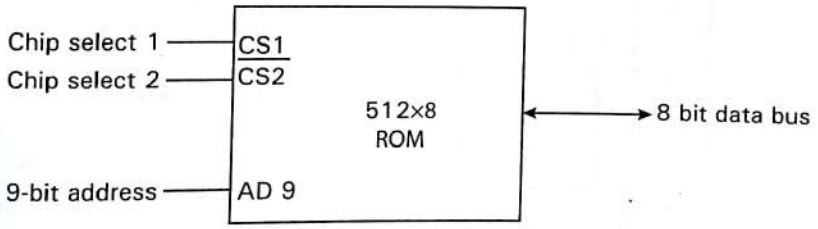


Figure 10.3: *Typical ROM chip*

Table 10.1 : *Memory address map for microcomputer*

Component	Hexa address	Address bus									
		10	9	8	7	6	5	4	3	2	1
RAM 1	0000 - 007F	0	0	0	×	×	×	×	×	×	×
RAM 2	0080 - 00FF	0	0	1	×	×	×	×	×	×	×
RAM 3	0100 - 017F	0	1	0	×	×	×	×	×	×	×
RAM 4	0180 - 01FF	0	1	1	×	×	×	×	×	×	×
ROM	0200 - 03FF	1	×	×	×	×	×	×	×	×	×

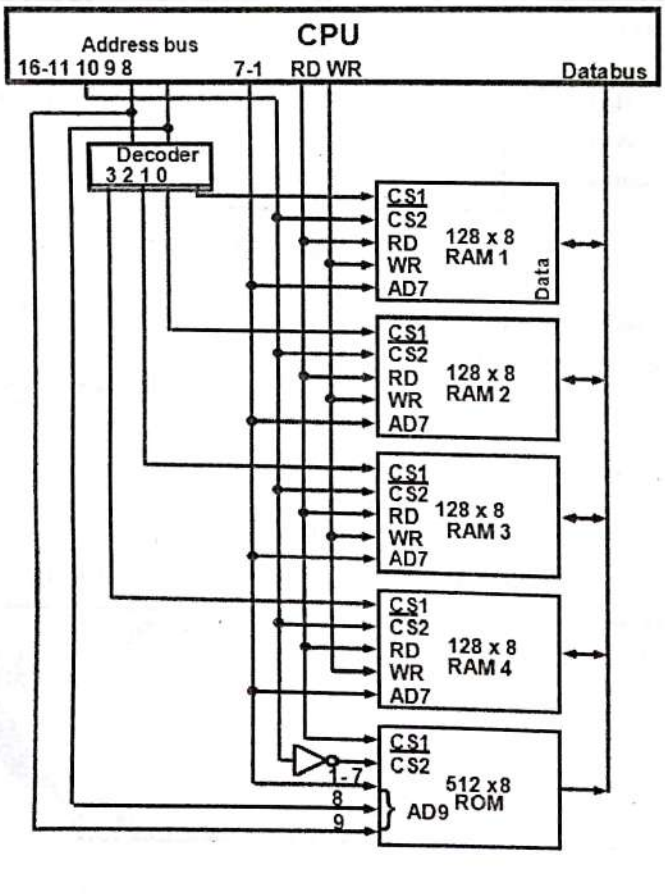
The component column specifies whether a RAM or a ROM chip is used. The hexadecimal address column assigns a range of hexadecimal equivalent addresses for each chip. The address bus lines are listed in the third column. Although there are 16 lines in the address bus, the table shows only 10 lines because the other 6 are not used in this example and are assumed to be zero. The small x's under the address bus lines designate those lines that must be connected to the address inputs in each chip.

The RAM chips have 128 bytes and need seven address lines. The ROM chip has 512 bytes and needs 9 address lines. The x's are always assigned to the low-order bus lines: lines 1 through 7 for the RAM and lines 1 through 9 for the ROM. It is now necessary to distinguish between four RAM chips by assigning to each a different address. For this particular example we choose bus lines 8 and 9 to represent four distinct binary combinations. The table clearly shows that the nine low-order bus lines constitute a memory space for RAM equal to $2^9 = 512$ bytes. The distinction between a RAM and ROM address is done with another bus line. Here we choose line 10 for this purpose. When line 10 is 0, the CPU selects a RAM, and when this line is equal to 1, it selects the ROM

Memory connections to CPU

RAM and ROM chips are connected to a CPU through the data and address buses. The low-order lines in the address bus select the byte within the chips and other lines in the address bus select a particular chip through its chip select inputs.

Fig. 10.4: Connection *of Memory To CPU*



10.3 Auxiliary Memory

Magnetic Tape: Magnetic tapes are used for large computers like mainframe computers where large volume of data is stored for a longer time. In PC also you can use tapes in the form of cassettes. The cost of storing data in tapes is inexpensive. Tapes consist of magnetic materials that store data permanently. It can be 12.5 mm to 25 mm wide plastic film-type and 500 meter to 1200 meter long which is coated with magnetic material. The deck is connected to the central processor and information is fed into or read from the tape through the processor. It's similar to cassette tape recorder.

Magnetic tape is an information storage medium consisting of a magnetisable coating on a thin plastic strip. Nearly all recording tape is of this type, whether used for video with a video cassette recorder, audio storage (reel-to-reel tape, compact audio cassette, digital audio tape (DAT), digital linear tape (DLT) and other formats including 8-track cartridges) or general purpose digital data storage using a computer (specialized tape formats, as well as the above-mentioned compact audio cassette, used with home computers of the 1980s, and DAT, used for backup in workstation installations of the 1990s).

Magneto-optical and optical tape storage products have been developed using many of the same concepts as magnetic storage, but have achieved little commercial success.

Magnetic Disk: You might have seen the gramophone record, which is circular like a disk and coated with magnetic material. Magnetic disks used in computer are made on the same principle. It rotates with very high speed inside the computer drive. Data is stored on both the surface of the disk. Magnetic disks are most popular for direct access storage device. Each disk consists of a number of invisible concentric circles called tracks. Information is recorded on tracks of a disk surface in the form of tiny magnetic spots. The presence of a magnetic spot represents one bit and its absence represents zero bit. The information stored in a disk can be read many times without affecting the stored data. So the reading operation is non-destructive. But if you want to write a new data, then the existing data is erased from the disk and new data is recorded. For Example-Floppy Disk.

The primary computer storage device. Like tape, it is magnetically recorded and can be re-recorded over and over. Disks are rotating platters with a mechanical arm that moves a read/write head between the outer and inner edges of the platter's surface. It can take as long as one second to find a location on a floppy disk to as little as a couple of milliseconds on a fast hard disk. See hard disk for more details.

The disk surface is divided into concentric tracks (circles within circles). The thinner the tracks, the more storage. The data bits are recorded as tiny magnetic spots on the tracks. The smaller the spot, the more bits per inch and the greater the storage.

Sectors

Tracks are further divided into sectors, which hold a block of data that is read or written at one time; for example, READ SECTOR 782, WRITE SECTOR 5448. In order to update the disk, one or more sectors are read into the computer, changed and written back to disk. The operating system figures out how to fit data into these fixed spaces. Modern disks have more sectors in the outer tracks than the inner ones because the outer radius of the platter is greater than the inner radius

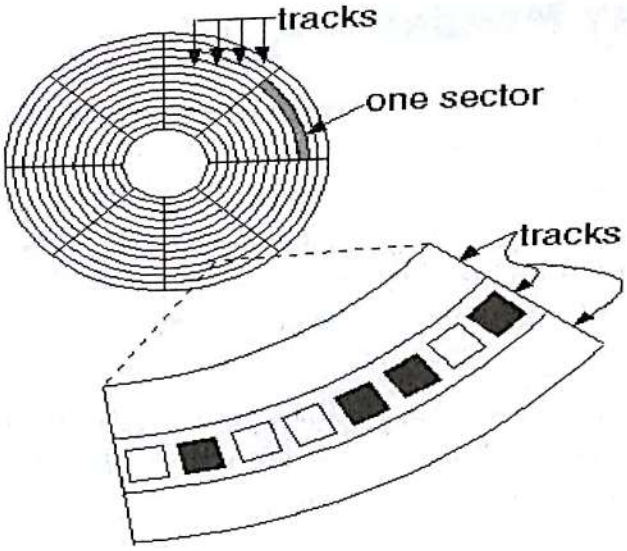


Figure 10.5: *Block diagram of Magnetic Disk*

Optical Disk: With every new application and software there is greater demand for memory capacity. It is the necessity to store large volume of data that has led to the development of optical disk storage medium. Optical disks can be divided into the following categories:

1. Compact Disk/ Read Only Memory (CD-ROM
2. Write Once, Read Many (WORM)
3. Erasable Optical Disk

10.4 Associative Memory: Content Addressable Memory (CAM)

The time required to find an item stored in memory can be reduced considerably if stored data can be identified for access by the content of the data itself rather than by an address. A memory unit accessed by content is called an associative memory or content addressable memory (CAM). This type of memory is accessed simultaneously and in parallel on the basis of data content rather than by specific address or location. The block diagram of an associative memory is shown in figure 10.6.

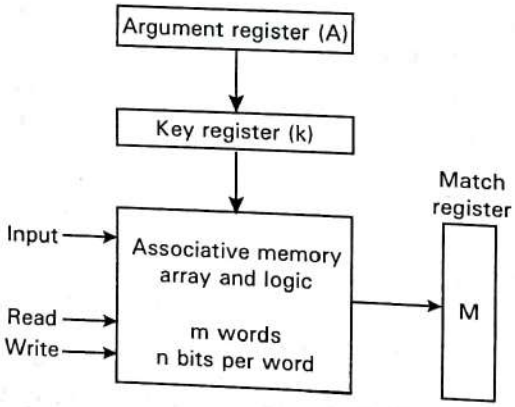


Figure 10.6: *Block diagram of associative memory*

It consists of a memory array and logic form words with n bits per word. The argument register A and key register K each have n bits, one for each bit of a word. The match register M has m bits, one for each memory word. Each word in memory is compared in parallel with the content of the argument register. The words that match the bits of the argument register set a corresponding bit in the match register. After the matching process, those bits in the match register that have been set indicate the fact that their corresponding words have been matched. Reading is accomplished by a sequential access to memory for those words whose corresponding bits in the match register have been set.

Hardware Organization

The key register provides a mask for choosing a particular field or key in the argument word. The entire argument is compared with each memory word if the key register contains all 1's. Otherwise, only those bits in the argument that have 1st in their corresponding position of the key register are compared. Thus the key provides a mask or identifying piece of information which specifies how the reference to memory is made. To illustrate with a numerical example, suppose that the argument register A and the key register K have the bit configuration shown below. Only the three leftmost bits of A are compared with memory words because K has 1's in these position. Word 2 matches the unmasked argument field because the three leftmost bits of the argument and the word are equal.

A	101 111100	
K	111 000000	
Word1	100 111100	no match
Word2	101 000001	match

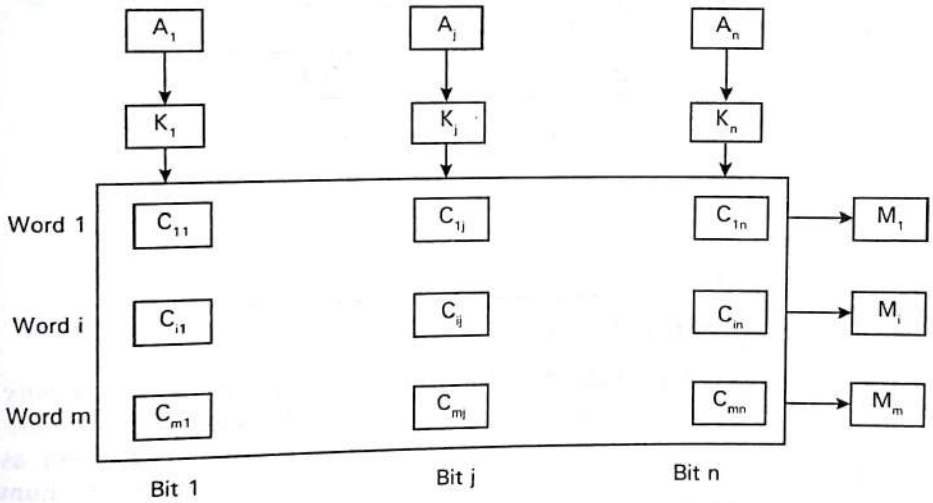


Figure 10.7: *Associative Memory of m word, n cells per word.*

The relation between the memory array and external registers in an associative memory is shown in figure 10.7. The cells in the array are marked by the letter C with two subscripts. The first subscript gives the word number and the second specifies the bit position in the word. Thus cell Cij is the cell for bit j in words i. A bit Aj in the argument register is compared with all the bits in column j of the array provided that Kj =1. This is done for all columns j = 1, 2... n. If a match occurs between all the unmasked bits of the argument and the bits in word i, the

corresponding bit Mi in the match register is set to 1. If one or more unmasked bits of the argument and the word do not match, Mi is cleared to 0.

10.5 CACHE MEMORY

Cache is a fast small capacity memory that should hold those information which are most likely to be accessed. The basic operation of the cache is, when the CPU needs to access memory, the cache is examined. If the word is found in the cache, it is read from the fast memory. If the word addressed by the CPU is not found in the cache, the main memory is accessed to read the word. The transformation of data from main memory to cache memory is referred to as a **mapping process.**

Associative mapping

Consider the main memory can store 32K words of 12 bits each. The cache is capable of storing 512 of these words at any given time. For every word stored in cache, there is a duplicate copy in main memory. The CPU communicates with both memories. It first sends a 15-bit address to cache. If there is a hit, the CPU accepts the 12-bit data from cache, if there is miss, the CPU reads the word from main memory and the word is then transferred to cache.

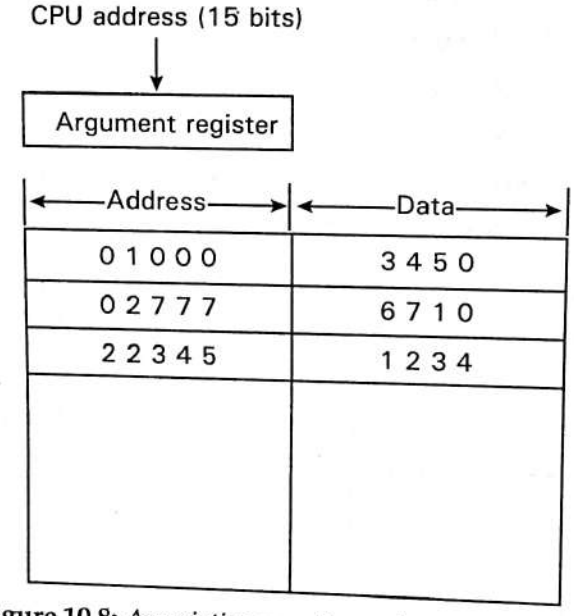


Figure 10.8: *Associative mapping cache (all numbers in octal)*

The associative memory stores both the address and content (data) of the memory word. This permits any location in cache to store any word from main memory. The figure 10.8 shows three words presently stored in the cache. The address value of 15 bits is shown as a five-digit octal number and its corresponding 12-bit word is shown as a four-digit octal number. A CPU address of 15 bits is placed in the argument register and the associative memory is searched for a matching address. If the address is found the corresponding 12-bit data is read and sent to CPU. If no match occurs, the main memory is accessed for the word. The address data pairs then transferred to the associative cache memory. If the cache is full, an address data pair must be displaced to make room for a pair that is needed and not presently in the cache. This constitutes a first-in first-one (FIFO) replacement policy.

Direct Mapping in Organization of Cache Memory

The CPU address of 15 bits is divided into two fields. The nine least significant bits constitute the index field and the remaining six bits from the tag field. The figure 10.9 shows that main memory needs an address that includes both the tag and the index.

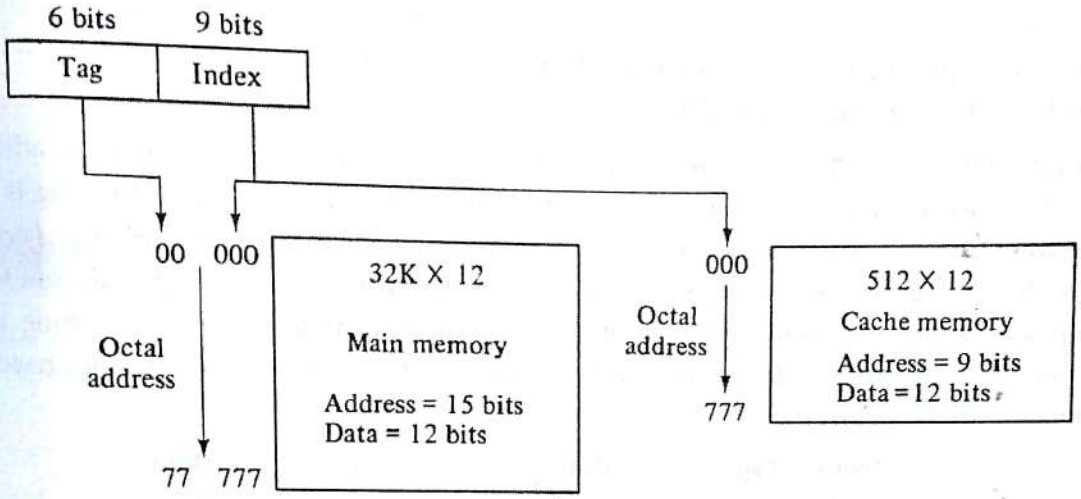


Figure 10.9: *Addressing relationships between main and cache memories*

The number of bits in the index field is equal to the number of address bits required to access the cache memory. The internal organization of the words in the cache memory is as shown in figure 10.9.

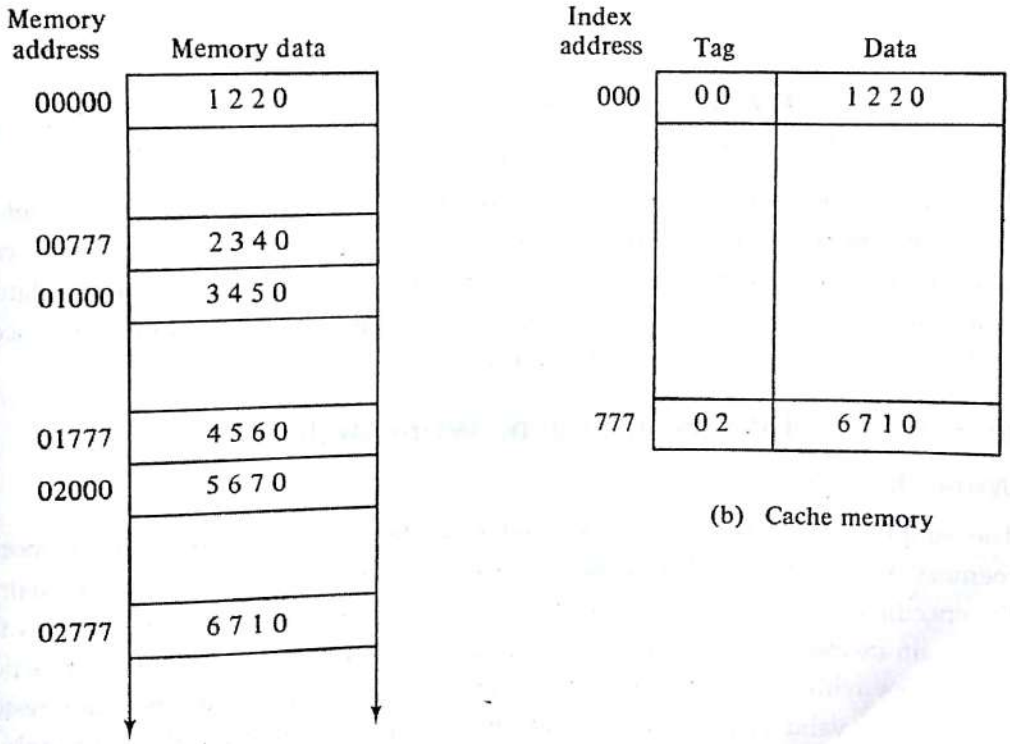


Figure 10.10: *Direct mapping cache organization*

Each word in cache consists of the data word and its associated tag. When a new word is first brought into the cache, the tag bits are stored alongside the data bits. When the CPU generates a memory request the index field is used for the address to access the cache. The tag field of the CPU address is compared with the tag in the word read from the cache. If the two tags match, there is a hit and the desired data word is in cache and if there is no match, there is a miss and the required word is read from main memory. It is then stored in the cache together with the new tag, replacing the previous value. The word at address zero is presently stored in the cache (index = 000, tag = 00, data = 1220).

Suppose that the CPU now wants to access the word at address 02000. The index address is 000, so it is used to access the cache. The two tags are then compared. The cache tag is 00 but the address tag is 02, which does not produce a match. Therefore, the main memory is accessed and the data word 5670 is transferred to the CPU. The cache word at index address 000 is then replaced with a tag of 02 and data of 5670. The **disadvantage** of direct mapping is that two words with the same index in their address but with different tag values cannot reside in cache memory at the same time.

Index	Tag	Data		Tag	Data
000	0 1	3 4 5 0		0 2	5 6 7 0
777	0 2	6 7 1 0		0 0	2 3 4 0

Figure 10.11: *Two way set associative mapping cache*

The comparison logic is done by an associative search of the tags in the set similar to an associative memory search: thus the name "set-associative". When a miss occurs in a set-associative cache and the set is full, it is necessary to replace one of the tag-data items with a new value. The most common replacement algorithms used are: random replacement, first-in first-out (FIFO), and least recently used (LRU).

Write-through and Write-back Cache Write Method

Write Through

The simplest and most commonly used procedure is to update main memory with every memory write operation. The cache memory being updated in parallel if it contains the word at the specified address. This is called the *write-through* method. This method has the advantage that main memory always contains the same data as the cache. This characteristic is important in systems with direct memory access transfers. It ensures that the data residing in main memory are valid at all times so that an I/O device communicating through DMA would receive the most recent updated data.

Write-Back (Copy-Back)

The second procedure is called the write-back method. In this method only the cache location is updated during a write operation. The location is then marked by a flag so that later when the word is removed from the cache it is copied into main memory. The reason for the write-back method is that during the time a word resides in the cache, it may be updated several times. However, as long as the word remains in the cache, it does not matter whether the copy in main memory is out of date, since requests from the word are filled from the cache. It is only when the word is displaced from the cache that an accurate copy need be rewritten into main memory.

10.6 Virtual Memory

Virtual memory is used to give programmers the illusion that they have a very large memory at their disposal, even though the computer actually has a relatively small main memory. This system provides a mechanism for translating program-generated addresses into correct main memory locations.

Address space

An address used by a programmer will be called a virtual address, and the set of such addresses is known as address space.

Memory space

An address in main memory is called a location or physical address. The set of such locations is called the memory space.

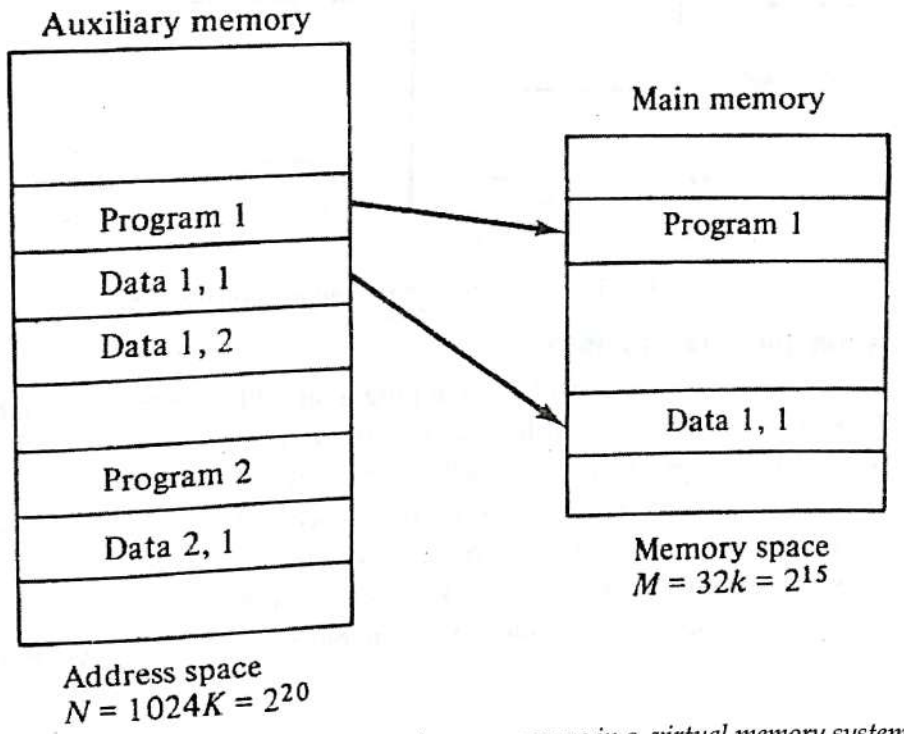
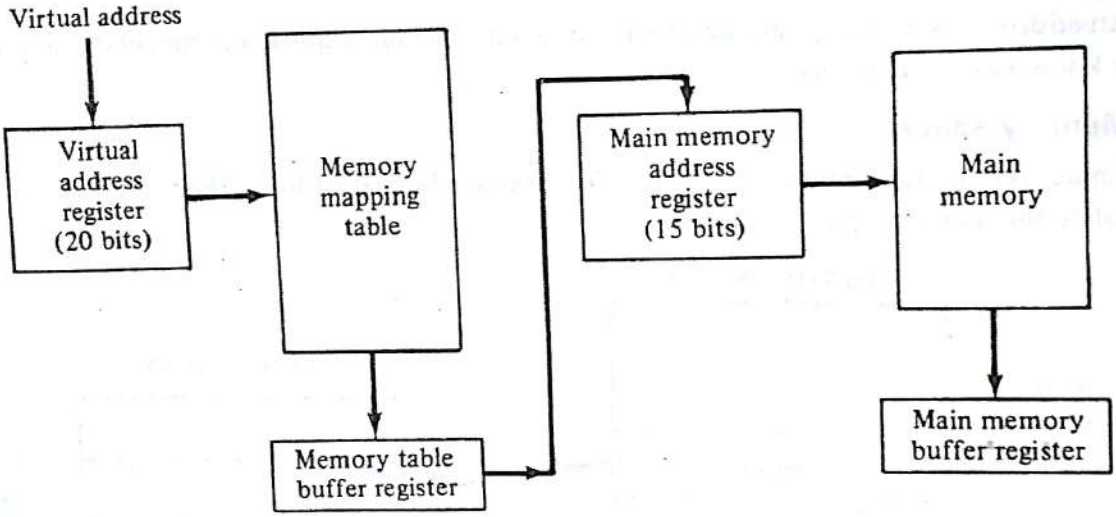


Figure 10.12: *Relation between address and memory space in a virtual memory system*

As an illustration, consider a computer with a main-memory capacity of 32K words (K = 1024). Fifteen bits are needed to specify a physical address in memory since 32K = 2^{15}. Suppose that the computer has available auxiliary memory for storing 2^{20} = 1024K words. Thus auxiliary memory has a capacity for storing information equivalent to the capacity of 32 main memories.

Denoting the address space by N and the memory space by M, we then have for this example N = 1024K and M = 32K. In a multiprogramming computer system, programs and data are transferred to and from auxiliary memory and main memory based on demands imposed by the CPU. Suppose that program 1 is currently being executed in the CPU. Program 1 and a portion of its associated data are moved from auxiliary memory into main memory as shown in figure 10.12. Portions of programs and data need not be in contiguous locations in memory since information is being moved in and out, and empty spaces may be available in scattered locations in memory. In our example, the address field of an instruction code will consist of 20 bits but physical memory addresses must be specified with only 15 bits. Thus CPU will reference instructions and data with a 20-bit address, but the information at this address must be taken from physical memory because access to auxiliary storage for individual words will be too long.



Figurer 10.13: *Memory table for mapping a virtual address*

Address mapping using pages

The table implementation of the address mapping is simplified if the information in the address space and the memory space are each divided into groups of fixed size. The physical memory is broken down into groups of equal size called blocks, which may range from 64 to 4096 words each. The term page refers to groups of address space of the same size. Consider a computer with an address space of 8K and a memory space of 4K. If we split each into groups of 1K words we obtain eight pages and four blocks as shown in figure 10.14. At any given time, up to four pages of address space may reside in main memory in any one of the four blocks.

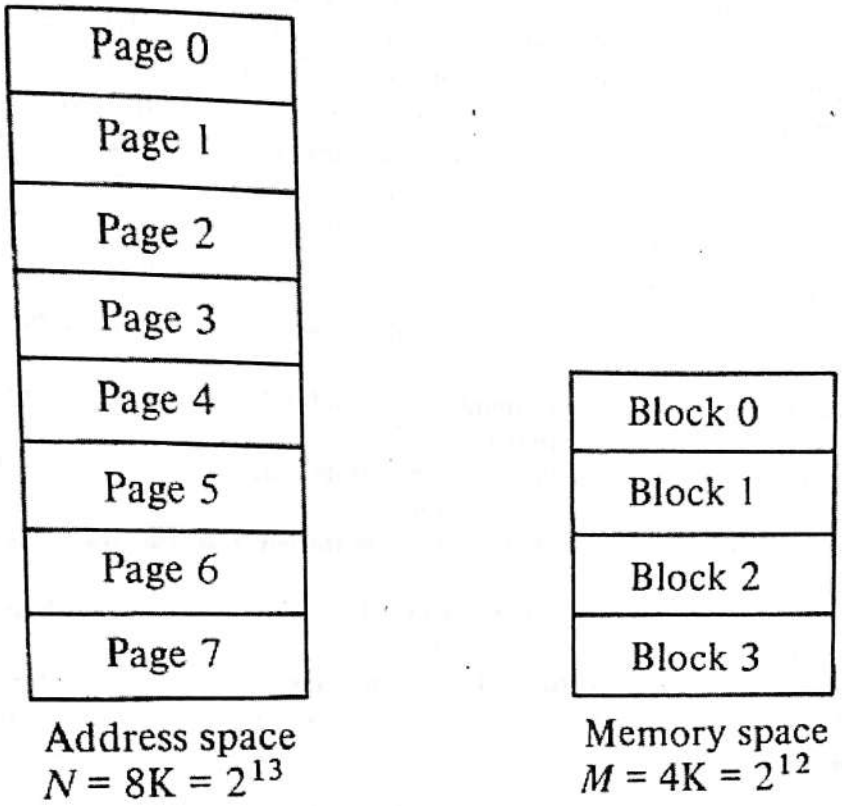


Figure 10.14: *Address and Memory*

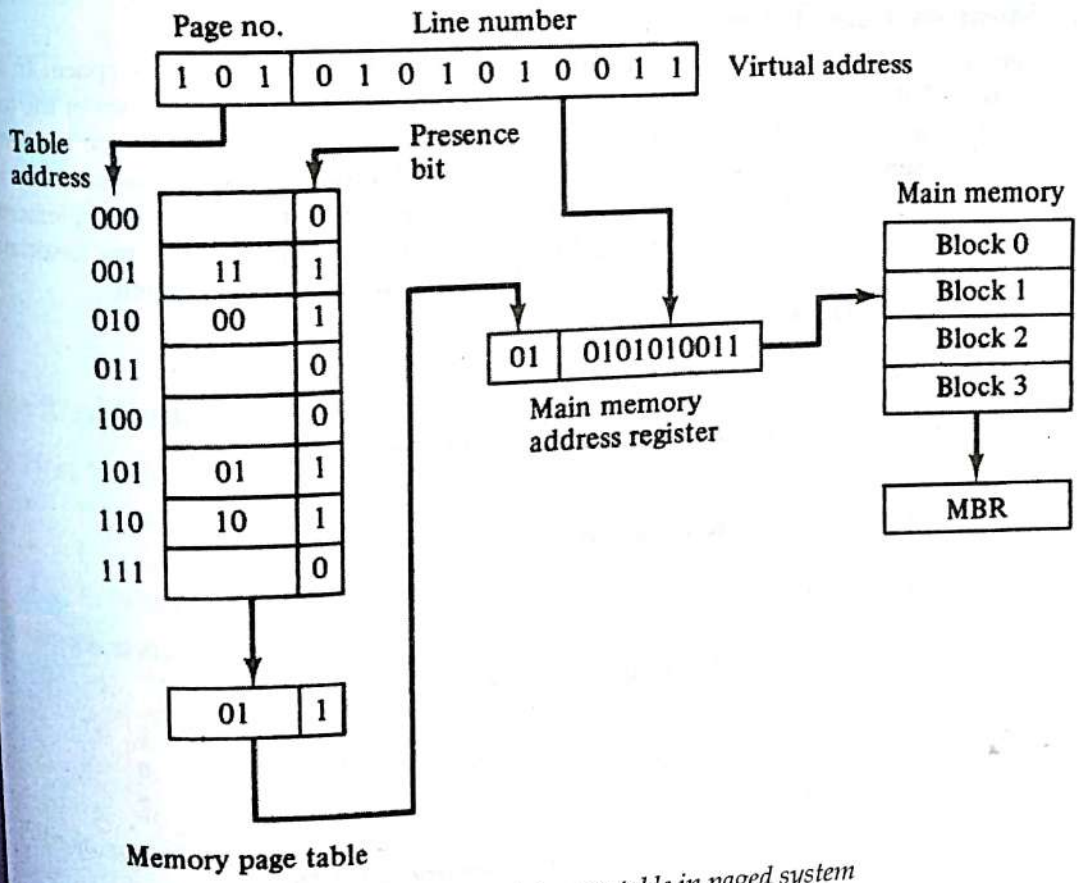


Figure 10.15: *Memory table in paged system*

The organization of the memory mapping table in a paged system is shown in figure 10.15. The memory-page table consists of eight words, one for each page. The address in the page table denotes the page number and the content of the word give the block number where that page is stored in main memory. The table shows that pages 1, 2, 5, and 6 are now available in main memory in blocks 3, 0, 1, and 2, respectively. A presence bit in each location indicates whether the page has been transferred from auxiliary memory into main memory. A 0 in the presence bit indicates that this page is not available in main memory.

The CPU references a word in memory with a virtual address of 13 bits.

- The three high-order bits of the virtual address specify a page number and also an address for the memory-page table.
- The content of the word in the memory page table at the page number address is read out into the memory table buffer register.
- If the presence bit is a 1, the block number thus read is transferred to the two high- order bits of the main memory address register.
- The line number from the virtual address is transferred into the 10 low-order bits of the memory address register.
- A read signal to main memory transfers the content of the word to the main memory buffer register ready to be used by the CPU.
- If the presence bit in the word read from the page table is 0, it signifies that the content of the word referenced by the virtual address does not reside in main memory.

Page Fault

If a page number cannot be found in the Page Table it is called page fault

Associative Memory Page Table

Suppose there are m blocks in memory n number of pages in virtual address space. In this organization page Table requires n entry table in memory. Out of which n-m entries of the table are empty. This is inefficient storage space utilization. More efficient method is use a m-entry page table where page table is made up of an Associative Memory. Associative memory is a memory which stores both data and it's actual address. There fore if page table is implemented by using associative memory it stores both page no. and it's associated block no. associative memory is fast because here normally parallel search is dine on the basis of content.

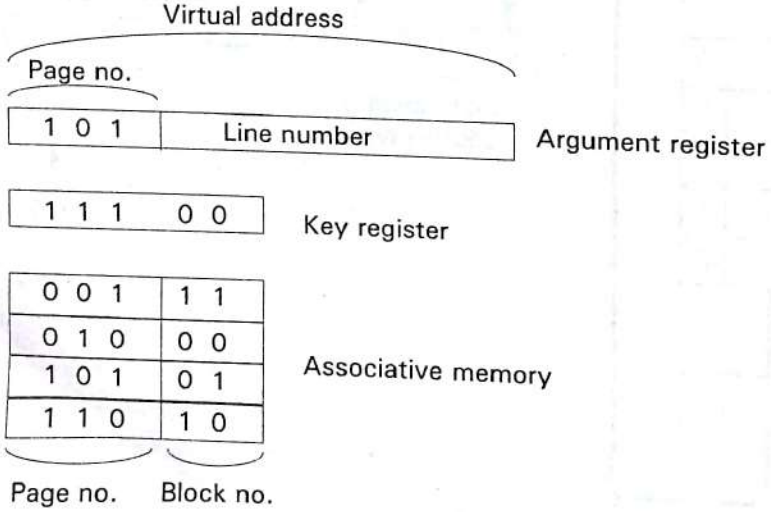
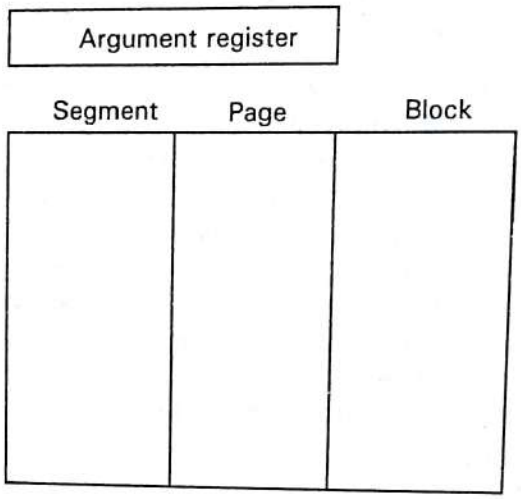


Figure 10.16: *An associative memory page table*

Here key register is used for masking. That is we need to match only first three bits.



(b) Associative memory translation look-aside buffer (TLB)

Figure10.18 :*Mapping of segmented-page memory management unit*

To avoid slow process, a fast associative memory is used to hold the most recently referenced table entries. The first time a given block is referenced, its value together with the corresponding segment and page numbers are entered into the associative memory as shown in figure b.

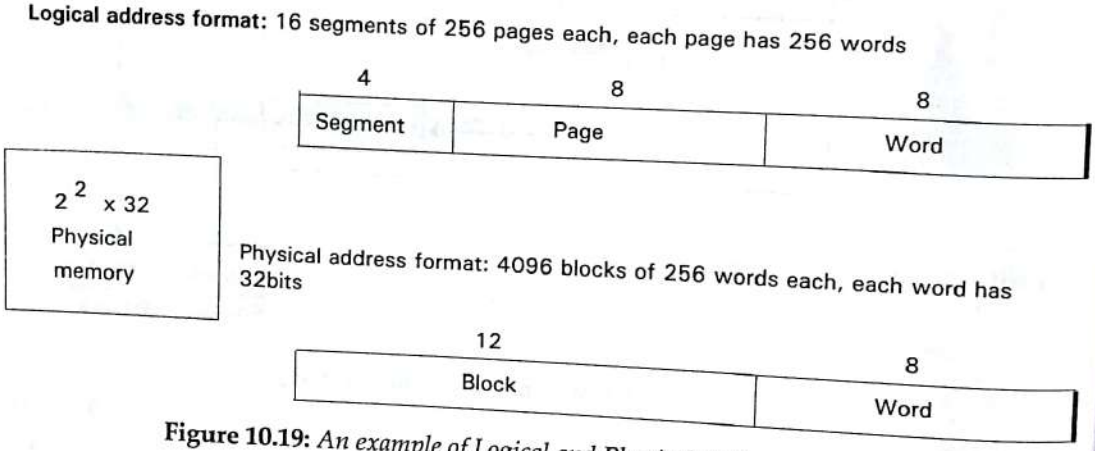
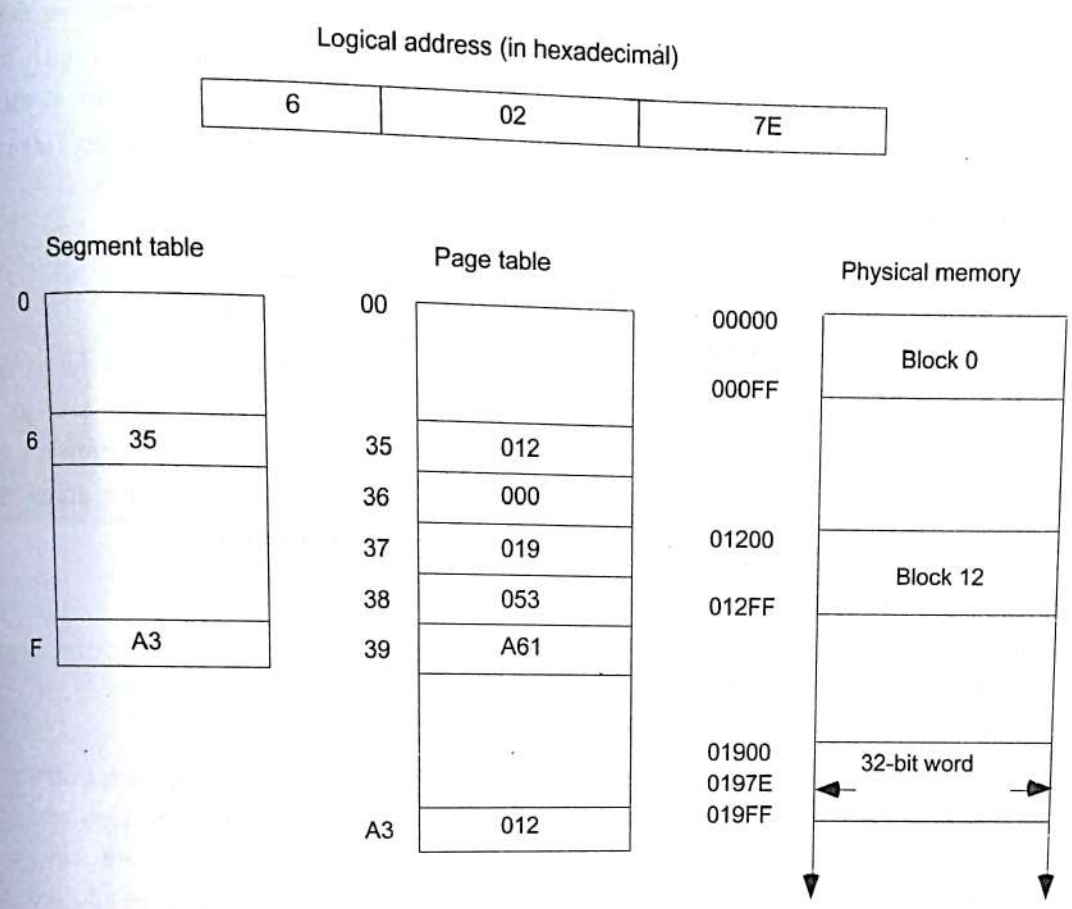


Figure 10.19: *An example of Logical and Physical Address*

Example:



Physical address = Page table [page no. + segment table [segment no]] + line no (word)

[x] → Content at location x

Figure 10.20 : *Logical to physical address mapping example*

Memory Protection

Memory protection can be assigned to the physical address or the logical address. The protection of memory through the physical address can be done by assigning to each block in memory a number of protection bits that indicate the type of access allowed to its corresponding block. Every time a page is moved from one block to another it would be necessary to update the block protection bits. A much better place to apply protection is in the logical address space rather than the physical address space. This can be done by including protection information within the segment table or segment register of the memory management hardware.

The content of each entry in the segment table or a segment register is called a descriptor. A typical descriptor would contain, in addition to a base address field, one or two additional fields for protection purposes. A typical format for a segment descriptor is shown in Fig. 12-25. The base address field gives the base of the page table address in a segmented-page

organization or the block base address in a segment register organization. This is the address used in mapping from a logical to the physical address. The length field gives the segment size by specifying the maximum number of pages assigned to the segment. The length field is compared against the page number is the logical address. A size violation occurs if the page number falls outside the segment length boundary. Thus a given program and its data cannot access memory not assigned to it by the operating system.

Figure 10.21: *Format of a typical segment descriptor*

Base address	Length	Protection

The protection field in a segment descriptor specifies the access rights available to the particular segment. In a segmented-page organization, each entry in the page table may have its own protection field to describe the access rights of each page. The protection information is set into the descriptor by the master control program of the operating system. Some of the access rights of interest that are used for protecting the programs residing in memory are:

1. Full read and write privileges
2. Read only (write protection)
3. Execute only (program protection)
4. System only (operating system protection)

Full read and write privileges are given to a program when it is executing its own instructions. Write protection is useful for sharing system programs such as utility programs and other library routines. These system programs are stored in an area of memory where they can be shared by many users. They can be read by all programs, but no writing is allowed. This protects them from being changed by other programs.

The execute-only condition protects programs from being copied. It restricts the segment to be referenced only during the instruction fetch phase but not during the execute phase. Thus it allows the users to execute the segment program instructions but prevents them from reading the instructions as data for the purpose of copying their content.

Portions of the operating system will reside in memory at any given time. These system programs must be protected by making them inaccessible to unauthorized users. The operating system protection condition is placed in the descriptors of all operating system programs to prevent the occasional user from accessing operating system segments.

Chapter 11

Multiprocessor

Objective

This chapter presents the basic characteristics of multiprocessors. Various interconnection structures are presented. The need for interprocess or arbitration, communication and synchronization is discussed. The cache coherence problem is explained together with some possible solutions.

OUTLINE

11.1Introduction
11.2 Interconnection Structures
11.3 Interprocessor Arbitration
11.4 Interprocessor Communication and Synchronization
11.5 Cache Coherence

11.1 INTRODUCTION

During past few years, it has been focus on the improvement in clock speed and architecture of the processor for increasing the speed of computer processors. But in recent year, chips manufacturers, have reached a limit in how small they can make the transistors inside CPUs without overheating them. Therefore, using multiple processors, or multiprocessing, has become the next step in increasing computing performance.

Multiprocessing

Multiprocessing is the use of two or more central processing units (CPUs) within a single computer system. The term also refers to the ability of a system to support more than one processor and/or the ability to allocate tasks between them Multiprocessing can be implemented in two different ways:

- using more than one physical processor
- using a processor with multiple cores.

However, the terms multitasking or multiprogramming are more appropriate to describe this concept, which is implemented mostly in software, whereas multiprocessing is more appropriate to describe the use of multiple hardware CPUs

Formal Definition of Multiprocessor

Multiprocessor system is an interconnection of two or more CPUs with memory and input-output equipment. The components that forms multiprocessor are CPUs IOPs connected to input -output devices , and memory unit that may be partitioned into a number of separate modules. Multiprocessor are classified as multiple instruction stream, multiple data stream (MIMD) system. *An example of a multiprocessor is one central tower connected to two computer systems.*

Why Choose a Multiprocessor?

- A single CPU can only go so fast, use more than one CPU to improve performance.
- Multiple users.
- Multiple application.
- Multi-tasking within an application.
- Responsiveness and/or throughput.
- Share hardware between CPUs

Difference between Multiprocessor and Multicomputer

Multiprocessor	Multiccomputer
A multiprocessor system is simply is simply a computer that has more than one CPU on its motherboard. It uses two or more CPUs within a single computer system.	A computer made up of several computers. The term generally refers to an architecture in which each processor has its own memory rather than multiple processors with a shared memory.

Characteristics

Multiprocessors System: An interconnection of two or more CPUs with memory and I /O equipment. Computation can proceed in parallel in one of two ways:

- Multiple independent jobs can be made to operate in parallel
- A single job can be partitioned into multiple parallel tasks

lassification of multiprocessor

Multiprocessor are classified by the way their memory is organized, mainly it is classified into two types:

- Shared memory system
- Distributed memory system

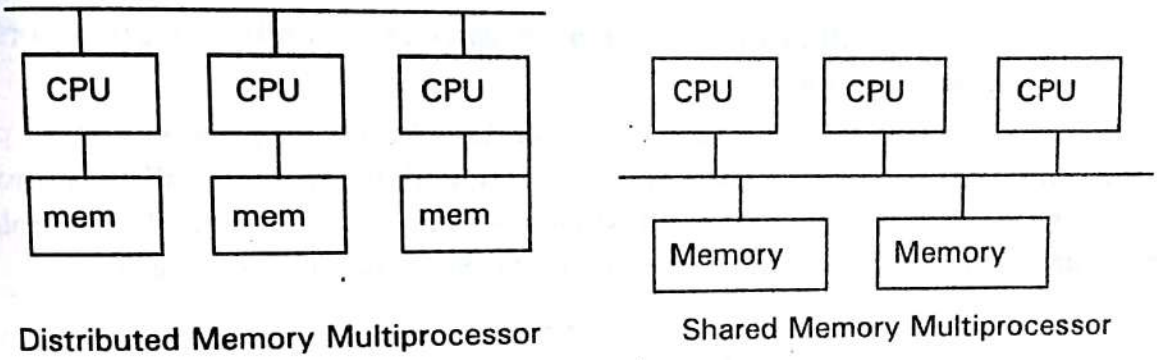


Figure 11.1: *Shared memory versus distributed memory multiprocessor*

Shared memory multiprocessor

- A multiprocessor is shared memory computer system having two or more processing units (Multiple Processors) each sharing main memory and peripherals, in order to simultaneously process programs.
- Shared memory system is also known as tightly coupled multiprocessor.

Distributed Memory Multiprocessor

- Distributed Memory Multiprocessor systems (often referred to as clusters) are based on multiple standalone single or dual processor commodity computers interconnected via high speed communication system.
- It is also known as loosely-coupled multiprocessor.

Difference between shared and distributed memory multiprocessor

Shared memory multiprocessor	Distributed memory multiprocessor
Shared memory systems are physically smaller than distributed memory system.	It is opposite of shared memory system.
More expensive.	Less expensive.

11.2 INTERCONNECTION STRUCTURES

The components that form a MP system are CPUs, IOPs connected to I/O devices, and a memory unit that may be partitioned into a number of separate modules. The physical forms for establishing an interconnection networks. Some of these schemes are given as:

1. Time-shared common bus
2. Multiport memory
3. Crossbar switch
4. Multistage switching network
5. Hypercube system

1. Time –shared common bus

A system common bus multiprocessor system consists of a number of processors connected through path to a memory unit.

In this structure only one processor can communicate with the memory or another processor at any given time when one processor is communicating with the memory, all other processors are either busy with internal operations or must be idle waiting for the bus. It is simplest form of common bus and its structure and interface is similar to single processor system

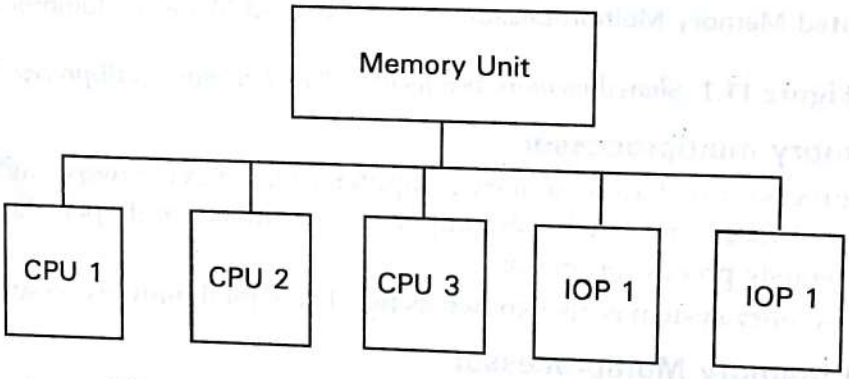


Figure 11.2: *Time shared common bus organization*

Advantages:

- Simplicity
- Flexibility
- Reliability

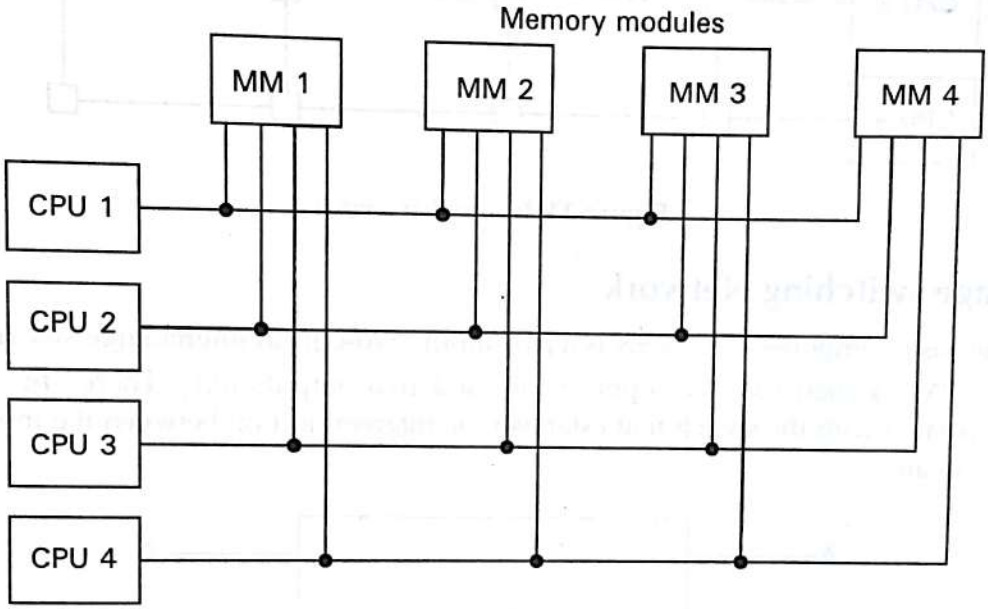
Disadvantages:

- Performance limited by bus cycle time
- Each processor should have local cache
  - Reduce number of bus accesses
- Leads to problems with cache coherence

2. Multiport Memory

A multiport memory system employs separate buses between each memory module and each CPU.

It provides direct independent access of memory modules by each processor and logic is required to resolve conflicts. In this bus system, little or no modification to processors or modules required and memory access conflicts are resolved by assigning fixed priorities to each memory port. CPU 1 will have priority over CPU 2, CPU 2 will have priority over CPU 3, and CPU 4 will have lowest priority



Advantages

- Better performance
  - Each processor has dedicated path to each module
- Can configure portions of memory as private to one or more processors
  - Increased security

Disadvantages

- More complex
  - Extra login in memory system
- Expensive memory control logic

3. Cross bar switch

The CS organization consists of a number of cross points that are placed at intersections between processor buses and memory module paths. Each switch point has control logic to set up the transfer path between a processor and memory. It consists of multiplexer that select the data, address, and control from one CPU for communication with the memory module. The Hardware required to implement the switch can become quite large and complex.

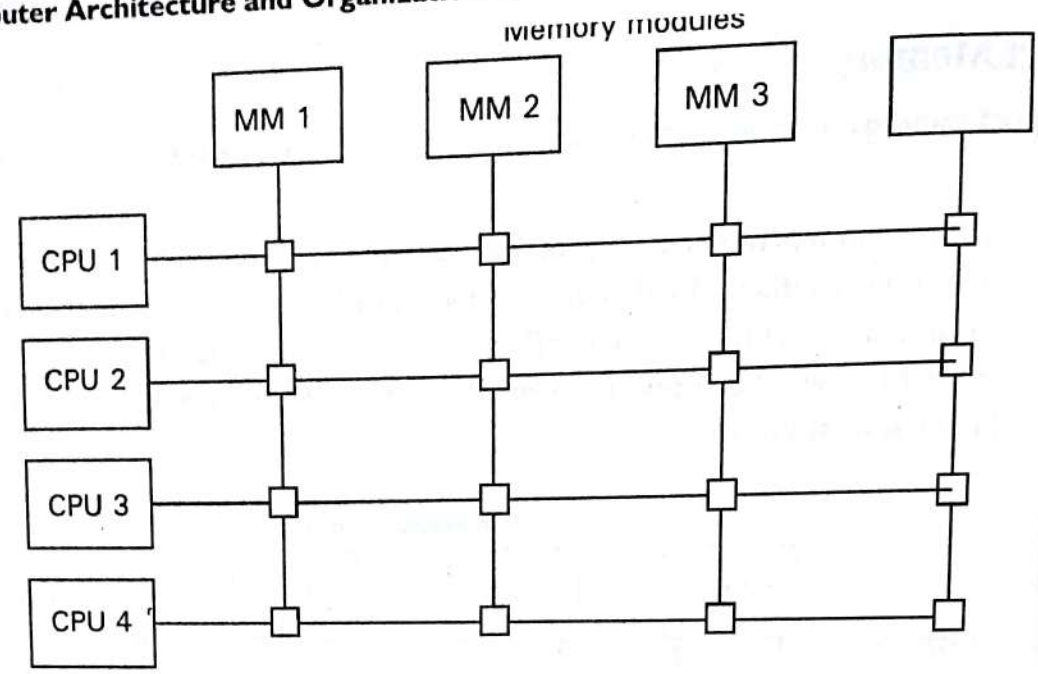
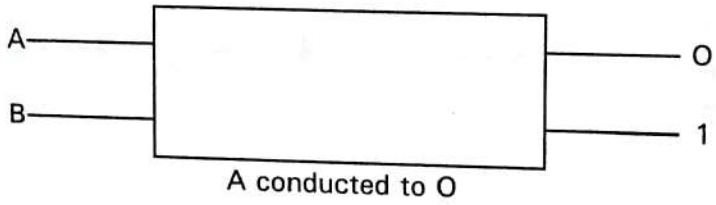


Figure 11.3: *Crossbar switch*

4. Multistage switching Network

- The basic component of a MSN is a two-input, two-output interchange switch.
- The 2X2 switch has two inputs (A,B), and two outputs (0,1). There are control signals associated with the switch that establish the interconnection between the input and output terminals.



- Using the 2 X 2 switch, it is possible to build a multistage network to control the communication between a number of sources and destination.

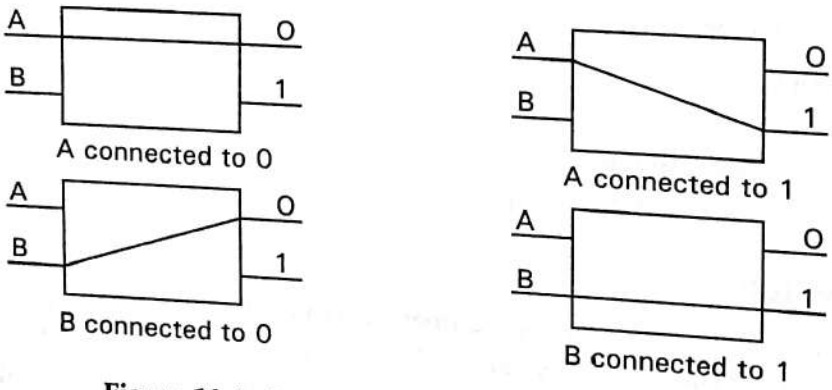


Figure 11.4: *Operation of a 2×2 interchange switch*

- Two processors P1 and P2 can be connected through switches to eight memory modules 000 through 111 as shown in figure given below.

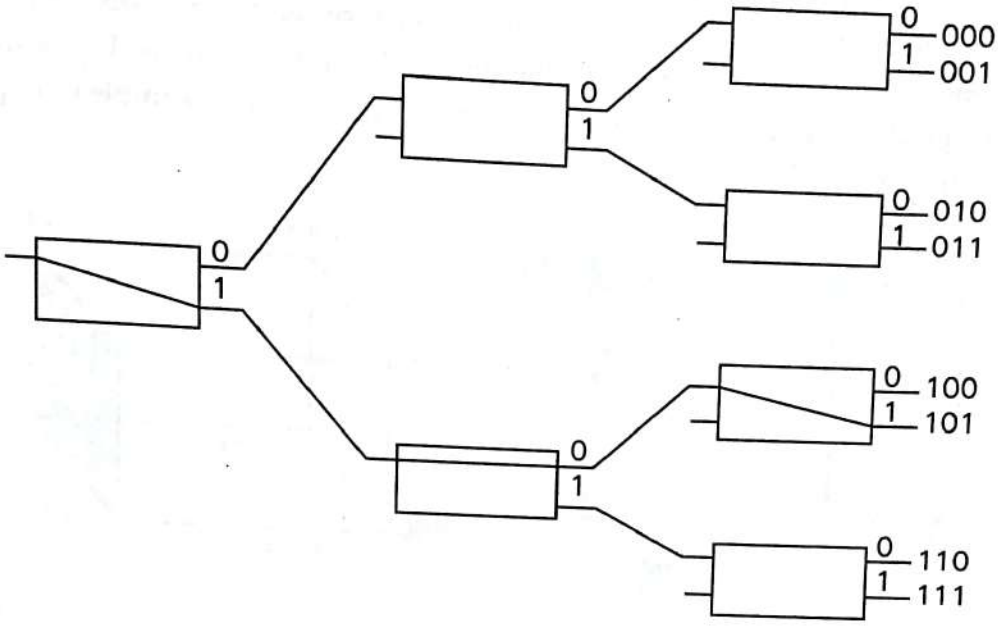


Figure 11.5: *Binary tree with 2×2 switches*

Omega Network

In omega network there is exactly one path from each source to any particular destination. Some request patterns, however, cannot be connected simultaneously (e.g. 000 and 001)

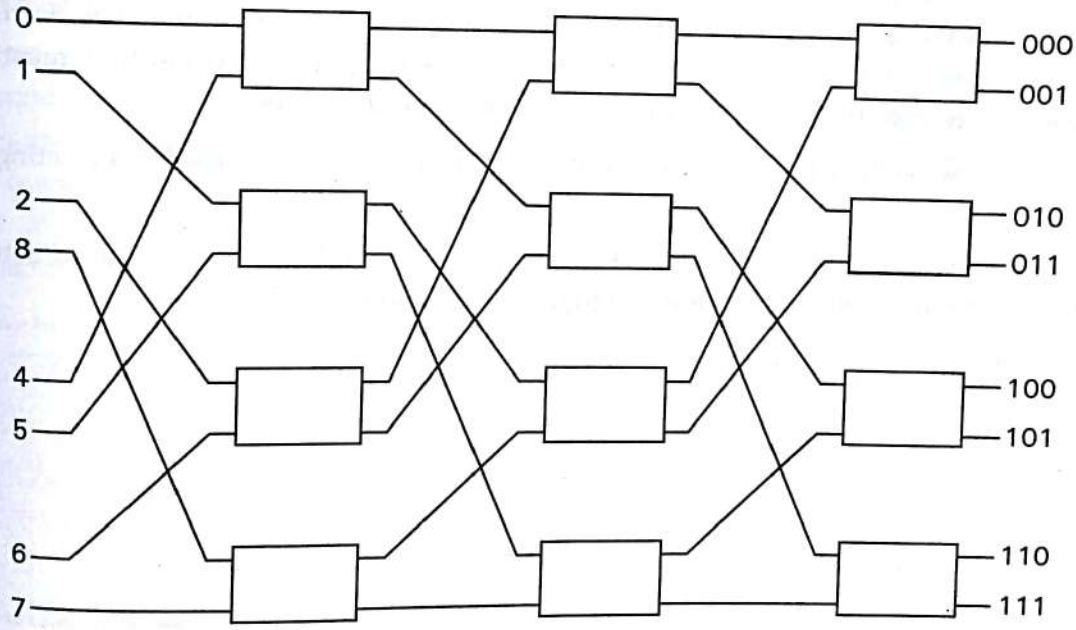
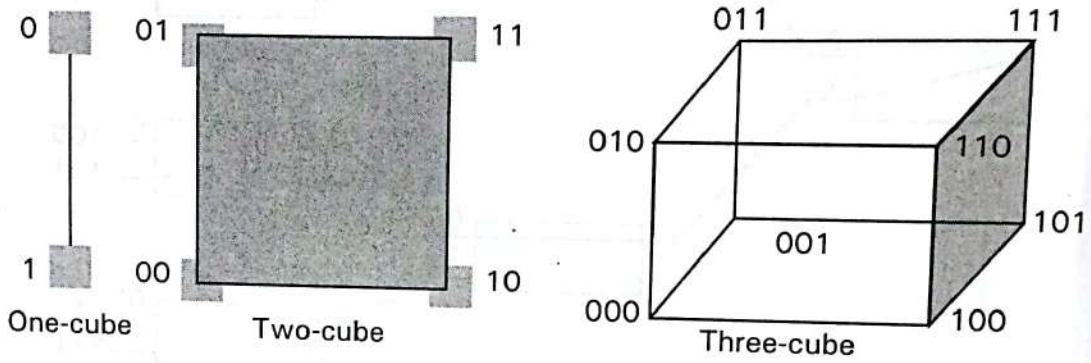


Figure 11.6: *8 × 8 omega switch network*

5. Hypercube Interconnection

The hypercube or binary n-cube multiprocessor structure is a loosely coupled system composed of $N = 2^n$ processors interconnected in a n-dimensional binary cube Each processor form a node of a cube. the hypercube structure for n = 1, 2, and 3 is shown in figure given below. Routing messages through an n-cube structure may take from one to n links from source node to

destination node. E.g. in a 3-cube structure, node 000 can communicate with 011 (from 000 to 001 to 011 or from 000 to 010 to 011). A routing procedure can be developed by computing the EX-OR of the source node address with the destination node address. E.g. a message at 010 going to 001 produces an EX_Or of the two addresses equal to 011. Example of hyper cube is the InteliPSC computer (n =7).



11.3 Inter Processor Arbitration(IA)

The processors in a shared memory multiprocessor system request access to common memory or other common resources through the system bus. If no other processor is currently utilizing the bus, the requesting processor may be granted access immediately. However, the requesting processor must wait if another processor is currently utilizing the system bus. Furthermore, other processors may request the system bus at the same time. Arbitration must then be performed to resolve this multiple contention for the shared recourses.

A single hardware device called bus controller or arbiter, is responsible for allocating time on the bus.

The arbitration logic would be part of the system bus controller or arbiter placed between the local bus and the system bus as shown in figure given below.

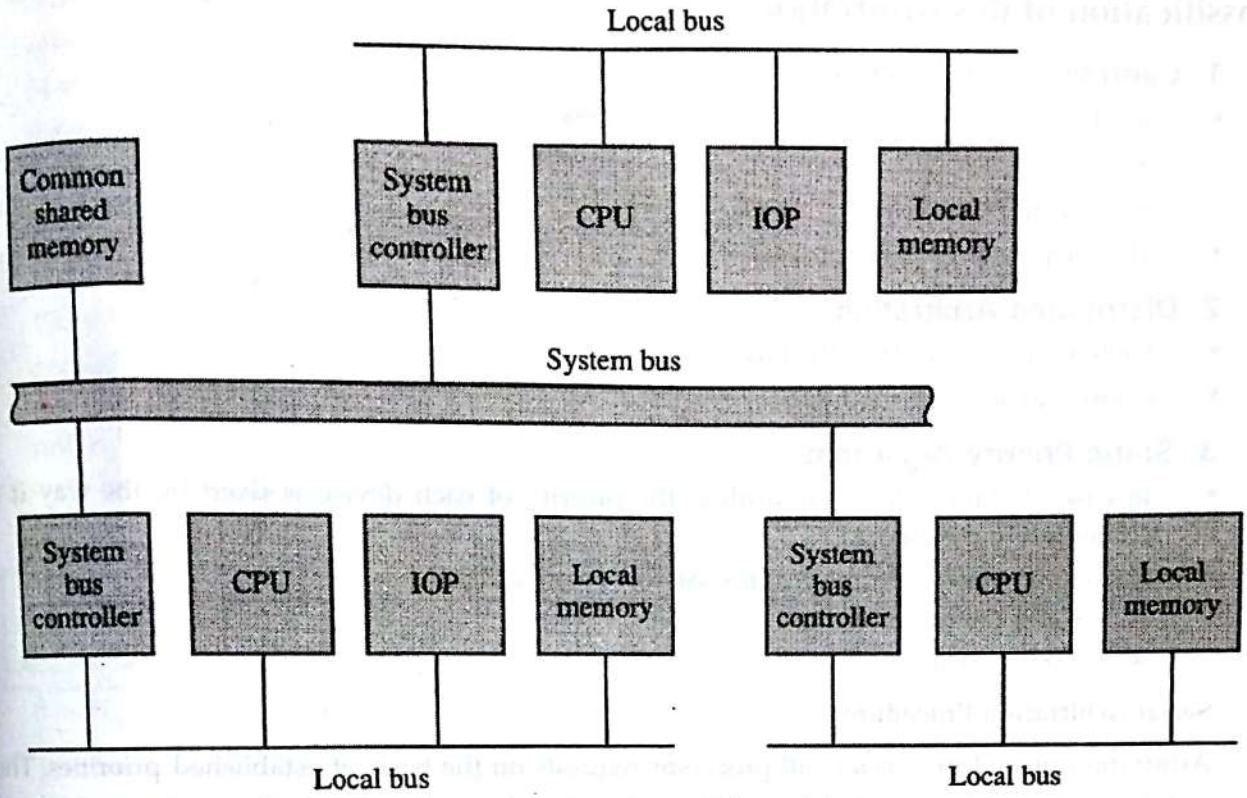


Figure 11.8: *System bus structure for multiprocessors.*

System Bus

A bus that connects major components in a multiprocessor system, like CPUs, IOPs, and memory, is called a system bus. A typical system bus consists of approximately 100 lines. These lines are divided into three functional groups: data, address, and control, and also power lines. For example, the IEEE796 multi bus system has 16 data lines, 24 address lines, 26 control lines, and 20 power lines, for a total of 86 lines. Data transfer method may be synchronous or asynchronous.

Synchronous Vs Asynchronous

Synchronous Bus	Asynchronous Bus
Each data item is transferred during a time slice known in advance to both source and destination units. Synchronization is achieved by driving both units from a common clock source.	Each data item being transferred is accompanied by handshaking control signal to indicate when data are transferred from source and received by destination.

Bus Types

- Dedicated: Separate data & address lines
- Multiplexed
  - Shared lines
  - Address valid or data valid control line

    Advantage: fewer lines

    Disadvantages: More complex control and ultimate performance

Classification of Bus Arbitration

1. Centralised Arbitration

- Single hardware device controlling bus access
  - Bus Controller
  - Arbiter
- May be part of CPU or separate

2. Distributed Arbitration

- Each module may claim the bus
- Control logic on all modules

3. Static Priority Algorithm

- In case of static priority algorithm, the priority of each device is fixed by the way it is connected to the bus.
- Following arbitration procedures use this priority algorithm:
  - Serial arbitration
  - Parallel arbitration

Serial Arbitration Procedure

Arbitration procedure services all processor requests on the basis of established priorities. The serial priority resolving technique is obtained from a daisy-chain connection of four arbiters of bus arbitration circuits similar to the priority interrupt logic. The processors connected to the system bus are assigned priority according to their position along the priority control line. Arbitration procedure services all processor requests on the basis of established priorities. The serial priority resolving technique is obtained from a daisy-chain connection of four arbiters of bus arbitration circuits similar to the priority interrupt logic. The processors connected to the system bus are assigned priority according to their position along the priority control line.

The figure given below shows the daisy-chain connection of four arbiters. It is assumed that each processor has its own bus arbiter logic with priority-in and priority-out lines. The priority-out (PO) of each arbiter is connected to the priority in (PI) of the next-lower-priority arbiter.

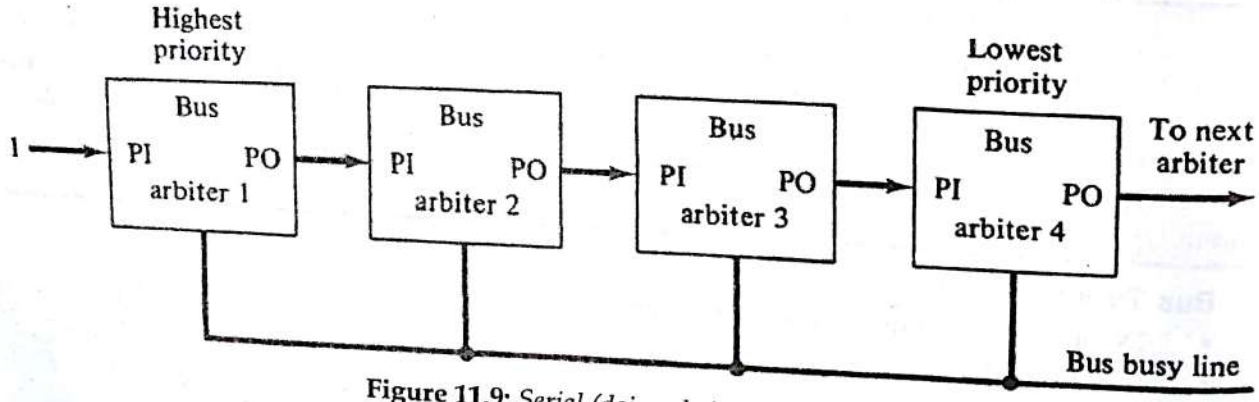


Figure 11.9: *Serial (daisy-chain) arbitration*

The PI of the highest priority unit will be at logic 1. PI=1 will always receive access to the system when it request it and PO=1. i.e highest priority is PI =1,PO=1 and lowest priority is PI =0, & PO=0. The processor whose arbiter has PI =1, & PO=0 is the one that is given control of the system bus. When the arbiter receives control of bus (because its PI =1, & PO=0) it examines the busy line. If the line is inactive, it means that no other processor is using the bus. The arbiter activates the busy line and its processor takes control of the bus.

Parallel Arbitration Logic

The PAP techniques uses an external priority encoder and a decoder. Each bus arbiter in the parallel scheme has a bus request output line and a bus acknowledge input line. The processor takes control of the bus if its acknowledge input line is enabled. The busy line provides an orderly transfer of control, as in the daisy-chaining case. Figure given below, shows the 2-bit code from the encoder output drives a 2 X 4 decoder which enables the proper acknowledge input to grant bus access to the highest-priority unit.

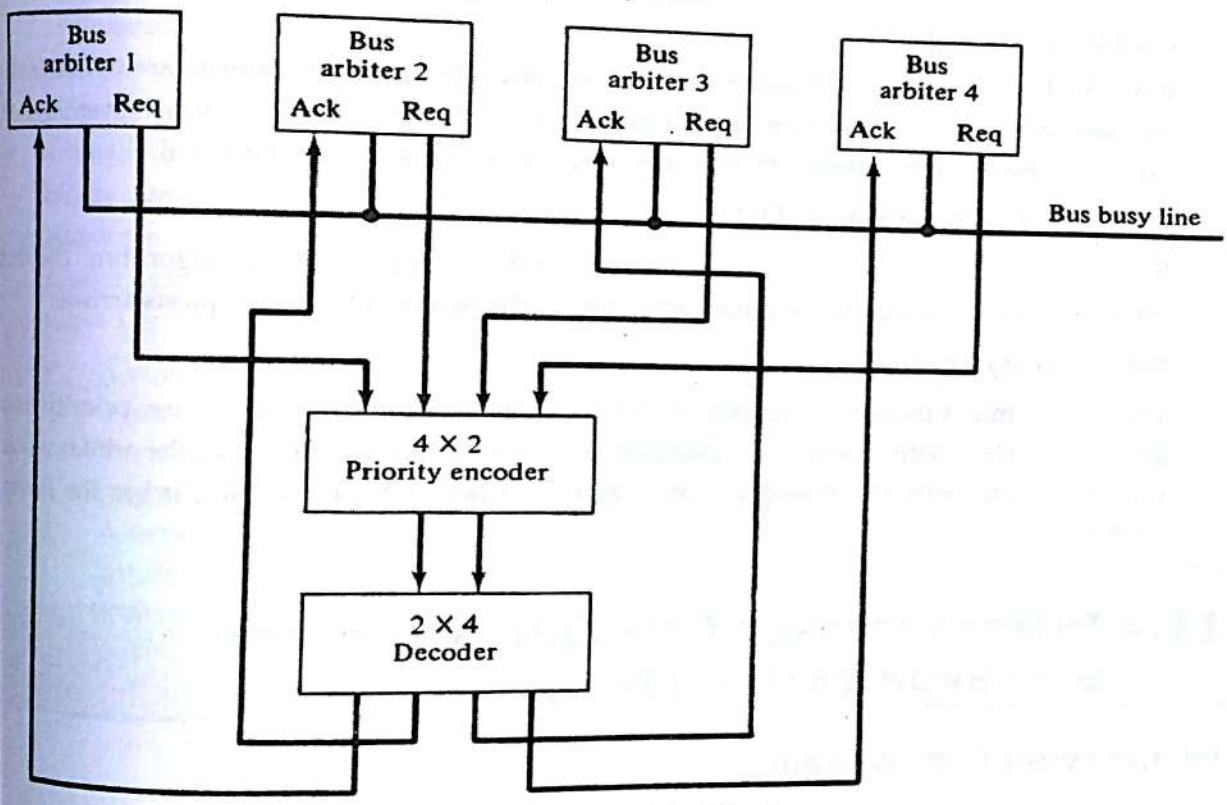


Figure 11.10: *Parallel arbitration*

4. Dynamic Arbitration Algorithm(DAA)

Serial and Parallel arbitration explained before use a static priority algorithm since the priority of each device is fixed by the way it is connected to the bus. But, a DAA gives the system the capability for changing the priority of the device while the system is in operation.

5 types of DAA:

- Time Slice
- Polling
- LRU
- First come First serve scheme

- Rotating daisy chain

TIME SLICE ALGORITHM

This algorithm allocates a fixed-length time slice of bus time that is offered sequentially to each processor. The service given to each system component with this scheme is independent of its location along the bus. No preference is given to any particular device since each is allocated the same amount of time to communicate with the bus.

POLLING

In Polling, the bus grant signal is replaced by a set of lines called poll lines which are connected to all units. These lines are used by the bus controller to define an address for each device connected to the bus. After a number of bus cycles, the polling process continues by choosing a different processor. The polling sequence is normally programmable, and as a result, the selection priority can be altered under program control.

Least Recently Used(LRU)

With this LRU, no processor is given priority over any other since the priority are dynamically changed to give every device an opportunity to access the bus. This algorithm gives highest priority to the requesting device that has not used the bus for the longest interval.

First Come First Serve Scheme(FIFO)

In FIFO, requests are served in the order received. To implement this algorithm, the bus controller establishes a queue arranged according to the time that the bus requests arrives.

Rotating Daisy Chain

This algorithm is a dynamic extension of the daisy-chain algorithm. Each arbiter priority for a given bus cycle is determined by its position along the bus priority line from the arbiter whose processor is currently controlling the bus. Once an arbiter releases the bus, it has the lowest priority.

11.4 INTERPROCESSOR COMMUNICATION AND SYNCHRONIZATION (ICS)

Interprocessor Communication

The various processors in a multiprocessor system must be provided with a facility for communicating with each other. A communication path can be established through common I/O channels. In a shared memory multiprocessor system (MP), the most common procedure is to set aside a portion of memory that is accessible to all processors. In addition to shared memory, a MP system may have other shared resources, e.g. a magnetic disk unit connected to an IOP may be available to all CPUs. To prevent conflicting use of shared resources by several processors there must be a provision for assigning resources to the processor. There are three organizations that have been used in design of operating system for multiprocessors:

- Master-Slave Configuration,
- Separate Operating System, and
- Distributed Operating System.

- In a ***Master-Slave mode***, one processor, designated the master, always executes the OS functions. The remaining processors as slaves, do not perform OS functions. If a slave processor needs an OS, it must request it by interrupting the master and waiting until current program can be interrupted.
- In the ***Separate Operating System***, each processor can execute the OS routines it needs. The organization is more suitable for loosely coupled system.
- In the ***Distributed Operating System***, the OS routines are distributed among the available processors. However, each special OS function is assigned to only 1 processor at a time.
- In a loosely coupled MP, the memory is distributed among the processors and there is no shared memory for passing information. The communication between processors is by means of message passing through I/O channels.

Interprocessor Synchronization (ICS)

The instruction set of a MP contains basic instructions that are used to implement communication and synchronization between cooperating processes. Here, synchronization refers to the special case where data used to communicate between processors is control information. Synchronization is needed to enforce the correct sequence of process and to ensure mutually exclusive access to shared writable data. One of the most popular methods is use of binary *Semaphore*. A properly functioning MP system must provide a mechanism that will guarantee orderly access to shared memory and other shared resources. This is necessary to protect data from being changed simultaneously by two or more processors and mechanism is termed as ***mutual exclusion***.

- A binary variable called a *semaphore* is often used to indicate whether or not a processor is executing a critical section. A is a software-controlled flag that is stored in memory location that all processors can access. Semaphore = 1 means processor is executing *critical program*, so that shared memory is not available to other processors.
- A semaphore can be initialized by means of a test and set instruction in conjunction with hardware *lock* mechanism. Assume that the semaphore is LSB of memory word whose address is symbolized by SEM and TSL is mnemonic designates "Test and set while locked:
- ***TSL SEM*** will be executed in two memory cycles:

$R \leftarrow M[SEM]$	Test semaphore
$M[SEM] \leftarrow 1$	Set semaphore

11.5 Cache Coherence

Caches are critical to modern high-speed processors as Multiple copies of a block can easily get inconsistent because of processor writes, I/O writes, etc. The figure given below demonstrates the inconsistency problem occurred during right through and write back cache policy.

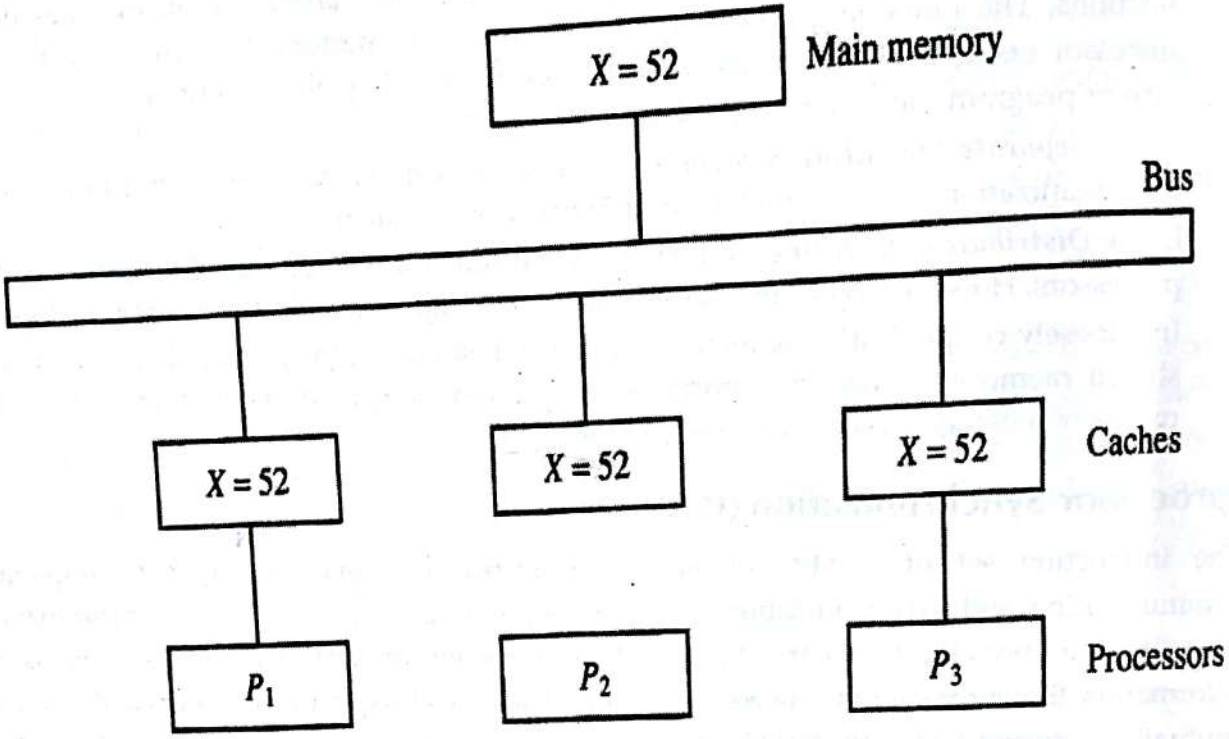
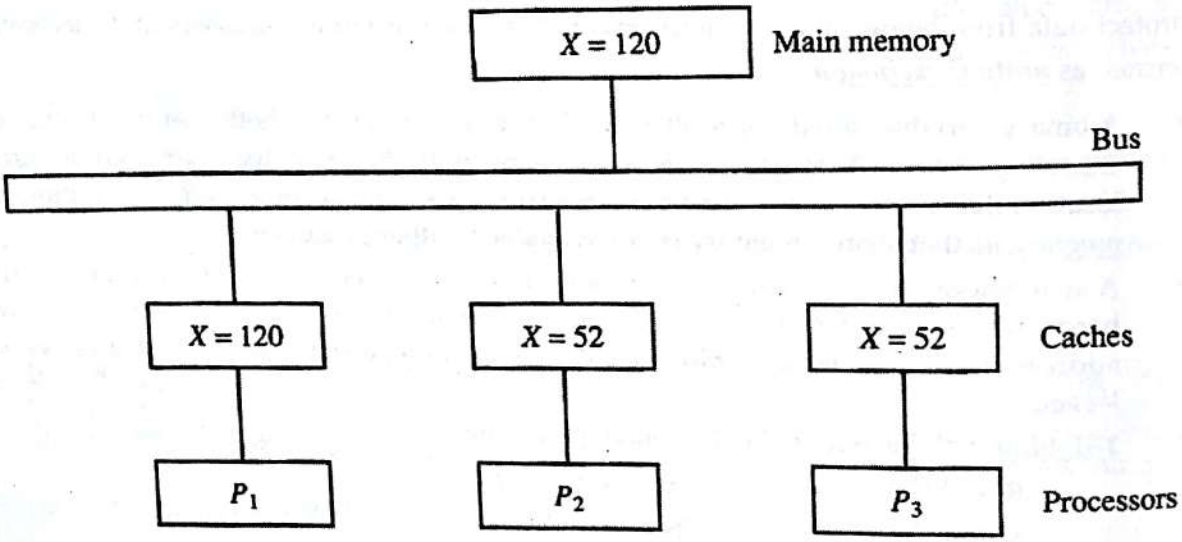
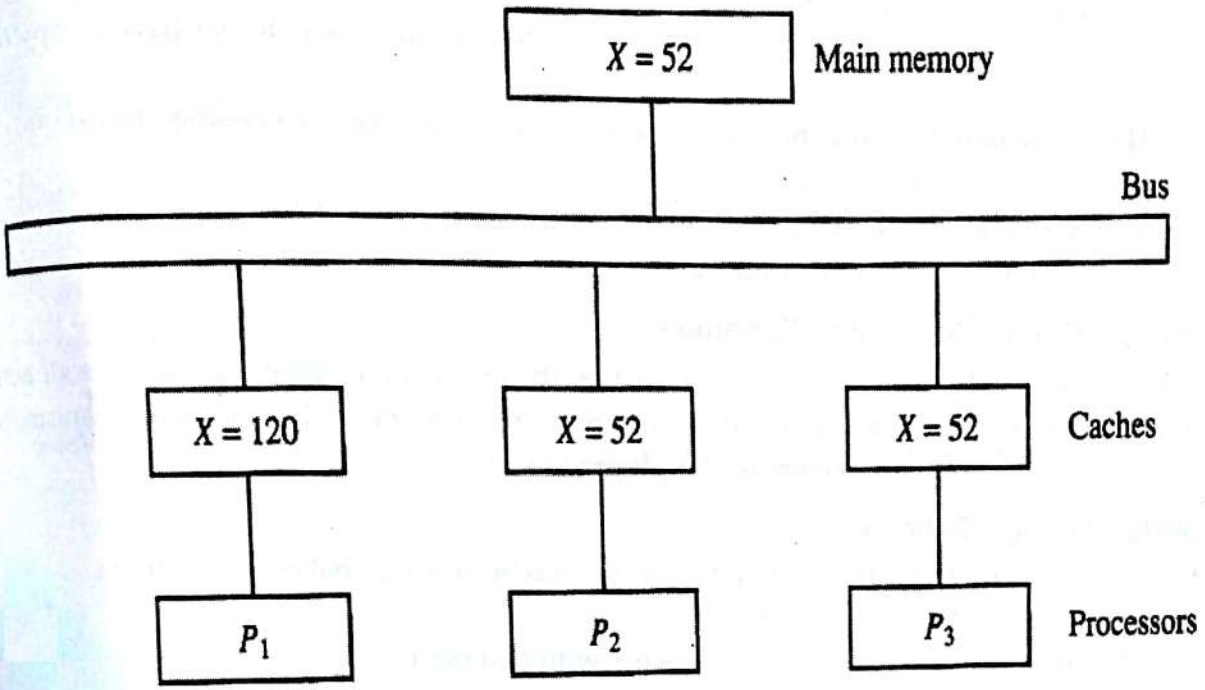


Figure 11.11: *Cache configuration after a load on X*



(a) With write-through cache policy



(b) With write-back cache policy

Figure 11.12: *Cache configuration after a store to X by processor P_1.*

Cache Coherence Solutions

Cache coherence problem could be solved by Software program or by using hardware system.

- **Software based cache coherence solution:**
  - It is compiler based or with run-time system support and it works with or without hardware assist. It has tough problem because perfect information is needed in the presence of memory aliasing and explicit parallelism. Thus need to focus on hardware based solutions as they are more common.
  - A simple scheme is to disallow private cache for each processor and have a shared cache memory associated with main memory. Every data access is made to the shared cache. This method violates the principal of closeness of CPU to cache and increases the average memory access time. In effect, this scheme solves the problem by avoiding it.
  - For performance consideration it is desirable to attach private cache to each processor. One scheme that has been used allows only non-shared and read only data to be stored in caches. Such items are called cacheable. Shared writable data are non-cacheable. The compiler must tag data as a either cacheable or non-cacheable, and the system hardware makes sure that only cacheable data are stored in caches. The non-cacheable data remain in main memory. This method restricts the type of data stored in caches and introduces an extra software overhead that may degrade performance.
  - A scheme that allows writeable data to exist in at least one cache is a method that employs aa centralized global table in its complier. The status of memory blocks is stored in the central global table. each block is identified as read-only 9RO) or read

and write (RW). All caches can have copies of blocks identified as RO. Only one cache can have a copy of an RW block. Thus, if the data are updated in the cache with an RW block, the other caches are not affected because they do not have a copy of this block.

- **Hardware based cache coherence solution:** The schemes can be classified based on :
  - Shared caches vs Snoopy schemes
  - Write through vs. write-back (ownership-based) protocols
  - update vs. invalidation protocols

Snoopy Cache Coherence Schemes

A distributed cache coherence scheme based on the notion of a snoop that watches all activity on a global bus, or is informed about such activity by some global broadcast mechanism. Most commonly used method in commercial multiprocessors

Write Through Schemes

- All processor writes result in update of local cache and a global bus write that :
  - updates main memory and
  - invalidates/updates all other caches with that item
- **Advantage :** Simple to implement
- **Disadvantages :** Since ~15% of references are writes, this scheme consumes tremendous bus bandwidth . Thus only a few processors can be supported.
  - Need for dual tagging caches in some cases

Write-Back/Ownership Schemes

- When a single cache has ownership of a block, processor writes do not result in bus writes thus conserving bandwidth.
- Most bus-based multiprocessors nowadays use such schemes.

Invalidation vs. Update Strategies

1. **Invalidation :** On a write, all other caches with a copy are invalidated
2. **Update :** On a write, all other caches with a copy are updated

- Invalidation is bad when :
  - single producer and many consumers of data.
- Update is bad when :
  - multiple writes by one PE before data is read by another PE.
  - Junk data accumulates in large caches (e.g. process migration).
- Overall, invalidation schemes are more popular as the default.

Bibliography

Computer System Architecture, M. Morris Mano, Seventh Edition, Prentice Hall, 2015.

Computer Organisation and Architecture, William Stallings, Ninth Edition, Prentice Hall, 2015.

Gorsline, G. W., *Computer Organization:* Hardware/Software, 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1986.

Hays, J.F., *Computer Architecture and Organization*, 2nd ed. New York: McGraw-Hill, 1988.

Hwang, K., and F. A. Briggs, *Computer Architecture and Parallel Processing*. New York: McGraw-Hill, 1984.

Kain, R., *Computer Architecture*: Software and Hardware, Vol. 2. Englewood Cliffs, NJ: Prentice Hall, 1989.

Exam Question- BSc. CSIT

Tribhuwan University

Office of Dean and Controller of Examination

Attempt any two questions [20]

1. Write down the code to evaluate Y=A/B+CD+E(F/G) in three address, one address and zero address instruction format.
2. Explain the mapping function. Why replacement algorithm is used in set associative mapping? Explain with example.
3. Differentiate between interrupt driven I/O with programmed I/O. Explain with example how data transfer is performed in direct memory access (DMA).

Attempt any ten questions [60]

1. Explain the floating point addition and subtraction with flowchart.
2. Comparison between RISC and CISC.
3. What are the key characteristics of computer memory system? Explain.
4. Explain input/output interface with example.
5. Comparer the different types of addressing modes with example.
6. Explain the microprogram control unit with example.
7. Explain the non-restoring division algorithm with example.
8. Explain the input-output processor with block diagram.
9. Explain the data manipulation instruction with example.
10. Explain with example of Arithmetic microoperations.
11. Explain memory management hardware with example.
12. Write short notes on the following:
    a) Virtual memory
    b) Register organization

Exam Question- BIM

Tribhuwan University

Office of Dean and Controller of Examination

Exercise problems

1. The 8-bit registers AR,BR,CR and DR initially have following values: AR=11110010, BR=11111111, CR=10111001, DR=11101010 Determine the 8-bit value in each register after the execution of the following sequence of micro operation: [16]
   AR <- AR+BR
   CR <- CR V DR, BR <- BR+1
   AR <- AR+CR, DR <- DR+1
2. Perform the operations (-19)x(+21) by using signed magnitude multiplication algorithm.
3. Write a program loop, using a pointer and counter that clears the content of hexadecimal location 500 through 5FF to 0.
4. What is associative memory? Given a cache memory with access time 200ns and RAM with access than 2000ns, if hit ratio is 90% calculate the average memory access time.

Comprehensive Answer Questions

1. What is the advantages of common bus system? Explain with diagram. What is vector processing? [10]
2. How are priorities of interrupt handled? What do you means by instruction set completeness?

Brief Answer Questions

1. What is the importance of complement in digital computer? [10]
2. What is the register value of A after arithmetic right shift if A=10011100?
3. What is the purpose of location counter in assembler?
4. How many bits are required to address 4096 bytes of memory?
5. What is the meaning of dynamic microprogramming?
6. What is the purpose of strobe pulse in asynchronous data transfer?
7. Write the major difficulties that causes the instruction pipeline to deviate from normal operation.
8. What is the principles behind the implementation of cache memory?
9. What is the function of IOP?
10. What is critical section?

Exercises

Chapter 1

1. Perform the following conversions (Convert from binary to decimal a to g and decimal to binary h to j):
   a. $101_2 = N_{10}$
   b. $11001_2 = N_{10}$
   c. $1111_2 = N_{10}$
   d. $10011110_2 = N_{10}$
   e. $10101_2 = N_{10}$
   f. $101101_2 = N_{10}$
   g. $11111110_2 = N_{10}$
   h. $100_{10} = N_2$
   i. $225_{10} = N_2$
   j. $129_{10} = N_2$
2. Using the double-dabble method, convert the following binary numbers to decimal equivalents.
   a. 100001
   b. 101101101
   c. 111111
   d. 11110000
   e. 1010101010
   f. 100100
3. Using the remainder method, convert the following decimal system numbers to their binary equivalents.
   a. 25
   b. 54
   c. 432
   d. 69
   e. 12008
4. Convert the following octal numbers to hexadecimal:
   a. 6
   b. 12
   c. 367
   d. 5555
   e. 123456
5. Convert the following hexadecimal numbers to decimal:
   a. AA
   b. FEDCBA
   c. D0-BE-D0
   d. B8C
   e. FF-FF-FF-FA
   f. BABE
6. Convert the following octal (base-8) numbers to decimal:
   a. 4
   b. 10
   c. 777
   d. 05726
7. Convert the following decimal numbers to octal:
   a. 5
   b. 9
   c. 625
   d. 1024
8. Convert the following octal numbers to binary:
   a. 107
   b. 2746

c. 66542 d. 76543210

9. Convert the following binary numbers to octal:
   a. 110 b. 011101
   c. 010101011 d. 1111011101101
   e. 111110101100011010001
10. Convert the following hexadecimal numbers to octal:
   a. 2 b. 8
   c. 10 d. EF
   e. 2A4F2D
11. Convert the following decimal numbers to hexadecimal:
   a. 9 b. 32
   c. 73 d. 255
   e. 1025 f. 65536
12. Perform the arithmetic operations (+42) + (-13) and (-42) – (-13) in binary using signed-2's complement representation for negative numbers.
13. Perform the arithmetic operations (+70) + (+80) and (-70) + (-80) with binary numbers in signed -2's complement representation. Use eight bits to accommodate each number together with its sign. Show that overflow occurs in both cases, that the last two carries are unequal, and that there is a sign reversal.
14. Differentiate between fixed point and floating point representation.
15. What is an error detection code? Explain with example.
16. Differentiate between parity generator and parity checker?
17. Derive the circuits for a 3-bit parity generator and 4-bit parity checker using an even-parity bit.

Chapter 2

1. Represent the following conditional control statement by two register transfer statements with control functions.
   If (P = 1) then (R1 ← R2) else if (Q = 1) then (R1←R3)
2. Design a 4-bit combinational circuit decrementer using four full-adder circuits.
3. What are the different types of microoperations? Explain each in brief.
4. Design 8 bit binary adder /subtractor circuit and explain its operation with an example.
5. Explain Arithmetic circuit with the help of a block diagram.
6. Draw the block diagram for the hardware that implements the following statements:
   x + yz: AR ← AR + BR.

where AR and BR are two n-bit registers and x, y, and z are control variables. Include the logic gates for the control function. (Remember that the symbol + designates an OR operation in a control or Boolean function but that it represents an arithmetic plus in a microoperation.)

7. Show the hardware that implements the following statement. Include the logic gates for the control function and a block diagram for the binary counter with a count enable input.

   $xyT_0 + T_1 + y'T_2$: $AR \leftarrow AR + 1$.

8. Explain the arithmetic logic shift unit with the help of block diagram and function table.

Chapter 3

1. Explain the instruction format of basic computer.
2. Differentiate between direct and indirect addressing modes.
3. What do you mean by interrupt? Explain instruction cycle with the help of flowchart.
4. Explain the process of fetch and decode with the help of block diagram and register transfer.
5. What is the process of I/O communication in basic computer? Explain.
6. Explain basic computer operation with the help of flowchart.
7. Design control logic gates associated with memory read and memory write operation.
8. Design control logic gates associated with program counter PC in basic computer.

Chapter 4

1. What is assembler? Explain first pass assembler with the help of flowchart.
2. Why second pass assembler is required? Explain with the help of flowchart.
3. A line of code in an assembly language program is as follows:

   DEC – 35

   a. Show that four memory words are required to store the line of code and give their binary content.
   b. Show that one memory word stores the binary translated code and give its binary content.

4. Write a program to multiply two unsigned positive numbers, each with 16 significant bits, to produce an unsigned double-precision product.

5. Write a program to multiply two signed numbers with negative numbers being initially in signed-2's complement representation. The product should be single-precision and signed-2's complement representation if negative.
6. Write a program to subtract two double-precision numbers.
7. Write a program that evaluates the logic exclusive-OR of two logic operands.
8. Write a program for the arithmetic shift-left operation. Branch to OVF if an overflow occurs.

Chapter 5

1. Differentiate between hardwired control unit and microprogram control unit.
2. Explain microprogram control organization with the help of block diagram.
3. What is address sequencer? Explain how address sequencer selects the address for CAR in four different ways?
4. Explain computer hardware configuration for microprogram based control unit.
5. Explain microinstruction format of control memory.
6. Explain the operation of microoperation field decoding.
7. Explain microprogram sequencer for a control memory with the help of block diagram.

Chapter 6

1. What are the different types of CPU organization? Explain each organizations in detail.
2. Differentiate between stack register organization and stack memory organization.
3. Explain reverse polish notation with an example.
4. Convert the following arithmetic expressions from reverse Polish notation to infix notation.
   a. A B C D E + * - /
   b. A B C D E * / - +
   c. A B C * / D - E F / +
   d. A B C D E F G + * + * + *
5. Convert the following numerical arithmetic expression into reverse Polish notation and show the stack operations for evaluating the numerical result.
   (3 + 4) [10(2 + 6) + 8]
6. Write down the code to evaluate Y = A/B + CD + E(F/G) in three address, two address, one address and zero address instruction format.
7. Why addressing modes are required? Explain different types of addressing modes.

8. A two-word instruction is stored in memory at an address designated by the symbol W. The address field of the instruction (stored at W + 1) is designated by the symbol Y. THe operand used during the execution of the instruction is stored at an address symbolized by Z. An index register contains the value X. Show how Z is calculated from the other addresses if the addressing mode of the instruction is

   a. direct b. indirect c. relative d. indexed

9. Differentiate between RISC and CISC architecture.
10. What is register window overlapping and how it makes the process of procedure call and return efficient?

Chapter 7

1. What is parallel processing? How it can be achieved in a single processor system?
2. Explain Flynn's classification in detail.
3. What is pipeline? How it helps to execute a task in a faster way? Explain with example.
4. What is segment. Prove that speedup ration is equal to no of segment.
5. Explain pipeline for floating point addition and subtraction with the help of block diagram and an example.
6. In certain scientific computations it is necessary to perform the arithmetic operation (Ai + Bi) (Ci + Di) with a stream of numbers. Specify a pipeline configuration to carry out this task. List the contents of all registers in the pipeline for i = 1 through 6.
7. Draw a space-time diagram for a six-segment pipeline showing the time it takes to process eight tasks.
8. Determine the number of clock cycles that it takes to process 200 tasks in a six-segment pipeline.
9. A nonpipeline system takes 50 ns to process a task. The same task can be processed in a six-segment pipeline with a clock of 10 ns. Determine the speedup ratio of the pipeline for 100 tasks. What is the maximum speedup that can be achieved?
10. What is Instruction pipeline? Explain 4 segment CPU pipeline with the help of flowchat.
11. What are pipeline conflicts and explain the measure of pipeline conflicts.
12. Explain the concept of vector processing and memory interleaving.
13. Explain the concept of SIMD array processor.

Chapter 8

14. Explain the process of sign magnitude addition and subtraction. List out its limitations.
15. Why signed 2's complement addition/subtraction is implemented in our computer? Explain with the help of hardware component and flowchart.
16. Show the steps of multiplication process using sign magnitude multiplication algorithm for the following numbers: a) -9 X 15 b) 12 X -16
17. Explain working principle of Booth Algorithm and its flowchart for multiplication.
18. Show the steps of multiplication process using booth algorithm of the following binary numbers:
    Y = 8 × 10
19. Show the step-by-step multiplication process using Booth's algorithm, when +14 is multiplied by -14. Assume 5-bit registers that hold signed numbers.
20. Explain nonrestoring division algorithm with example.
21. Explain restoring division algorithm with example.
22. Differentiate between nonrestoring and restoring division algorithm.

Chapter 9

1. Explain the communication link between processor and peripherals with the help of I/O bus and interface modules.
2. What are the three possible modes to transfer the data to and from peripherals? Explain.
3. What are the different types of I/O commands? Explain.
4. Differntiate between memory space and address space.
5. What are the modes of data transfer? Explain each in brief.
6. Why DMA is required? Expalin the working operation of DMA with the help of block diagram.
7. What is the role of input-output processor (IOP) in computer system? Explain.
8. Explain synchronous and asynchronous data transmission.
9. Explain hardware and software interrupt.
10. Explain CPU and I/O communication with the help of block diagram.

Chapter 10

1. Why memory hierarchy is required in a computer system? Explain its block diagram.
2. What is address decoding?

3. Show the memory organization (1024 bytes) of a computer with four 128×8 RAM Chips and 512×8 ROM chip. How many address lines are required to access memory?
4. A computer employs RAM chips of 256 × 8 and ROM chips of 1024 × 8. The computer system needs 2K bytes of RAM, 4K bytes of ROM, and four interface units, each with four registers. A memory-mapped I/O configuration is used. The two highest-order bits of the address bus are assigned 00 for RAM, 01 for ROM, and 10 for interface registers.
   a. How many RAM and ROM chips are needed?
   b. Draw a memory-address map for the system.
   c. Give the address range in hexadecimal for RAM, ROM, and interface.
5. An 8-bit computer has a 16-bit address bus. The first 15 lines of the address are used to select a bank of 32K bytes of memory. The high-order bit of the address is used to select a register which receives the contents of the data bus. Explain how this configuration can be used to extend the memory capacity of the system to eight banks of 32K bytest each, for a total of 256K bytes of memory.
6. What is associative memory? How it is faster than other memory?
8. What is hit ratio of Cache momory? Explain different types of mapping used in cache.
9. Differentiate between write through and write back method used to update main memory.
10. What is virtual memory ? Explain its address space and memory space .
11. Explain the organization of memory mapping table in a paged system.
12. How associative memory page table is efficient than random access memory page table? Explain.
13. Explain the operation of page replacement in virtual memory system.
14. What is memory management hardware? Explain with the help of block diagram.
15. What is memory protection? Explain different access rights for memory protection.
16. The logical address space in a computer system consists of 128 segments. Each segment can have up to 32 pages of 4K words in each. Physical memory consists of 4K blocks of 4K words in each. Formulate the logical and physical address formats.

Chapter 11

1. Define multiprocessing. Explain its different ways of implementation.
2. Differentiate between multiprocessor and multicomputer? Why we choose multiprocessor?
3. Explain the concept of shared memory and distributed memory multiprocessor.
4. Differentiate between time-shared common bus multiport memory interconnection structure.

5. Explain cross bar switch interconnection structure and multistage switching network.
6. How hypercube interconnection structure is better than previous interconnection structure.
7. Differentiate between synchronous and asynchronous bus.
8. Explain serial arbitration procedure and parallel arbitration procedure.
9. Explain the process of interprocessor communication and synchronization.
10. What is cache coherence problem and its solutions.

■■■

KEC's B.Sc. CSIT Text Book Series

First Semester
- Introduction to Informatioin Technology
- C Programming
- Digital Logic
- Mathematics I
- Physics

Second Semester
- Discrete Structure
- Object Oriented Progamming
- Microprocessor
- Mathematics II
- Statistics I

Third Semester
- Data Structure and Algorithms
- Numerical Method
- Computer Architecture
- Computer Graphics
- Statistics II

Fourth Semester
- Theory of Computation
- Computer Networks
- Operating System
- Database Management System
- Artificial Intelligence

Fifth Semester
- Design and Analysis of Algorithms
- System Analysis and Design
- Cryptography
- Simulation and Modeling
- Web Technology
- Elective I

Sixth Semester
- Software Engineering
- Complier Design and Construction
- E-Governance
- NET Centric Computing
- Technical Writing
- Elective II

Seventh Semester
- Advanced Jave Programming
- Data Warehousing and Data Mining
- Principles of Management
- Project Work
- Elective III

Eighth Semester
- Advanced Database
- Internship
- Elective IV
- Elective V

KEC
Publication and Distribution Pvt. Ltd.
Kathmandu, Nepal, Tel.: 01-4168301, 01-4241777
E-mail: kecpublication14@gmail.com