COMPUTER ORGANIZATION

MASTER OF COMPUTER APPLICATIONS (MCA)

SEMESTER-I, PAPER-V

LESSON WRITERS

Dr. U. Surya Kameswari
Assistant Professor
Department of CS&E
University College of Sciences
Acharya Nagarjuna University

Dr. Neelima Guntupalli
Assistant Professor
Department of CS&E
University College of Sciences
Acharya Nagarjuna University

Dr. Kampa Lavanya
Assistant Professor
Department of CS&E
University College of Sciences
Acharya Nagarjuna University

Dr. Vasantha Rudramalla
Faculty
Department of CS&E
University College of Sciences
Acharya Nagarjuna University

EDITOR

Dr. U. Surya Kameswari
Assistant Professor
Department of CS&E
University College of Sciences
Acharya Nagarjuna University

DIRECTOR, I/c.

Prof. V. Venkateswarlu
M.A., M.P.S., M.S.W., M.Phil., Ph.D.
Professor
Centre for Distance Education
Acharya Nagarjuna University
Nagarjuna Nagar 522 510

Ph: 0863-2346222, 2346208
0863- 2346259 (Study Material)
Website www.anucde.info
E-mail: anucdedirector@gmail.com

MCA: COMPUTER ORGANIZATION

First Edition : 2025

No. of Copies :

Published by:

Prof. V. VENKATESWARLU
Director, I/c
Centre for Distance Education,
Acharya Nagarjuna University

Printed at:

FOREWORD

Since its establishment in 1976, Acharya Nagarjuna University has been forging ahead in the path of progress and dynamism, offering a variety of courses and research contributions. I am extremely happy that by gaining 'A+' grade from the NAAC in the year 2024, Acharya Nagarjuna University is offering educational opportunities at the UG, PG levels apart from research degrees to students from over 221 affiliated colleges spread over the two districts of Guntur and Prakasam.

The University has also started the Centre for Distance Education in 2003-04 with the aim of taking higher education to the door step of all the sectors of the society. The centre will be a great help to those who cannot join in colleges, those who cannot afford the exorbitant fees as regular students, and even to housewives desirous of pursuing higher studies. Acharya Nagarjuna University has started offering B.Sc., B.A., B.B.A., and B.Com courses at the Degree level and M.A., M.Com., M.Sc., M.B.A., and L.L.M., courses at the PG level from the academic year 2003-2004 onwards.

To facilitate easier understanding by students studying through the distance mode, these self-instruction materials have been prepared by eminent and experienced teachers. The lessons have been drafted with great care and expertise in the stipulated time by these teachers. Constructive ideas and scholarly suggestions are welcome from students and teachers involved respectively. Such ideas will be incorporated for the greater efficacy of this distance mode of education. For clarification of doubts and feedback, weekly classes and contact classes will be arranged at the UG and PG levels respectively.

It is my aim that students getting higher education through the Centre for Distance Education should improve their qualification, have better employment opportunities and in turn be part of country's progress. It is my fond desire that in the years to come, the Centre for Distance Education will go from strength to strength in the form of new courses and by catering to larger number of people. My congratulations to all the Directors, Academic Coordinators, Editors and Lesson-writers of the Centre who have helped in these endeavors.

Prof. K. Gangadhara Rao
M.Tech., Ph.D.,
Vice-Chancellor I/c
Acharya Nagarjuna University

MASTER OF COMPUTER APPLICATIONS (MCA)

Semester-I, Paper-V

105MC24: COMPUTER ORGANIZATION

SYLLABUS

Unit-I

Digital Logic Circuits: Digital Computers, Logic Gates, Boolean Algebra, Map Simplification, Combinational Circuits, Flip-Flops, Sequential Circuits.

Digital Components: Integrated Circuits, Decoders, Multiplexers, Registers, Shift Registers, Binary Counters, Memory Unit.

Data Representation: Data Types, Complements, Fixed Point Representation, Floating Point Representation, Other Binary Codes, error Detection Codes.

Unit-II

Register Transfer and Microoperations: Register Transfer Languages, Register Transfer, Bus and Memory Transfer, Arithmetic Micro Operations, Logic Micro Operations, Shift Micro Operations, Arithmetic Logic Shift Unit

Basic Computer Organization and Design: Instruction Codes, Computer Registers, Computer Instructions, Timing and Control, Instruction Cycle, Memory Reference Instructions, Input- Output and Interrupt.

Unit-III

Microprogrammed Control: Control Memory, Address Sequencing, Micro Program Example, Design of Control Unit.

Central Processing Unit: Introduction, General Register Organization, Stack Organization, Instruction Format, Addressing Modes, Data Transfer and Manipulation, Program Control.

Unit-IV

Computer Arithmetic: Addition, Subtraction, Multiplication, Division Algorithms, Floating Point Arithmetic Operations.

Input-Output Organization: Peripheral Devices, Input-Output Interface, Asynchronous Data Transfer, Modes of Transfer, Priority Interrupt.

Memory Organization: Memory Hierarchy, Main Memory, Auxiliary memory, Associative Memory, Cache Memory.

Prescribed Book:

M.Morris Mano, “Computer System Architecture”, 3rd Edition, Pearson Education (2008).

Reference Books:

1) V. Rajaraman, T. Radha Krishnan, “Computer Organization and Architecture”, PHI
2) BehroozParhami, “Computer Architecture”, Oxford (2007)
3) ISRD group, “Computer Organization”, ace series, TMH (2007)
4) William Stallings, “Computer Organization and Architecture - Designing for Performance”, Pearson Education (2005)
5) P.Chakraborty, “Computer Architecture and Organization”, Jaico Books (2008).

(105MC24)

M.C.A. DEGREE EXAMINATION, MODEL QUESTION PAPER

MCA-FIRST SEMESTER

COMPUTER ORGANIZATION

Time: Three hours **Maximum: 70 marks**

SECTION-A

Answer Question No.1 Compulsory: **7 x 2 = 14 M**

1 a) Define Universal Logic gate.
  b) Write about Tristate Buffer.
  c) State about Interrupt Cycle.
  d) List the RISC Characteristics.
  e) Explain about Associative Memory.
  f) Perform (67) – (42) in binary using $2^{'s}$ complement method.
  g) What is Instruction Format.

SECTION-B

Answer ONE Question from each unit: **4 x 14 = 56 M**

<u>UNIT-I</u>

2 a) Simplify the following Boolean functions using K-maps and draw the relevant logic diagram. $f(abcd) = \Sigma m\,(0,3,7,8,9,11,12,13) + \Sigma d\,(1,4,14,15)$
  b) Explain the operation of Full– Adder circuit.

OR

3 a) Design a 4-bit Synchronous counter using T-Flip Flops.
  b) Explain the Operation of Bidirectional Shift Register.

<u>UNIT-II</u>

4 a) Explain about Instruction cycle in detail.
  b) Explain various Memory reference instructions.

OR

5 a) Explain any one stage of arithmetic logic shift unit.
  b) What is various logic micro operations and their implementation?

<u>UNIT-III</u>

6 a) Explain different addressing modes with an example.
  b) What is an interrupt? Explain various types of interrupts.

OR

7 What is Control memory? Explain address sequencing with suitable diagram.

<u>UNIT-IV</u>

8 a) Explain Booth multiplication Algorithm with an example.
  b) Explain memory mapped I/O and Isolated I/O.

OR

9 a) What is Locality of reference? Explain various organizations of Cache memory.
  b) What is an I/O Interface? Explain DMA data transfer in detail.

CONTENTS

LESSON- 1
DIGITAL LOGIC CIRCUITS

OBJECTIVES:

After going through this lesson, you will be able to

- ❖ Understanding Digital Logic Fundamentals.
- ❖ Mastering Boolean Algebra and Logic Simplification.
- ❖ Analyzing Combinational Circuits.
- ❖ Designing and Implementing Sequential Logic.

STRUCTURE OF THE LESSION:

1.1 DIGITAL COMPUTERS:

Digital computers are digital systems that perform computational tasks using variables that take a limited number of discrete values. These values are processed internally by components that maintain a limited number of discrete states. The decimal digits 0, 1, 2, ..., 9, for example, provide 10 discrete values. The term digital computer emerged from the use of decimal digits for numerical computations. In practice, digital computers function more reliably with only two states. Digital components constrained to take discrete values are binary, as human logic tends to be binary.

Digital computers use the binary number system, consisting of 0 and 1 digits, and represent information in groups of **bits**. These bits can be used to represent other discrete symbols like decimal digits or alphabet letters. By judicious use of binary arrangements and coding techniques, complete sets of instructions for computations are developed.

A computer system is divided into hardware and software, with hardware comprising electronic components and electromechanical devices, and software containing instructions and data for data-processing tasks.

A program is a sequence of instructions for a computer, manipulated by data, and constitutes the data base. A computer system consists of hardware and system software, which is a collection of programs designed to make the computer more effective. The operating system, or system software, is different from application programs, which are written by the user for specific problems. A customer purchasing a computer system needs any available software for effective operation, as system software compensates for differences between user needs and hardware capabilities.

1.1.1 Computer Hardware

The hardware of the computer is usually divided into three major parts, as shown in Fig. 1.1.

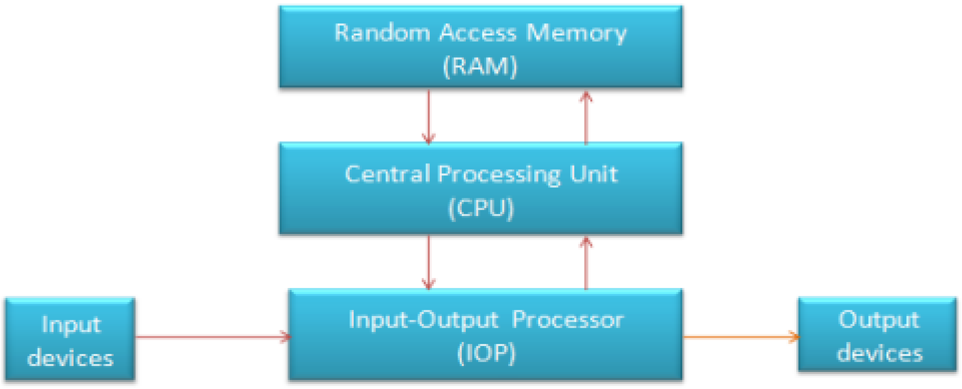

Random Access Memory (RAM)
Central Processing Unit (CPU)
Input devices
Input-Output Processor (IOP)
Output devices

Figure 1.1 Block diagram of a digital computer.

The central processing unit (CPU) contains an arithmetic and logic unit for manipulating data, a number of registers for storing data, and control circuits for fetching and executing instructions. The memory of a computer contains storage for instructions and data. It is called a random access memory (RAM) because the CPU can access any location in memory at random and retrieve the binary information within a fixed interval of time. The input and output processor (lOP) contains electronic circuits for communicating and controlling the transfer of information between the computer and the outside world. The input and output devices connected to the computer include keyboards, printers, terminals, magnetic disk drives, and other communication devices.

1.1.2 Computer Organization, Design and Architecture

- **Computer Organization:**

  Computer organization deals with the way a computer system is structured internally. It includes topics such as data representation, instruction sets, addressing modes, memory organization, and hardware components like processors, memory units, input/output devices, etc and its goal is to understand how the hardware components work together to execute instructions and process data efficiently.

- **Computer Design:**

  Computer design (or digital design) involves the process of designing the structure and organization of digital circuits to achieve a specific functionality or performance goal. This includes designing components like arithmetic logic units (ALUs), registers, multiplexers, etc.

It goes beyond just the architecture and includes detailed circuit design, logic design, and the use of hardware description languages (HDLs) like Verilog or VHDL to implement digital systems.

❖ **Computer Architecture:**

Computer architecture refers to the conceptual design and fundamental operational structure of a computer system. It defines the instruction set architecture (ISA), addressing modes, memory organization, and how various hardware components interact to process information.

Architectural decisions impact performance, power consumption, scalability, and overall system design. It is concerned with the interface between hardware and software, specifying how software instructions are executed at the hardware level.

1.2 LOGIC GATES:

Binary information is represented in digital computers by physical quantities called signals. Electrical signals such as voltages exist throughout the computer in either one of two recognizable states. The two states represent a binary variable that can be equal to 1 or 0. For example, a particular digital computer may employ a signal of 3 volts to represent binary 1 and 0.5 volt to represent binary 0. The input terminals of digital circuits accept binary signals of 3 and 0.5 volts and the circuits respond at the output terminals with signals of 3 and0.5 volts to represent binary input and output corresponding to 1 and 0, respectively.

Binary logic deals with binary variables and with operations that assume a logical meaning. It is used to describe, in algebraic or tabular form, the manipulation and processing of binary information. The manipulation of binary information is done by logic circuits called gates. Gates are blocks of hardware that produce signals of binary 1 or 0 when input logic requirements are satisfied. A variety of logic gates are commonly used in digital computer systems. Each gate has a distinct graphic symbol and its operation can be described by means of an algebraic expression. The input-output relationship of the binary variables for each gate can be represented in tabular form by a truth table.

The names, graphic symbols, algebraic functions, and truth tables of eight logic gates are listed in Figure 1.2. Each gate has one or two binary input variables designated by A and Band one binary output variable designated by x.

The **AND** gate is named so because, if 0 is false and 1 is true, the gate acts in the same way as the logical "and" operator. The following illustration and table show the circuit symbol and logic combinations for an AND gate. (In the symbol, the input terminals are on the left, and the output terminal is on the right.) The output is "true" when both inputs are "true." Otherwise, the output is "false." In other words, the output is 1 only when both inputs are 1.

The **OR** gate gets its name from behaving like the logical inclusive "or." The output is true if one or both of the inputs are true. If both inputs are false, then the output is false. In other words, for the output to be 1, at least one input must be 1.

A logical inverter, sometimes called a **NOT** gate to differentiate it from other types of electronic inverter devices, has only one input. A NOT gate reverses the logic state. If the input is 1, then the output is 0. If the input is 0, then the output is 1.

The **XOR** (exclusive-OR) gate acts in the same way as the logical "either/or." The output is true if either, but not both, of the inputs are true. The output is false if both inputs are "false" or if both inputs are true. Similarly, the output is 1 if the inputs are different but 0 if the inputs are the same.

The **NAND** (Negated AND) gate operates as an AND gate followed by a NOT gate. It acts in the manner of the logical operation "and" followed by negation. The output is false if both inputs are true. Otherwise, the output is true. Another way to visualize it is that a NAND gate inverts the output of an AND gate. The NAND gate symbol is an AND gate with the circle of a NOT gate at the output.

The **NOR** (NOT OR) gate is a combination OR gate followed by an inverter. Its output is true if both inputs are false. Otherwise, the output is false.

The **XNOR** (exclusive-NOR) gate is a combination of an XOR gate followed by an inverter. Its output is true if the inputs are the same and false if the inputs are different.

1.3 BOOLEAN ALGEBRA:

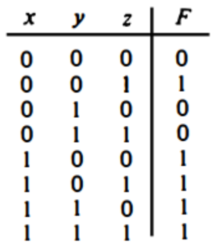
Boolean algebra is an algebra that deals with binary variables and logic operations. The variables are designated by letters such as A, B, x, and y. The three basic logic operations are AND, OR, and complement.

A Boolean function can be expressed algebraically with binary variables, the logic operation symbols, parentheses, and equal sign. For a given value of the variables, the Boolean function can be either 1 or 0. Consider, for example, the Boolean function

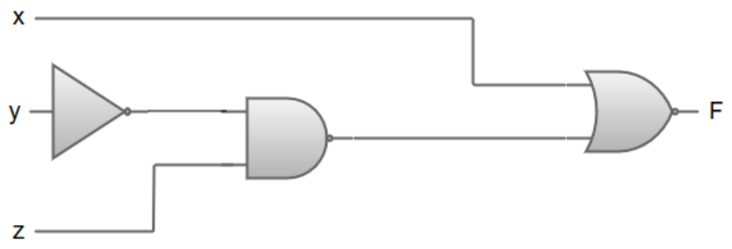
$$F = x + y'z$$

The function F is equal to 1 if x is 1 or if both y' and z are equal to I; F is equal to 0 otherwise. But saying that y' = 1 is equivalent to saying that y = 0 since y' is the complement of y. Therefore, we may say that F is equal to 1 if x = 1 or if yz = 01. The relationship between a function and its binary variables can be represented in a truth table. To represent a function in a truth table we need a list of the 2' combinations of then binary variables.

As shown in Figure l-3(a), there are eight possible distinct combinations for assigning bits to the three variables x, y, and z. The function F is equal to 1 for those combinations where x = 1 or yz = 01; it is equal to 0 for all other combinations. A Boolean function can be transformed from an algebraic expression into a logic diagram composed of AND, OR, and inverter gates.



x	*y*	*z*	*F*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1



(a) Truth Table **(b) Logic diagram**

Figure 1.3 logic diagram F=x+y'z

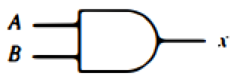
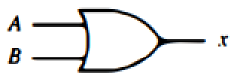
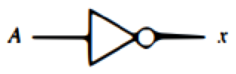
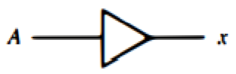
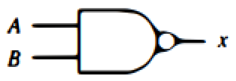
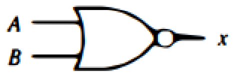
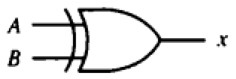
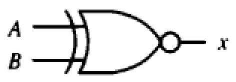
Name	*Graphic symbol*	*Algebraic function*	*Truth table*
AND		$x = A \cdot B$ or $x = AB$	A B | x 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR		$x = A + B$	A B | x 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
Inverter		$x = A'$	A | x 0 | 1 1 | 0
Buffer		$x = A$	A | x 0 | 0 1 | 1
NAND		$x = (AB)'$	A B | x 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR		$x = (A + B)'$	A B | x 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0
Exclusive-OR (XOR)		$x = A \oplus B$ or $x = A'B + AB'$	A B | x 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0
Exclusive-NOR or equivalence		$x = (A \oplus B)'$ or $x = A'B' + AB$	A B | x 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 1

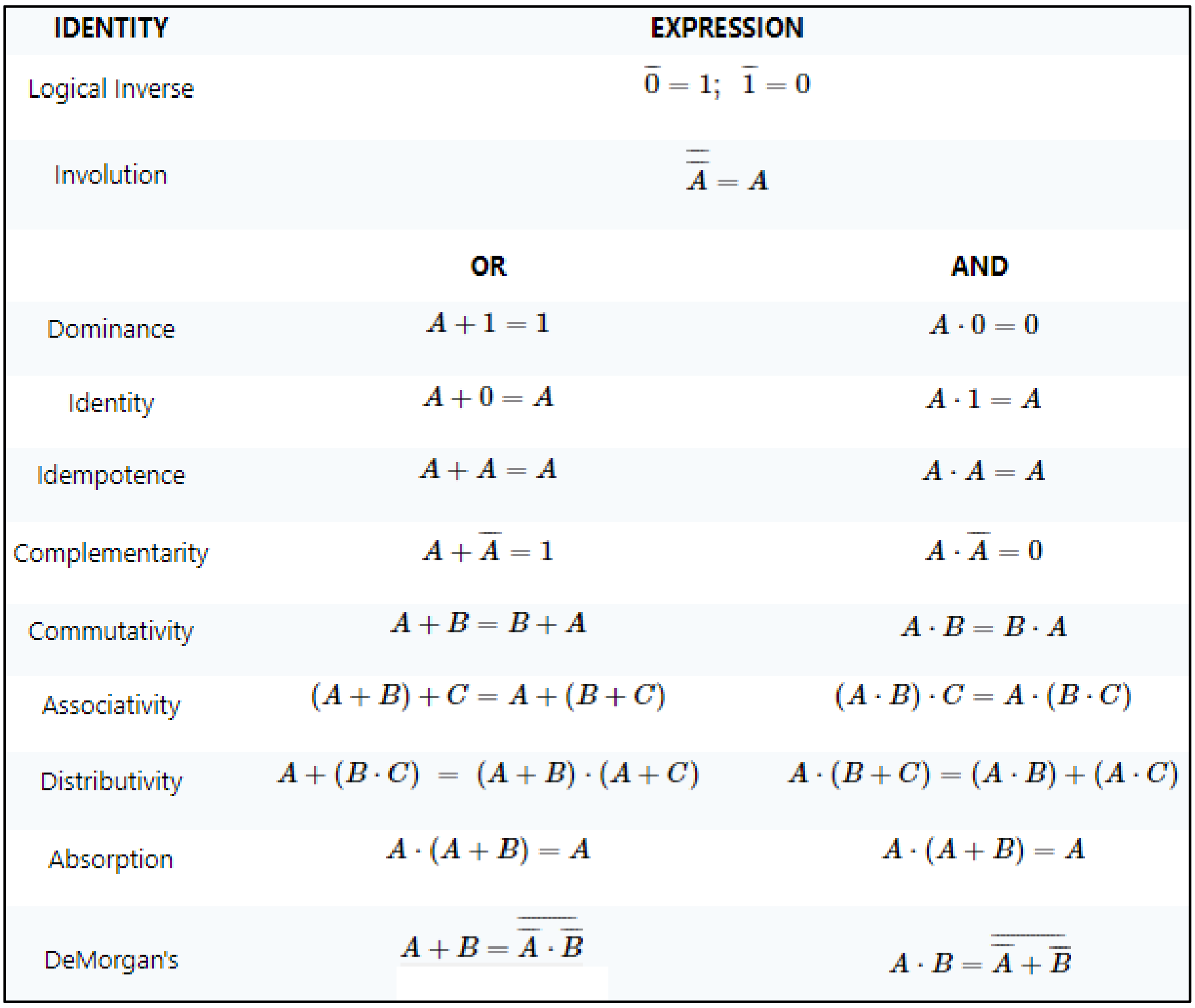
Figure 1.2 Logic Gates

The logic diagram for F is shown in Fig. l-3(b). There is an inverter for input y to generate its complement y'. There is an AND gate for the term y'z, and an OR gate is used to combine the two terms. In a logic diagram, the variables of the function are taken to be the inputs of the circuit, and the variable symbol of the function is taken as the output of the circuit

Boolean algebra is a mathematical system used in digital logic and computer science, where variables can only have two possible values: true (1) or false (0).

Boolean expressions can be manipulated algebraically to form simpler expressions that implement the same logic. This may allow a simpler circuit to be produced with fewer logic gates. This will reduce the cost of the circuit, the amount of heat generated, and the processing time.

Here are some basic identities and rules of Boolean algebra:



IDENTITY	EXPRESSION	
Logical Inverse	$\bar{0} = 1;\ \bar{1} = 0$	
Involution	$\bar{\bar{A}} = A$	
	OR	**AND**
Dominance	$A + 1 = 1$	$A \cdot 0 = 0$
Identity	$A + 0 = A$	$A \cdot 1 = A$
Idempotence	$A + A = A$	$A \cdot A = A$
Complementarity	$A + \bar{A} = 1$	$A \cdot \bar{A} = 0$
Commutativity	$A + B = B + A$	$A \cdot B = B \cdot A$
Associativity	$(A + B) + C = A + (B + C)$	$(A \cdot B) \cdot C = A \cdot (B \cdot C)$
Distributivity	$A + (B \cdot C) = (A + B) \cdot (A + C)$	$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$
Absorption	$A \cdot (A + B) = A$	$A \cdot (A + B) = A$
DeMorgan's	$A + B = \overline{\bar{A} \cdot \bar{B}}$	$A \cdot B = \overline{\bar{A} + \bar{B}}$

Figure 1.4 Basic identities of Boolean Algebra

De Morgan's Theorem:

A theorem put out by the British mathematician Augustus De Morgan in the nineteenth century, De Morgan's theorem consists of two rules. The AND, NOT, and OR rules are utilised by the two statutes. One Boolean expression can be transformed from one form to another by utilising the qualities of these functions. De Morgan's theorem holds true for two or more variables.

First Theorem: the inversion of the product is the same as the sum of the inversions

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

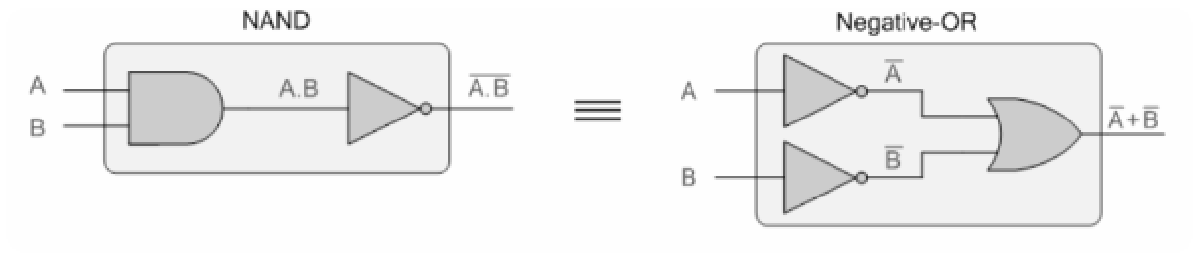


Figure 1.5 Demorgans First theorem

Second Theorem: the inversion of the sum is the same as the product of the inversions.

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

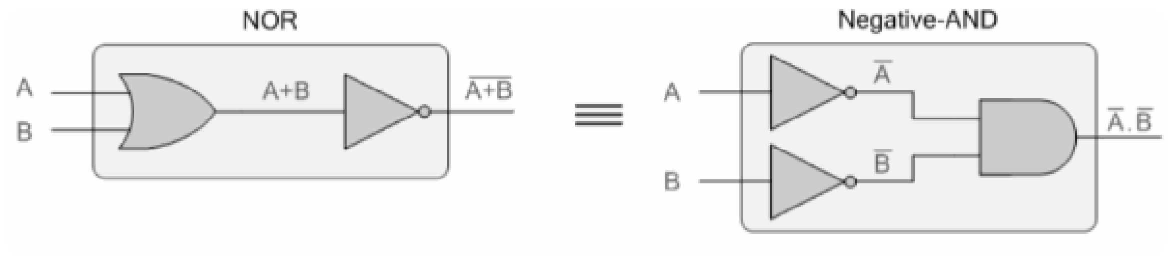


Figure 1.6 Demorgan's Second theorem

Complement of a Function F: The complement of a function F when expressed in a truth table is obtained by interchanging l's and D's in the values of F in the truth table. When the function is expressed in algebraic form, the complement of the function can be derived by means of DeMorgan's theorem. The general form of DeMorgan's theorem can be expressed as follows:

$$(x_1 + x_2 + x_3 + \ldots. + x_n)' = x_1' x_2' x_3' \ldots x_n'$$

$$(x_1 x_2 x_3 \ldots.. x_n)' = x_1' + x_2' + x +_3' \ldots. + x_n'$$

From the general DeMorgan's theorem we can derive a simple procedure for obtaining the complement of an algebraic expression. This is done by changing all OR operations to AND operations and all AND operations to OR operations and then complementing each individual letter variable. As an example, consider the following expression and its complement:

F = AB + C'D' + B'D

F' = (A' + B')(C + D)(B + D')

1.4 MAP SIMPLIFICATION:

The Karnaugh map (K-map) approach is a graphical tool used to simplify Boolean equations. K-maps, often known as 2D truth tables, are an alternative way of displaying the values found in a one-dimensional truth table. K-maps involve the process of placing the output variable values in cells within a rectangular or square grid, following a specific pattern..

The number of cells in the K-map is determined by the number of input variables and is mathematically expressed as two raised to the power of the number of input variables, i.e., 2n, where the number of input variables is n.

Thus, to simplify a logical expression with two inputs, we require a K-map with 4 (= 22) cells. A four-input logical expression would lead to a 16 (= 24) celled-K-map, and so on.

Gray Code:

Additionally, a method of encoding known as Grey code is utilised in order to assign a specific place value to each individual cell that is contained within a K-map.

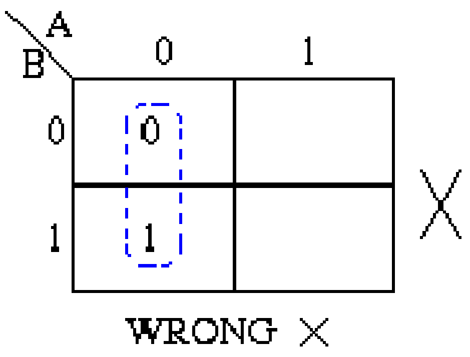
One of the most notable characteristics of this code is that the values of the adjacent codes differ from one another by only one bit. In other words, if the code-word that is being used is 01, then the code-words that come before and after it can be either 11 or 00, in any order, but they cannot be 10 under any circumstances.

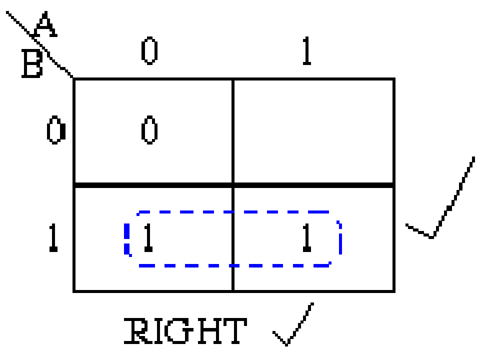
In K-maps, the rows and columns of the table are labelled with grey code, which in turn reflects the values of the input variables that correlate to those rows and columns with grey code. Therefore, it is possible to address each K-map cell by utilising a Grey Code-Word that is unique to that cell.

K-Map Rules:

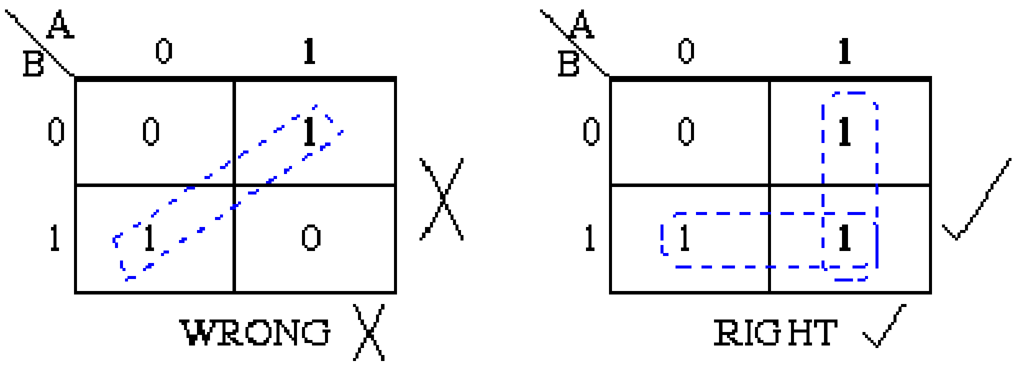
The Karnaugh map uses the following rules for the simplification of expressions by *grouping* together adjacent cells containing *ones*

Groups may not include any cell containing a zero

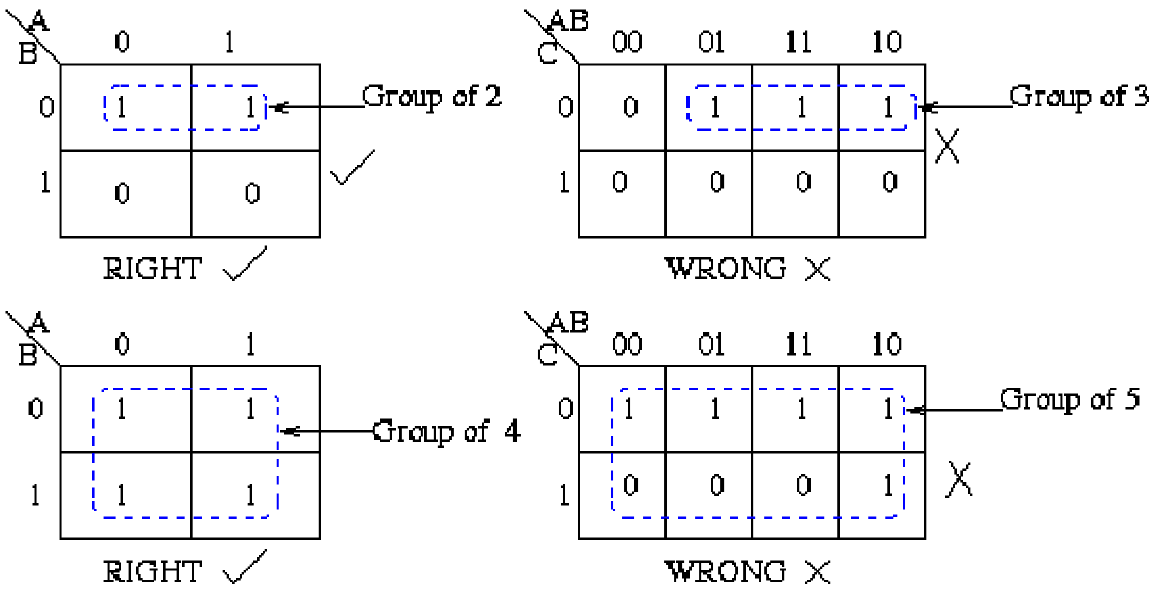




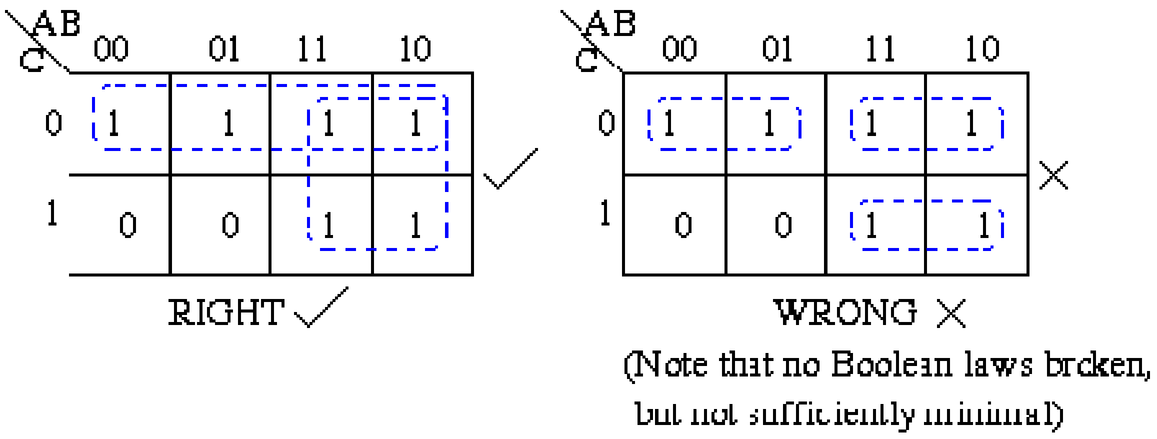
Groups may be horizontal or vertical, but not diagonal.



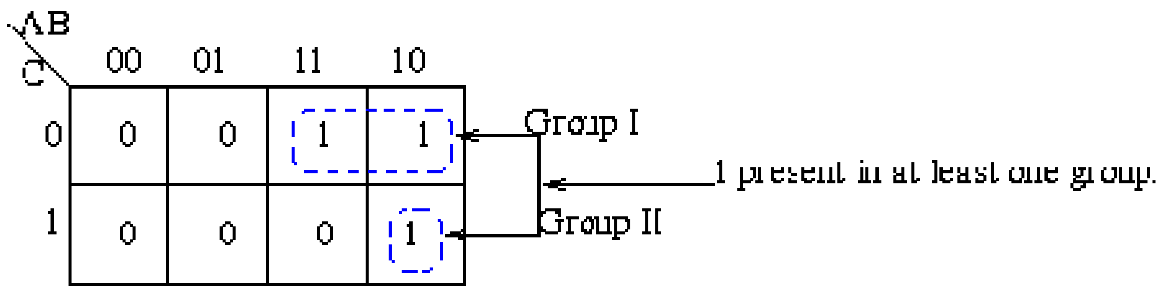
Groups must contain 1, 2, 4, 8, or in general 2n cells.That is if n = 1, a group will contain two 1's since 21 = 2.If n = 2, a group will contain four 1's since 22 = 4.



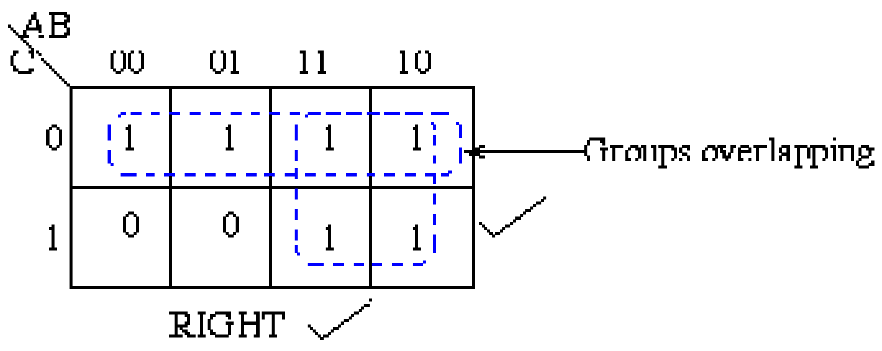
Each group should be as large as possible.

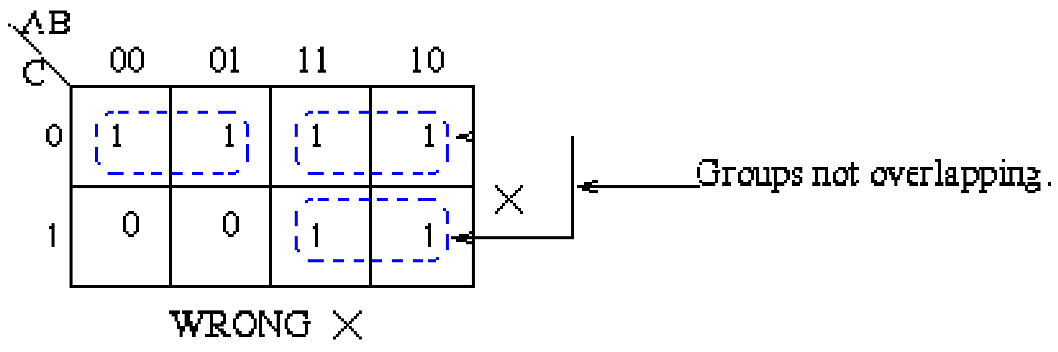

RIGHT
WRONG
(Note that no Boolean laws broken,
but not sufficiently minimal)

Each cell containing a one must be in at least one group.

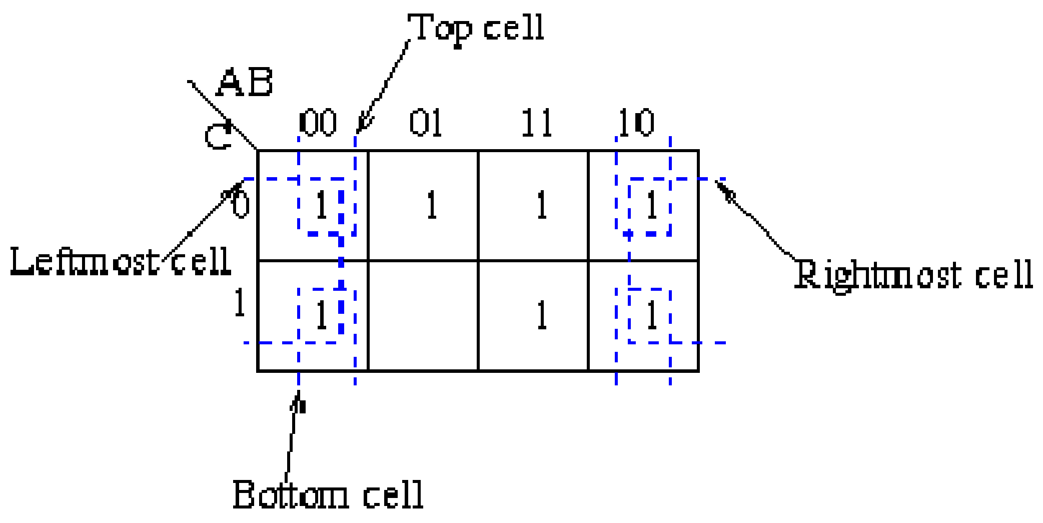


Groups may overlap.





Groups may wrap around the table. The leftmost cell in a row may be grouped with the rightmost cell and the top cell in a column may be grouped with the bottom cell.



There should be as few groups as possible, as long as this does not contradict any of the previous rules.

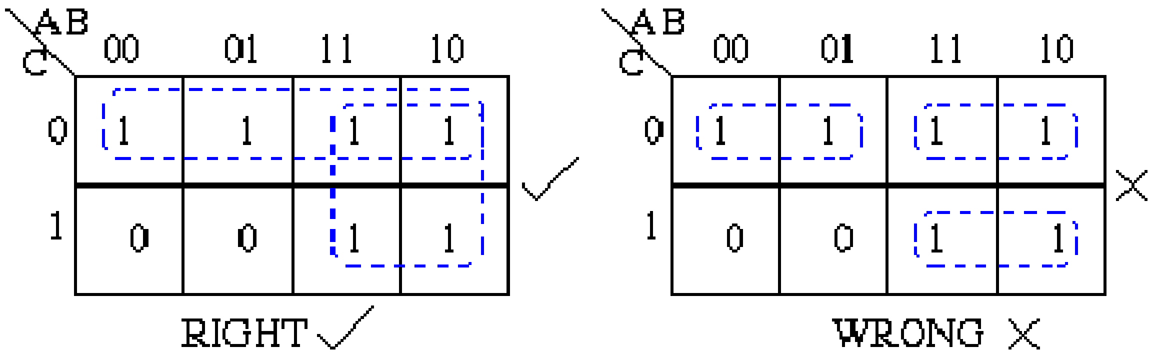


Figure 1.7 K-map rules

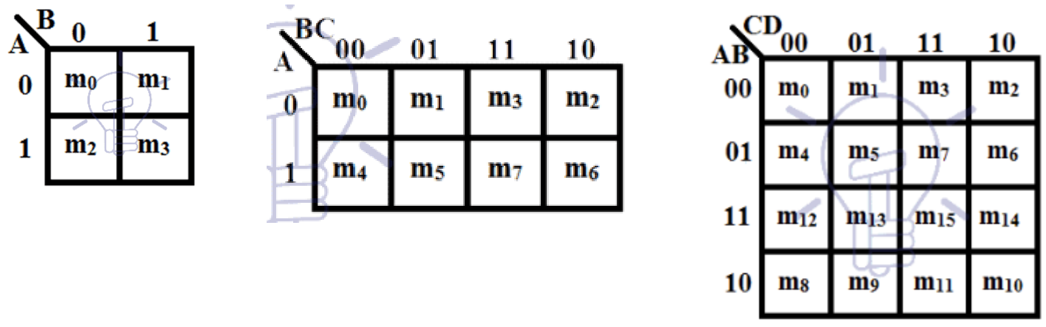
K-map can take two forms:

Sum of product (SOP)

Product of Sum (POS)

Steps to Solve Expression using K-map

- Select the K-map according to the number of variables.
- Identify minterms or maxterms as given in the problem.
- For SOP put 1's in blocks of K-map respective to the minterms (0's elsewhere).
- For POS put 0's in blocks of K-map respective to the max terms (1's elsewhere).
- Make rectangular groups containing total terms in power of two like 2,4,8 ..(except 1) and try to cover as many elements as you can in one group.
- From the groups made in step 5 find the product terms and sum them up for SOP form.

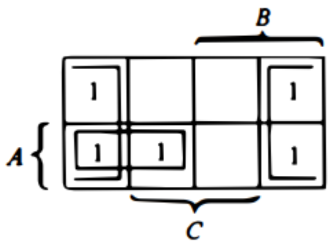


2-Variablle K-Map **3-Variable K-map** **4-variable K-map**

Figure 1.8 Types of K-maps

Examples:

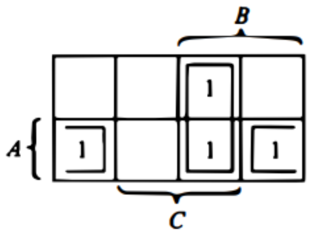
F(A, B, C) = $\sum$(0, 2, 4, 5, 6)



The simplified function is

F = C'+AB'

F(A. B. C) = $\sum$(3,4.6, 7).



The Simplified Function is

F = BC + AC'

1.5 COMBINATIONAL CIRCUITS:

Combinational circuit is a circuit in which we combine the different gates in the circuit. A logic gate is a basic building block of any electronic circuit. The output of the combinational circuit depends on the values at the input at any given time. The circuits do not make use of any memory or storage device

Examples are: encoder, decoder, multiplexer and demultiplexer.

Combinational logic is used in computer circuits to perform Boolean algebra on input signals and on stored data. Other circuits used in computers, such as half adders, full adders, half subtractors, full subtractors, multiplexers, demultiplexers, encoders and decoders are also made by using combinational logic.

Some of the characteristics of combinational circuits are following:

- The output of a combinational circuit at any instant of time, depends only on the levels present at input terminals.
- It does not use any memory. The previous state of input does not have any effect on the present state of the circuit.
- It can have an n number of inputs and m number of outputs.

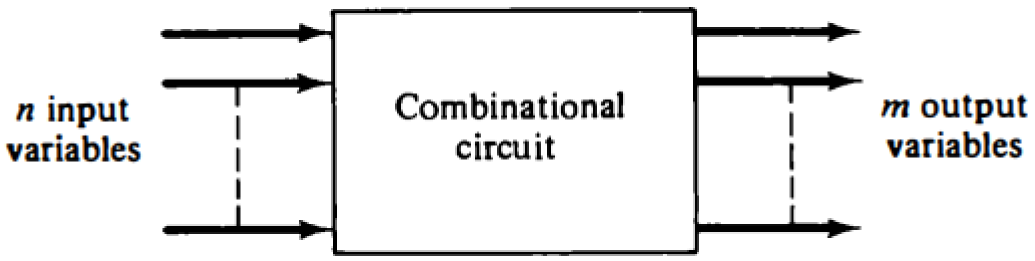


Figure 1.9 Block diagram of Combinational Circuit

An adder is a circuit that can be integrated with many other circuits for a wide range of applications. It is a kind of calculator used to add two binary numbers. There are two kinds of adders;

1. Half adder

2. Full adder

1.5.1 Half Adder

With the help of half adder, we can design circuits that are capable of performing simple addition with the help of logic gates.

Example of the addition of single bits.

0+0 = 0

0+1 = 1

1+0 = 1

1+1 = 10

These are the possible single-bit combinations. But the result for 1+1 is 10. Though this problem can be solved with the help of an XOR Gate, the sum result must be re-written as a 2-bit output.

Thus the above equations can be written as:

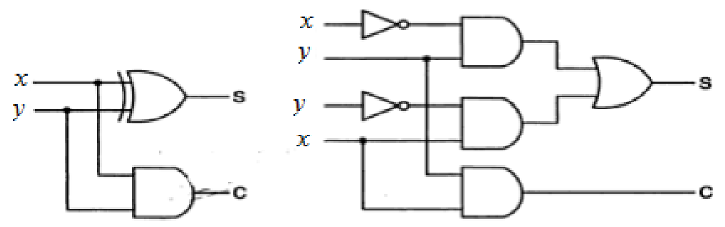
0+0 = 00

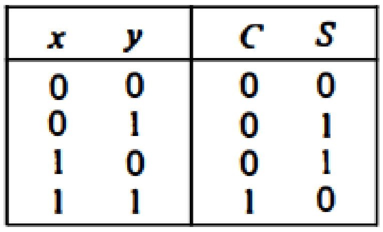
0+1 = 01

1+0 = 01

1+1 = 10

A Half Adder is a combinational circuit with two binary inputs (augends and addend bits and two binary outputs (sum and carry bits.) It adds the two inputs (x and y) and produces the sum (S) and the carry (C) bits. It is an arithmetic operation of addition of two single bit words.





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

Figure 1.10 Half - adder

The Sum(S) bit and the carry (C) bit, according to the rules of binary addition, the sum (S) is the X-OR of x and y (It represents the LSB of the sum). Therefore,

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

The carry (C) is the AND of x and y (it is 0 unless both the inputs are 1).Therefore,

$$C = xy$$

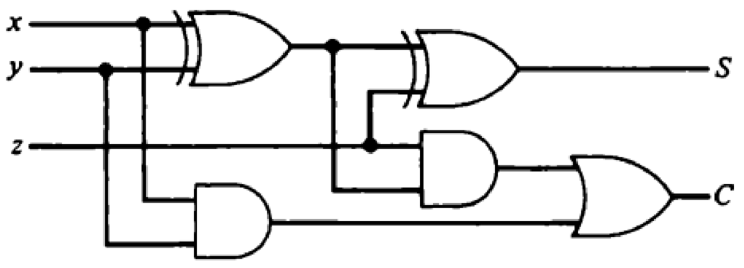
A half-adder can be realized by using one X-OR gate and one AND gate

1.5.2 Full Adder

A Full-adder is a combinational circuit that adds two bits and a carry and outputs a sum bit and a carry bit. To add two binary numbers, each having two or more bits, the LSBs can be added by using a half-adder. The carry resulted from the addition of the LSBs is carried over to the next significant column and added to the two bits in that column. So, in the second and higher columns, the two data bits of that column and the carry bit generated from the addition in the previous column need to be added.

The full-adder adds the bits x and y and the carry from the previous column called the carry-in C_{in} and outputs the sum bit S and the carry bit called the carry-out C_{out} . The variable S gives the value of the least significant bit of the sum. The variable C_{out} gives the output carry. The eight rows under the input variables designate all possible combinations of 1s and 0s that these variables may have. The 1s and 0s for the output variables are determined from the arithmetic sum of the input bits. When all the bits are 0s, the output is 0. The S output is equal to 1 when only 1 input is equal to 1 or when all the inputs are equal to 1. The C_{out} has a carry of 1 if two or three inputs are equal to 1.

From the truth table, a circuit that will produce the correct sum and carry bits in response to every possible combination of x, y and z in is described by



Logic diagram

Inputs			Outputs	
x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Truth Table

Figure 1.11 Half -adder

1.6 SUMMARY:

The chapter on digital computers and circuits covers essential concepts in the field of digital electronics. It begins with an introduction to digital computers, emphasizing their reliance on binary representation and the role of logic gates in processing information. Boolean Algebra is introduced as the mathematical foundation for expressing and simplifying logical operations, crucial for designing efficient circuits. Map simplification techniques like Karnaugh maps are explored to optimize Boolean expressions. Combinational circuits, which derive outputs solely from present inputs, are detailed alongside examples such as adders and multiplexers. This comprehensive overview equips readers with a foundational understanding of how digital systems are designed, from basic logic operations to sophisticated circuit architectures.

1.7 TECHNICAL TERMS:

Digital Computers, Logic Gates, Boolean algebra, Map Simplification, Combinational Circuits, and Sequential Circuits

1.8 SELF ASSESSMENT QUESTIONS:

Essay questions:

1) Explain the significance of Boolean algebra in digital circuit design. Provide examples to illustrate its application.
2) Compare and contrast combinational and sequential circuits. Discuss their respective uses and provide examples of each.
3) Simplify the following Boolean functions using three-variable maps.

a. $F(x,y,z) = \sum (0, 1,5, 7)$ b. $F(x,y,z) = \sum (1,2,3,6,7)$ c. $F(x,y,z) = \sum (3, 5, 6, 7)$

Short Questions:

1) Define a logic gate. Provide examples of common logic gates and their Boolean expressions.
2) Simplify the Boolean expressions using boolean algebra

   a. AB + A(CD + CD') b. (BC' + A'D) (AB' + CD')

3) Simplify the following expressions in (1) sum-of-products form and (2) product-of-sums form.

   a. x'z' + y'z' + y z' + xy   b. AC' + B'D + A'CD + ABCD

4) Simplify the Boolean function F together with the don't-care conditions d in

   (1) sum-of-products form and (2) product-of-sums form.

   $F(w,x, y, z) =\sum(0, 1,2, 3, 7, 8, 10)$ $d(w,x,y ,z) =\sum (5 , 6, 11, 15)$

1.9 FURTHER READINGS:

1) M.Morris Mano, "Computer System Architecture", 3rd Edition, Pearson Education (2008).
2) V. Rajaraman, T. Radha Krishnan, "Computer Organization and Architecture", PHI.
3) BehroozParhami, "Computer Architecture", Oxford (2007).
4) ISRD group, "Computer Organization", ace series, TMH (2007).
5) William Stallings, "Computer Organization and Architecture–Designing for Performance", Pearson Education (2005).

Dr. U. Surya Kameswari

LESSON-2

NUMBER SYSTEM

OBJECTIVES:

After going through this lesson, you will be able to

- Understand the fundamental concepts of different number systems,
- Convert numbers between these different number systems accurately.
- Explain the significance of each number system in the context of digital electronics and computing.
- Analyze the advantages and disadvantages of using different number systems in various applications.

STRUCTURE OF THE LESSION:

2.1 INTRODUCTION:

A number system of base, or radix, r is a system that uses distinct symbols for r digits. Numbers are represented by a string of digit symbols. To determine the quantity that the number represents, it is necessary to multiply each digit by an integer power of r and then form the sum of all weighted digits.

The number system is fundamental in digital electronics as it provides a way to represent and manipulate data. Different number systems are used to encode information, perform arithmetic operations, and communicate between devices.

The primary number systems in digital electronics include:

2.2 BINARY NUMBER SYSTEM:

The binary number system, also known as the base-2 system, is a fundamental numerical system used extensively in digital electronics and computing. It consists of only two digits: 0 and 1. Each digit in a binary number represents a power of 2, with the rightmost digit representing 2^0, the next 2^1, 2^2, and so forth.

For example, the binary number 1011 translates to decimal as $1\times2^3+0\times2^2+1\times2^1+1\times2+$ $=8+0+2+1=11$. This simplicity allows computers and digital devices to efficiently process and store data using two distinct states, which correspond to the on and off states of electronic circuits.

The binary system is essential in the design and operation of digital systems, as it aligns perfectly with the architecture of computers. Each bit (binary digit) can represent a state either low (0) or high (1)enabling the implementation of logical operations and data manipulation through combinations of bits. Moreover, larger binary numbers can be grouped into bytes (typically 8 bits), facilitating more complex data representations. The binary system's structure underpins everything from basic arithmetic operations in computer algorithms to complex data structures and programming languages, making it a cornerstone of modern computing technology.

2.3 DECIMAL NUMBER SYSTEM:

The decimal number system, also known as the base-10 system, is the standard system for denoting integer and non-integer numbers in everyday life. It employs ten digits, from 0 to 9, to represent values. Each digit's position within a number signifies its weight, which is determined by powers of 10. For example, in the number 345, the digit '3' is in the hundreds place, '4' is in the tens place, and '5' is in the ones place. This positional notation allows for the representation of large numbers using a compact form, making it intuitive for human use.

The fundamental principle of the decimal system is that each place value represents a power of 10. For instance, in the number 2,578, the digit '2' represents 2×10^3 (or 2000), '5' represents 5×10^2 (or 500), '7' represents 7×10^1 (or 70), and '8' represents 8×10^0 (or 8). This structure allows for the easy calculation of a number’s value by summing the contributions of each digit based on its position, making arithmetic operations straightforward.

2.4 HEXADECIMAL NUMBER SYSTEM:

The hexadecimal number system, also known as base-16, is a numerical system that uses sixteen distinct symbols to represent values. These symbols include the digits 0 to 9, which represent values zero to nine, and the letters A to F, which represent values ten to fifteen. This compact representation allows for a more efficient encoding of large binary numbers, as each hexadecimal digit corresponds to four binary digits (bits). For example, the binary number 1010 translates to the hexadecimal digit A, while the binary number 1111 corresponds to F. This relationship makes it easy to convert between binary and hexadecimal, as each group of four bits can be directly represented by a single hexadecimal digit.

Hexadecimal is widely used in computing and digital electronics for several reasons. It simplifies the representation of binary data, making it easier for programmers and engineers to read and interpret large numbers. For instance, a binary number like 11111111 can be succinctly expressed as FF in hexadecimal, which is much easier to work with. Hexadecimal is commonly used in programming languages, memory addressing, and color codes in web design (e.g., HTML color values), where it enhances clarity and efficiency. Its ability to represent large values in a compact form makes the hexadecimal system an essential tool in various aspects of computer science and digital technology.

2.5 OCTAL NUMBER SYSTEM:

The octal number system, also known as base-8, utilizes eight distinct digits: 0 through 7. Each digit in an octal number represents a power of 8, which allows for a more compact representation of binary numbers compared to the decimal system. For example, the octal number 157 can be interpreted as 1×82+5×81+7×80=64+40+7=1111 \times 8^2 + 5 \times 8^1 + 7 \times 8^0 = 64 + 40 + 7 = 1111×82+5×81+7×80=64+40+7=111 in decimal. The octal system is particularly useful in computing, as each octal digit corresponds to three binary digits (bits), making conversions between the two systems straightforward.

Historically, the octal system was more prominent in early computing systems and programming languages, especially when dealing with file permissions in Unix-based systems, where each octal digit represents a set of read, write, and execute permissions. However, its use has diminished in favor of the hexadecimal system, which provides a more compact representation for binary data. Despite this decline, octal remains relevant in specific applications and provides a clear method for representing binary values in a more manageable format, especially in contexts where space or simplicity is a concern.

2.6 NUMBER CONVERSION:

The following number system conversion methods are mostly used in digital computers

- Decimal to Binary Number System
- Octal to Binary Number System
- Binary to Octal Number System
- Binary to Hexadecimal Number System
- Hexadecimal to Binary Number System

Number system conversions deal with the process of altering the numbers' base. For instance, converting a base-10 decimal value to a base-2 binary number. On the number system, we can also carry out arithmetic operations like addition, subtraction, and multiplication. For each base number, the general representation of number system base conversion is as follows:

$(\text{Number})_b = d_{n-1}\ d_{n-2}\ .\ .\ .\ d_1\ d_0\ .\ d_{-1}\ d_{-2}\ .\ .\ .\ d_{-m}$

In the above expression, $d_{n-1}\ d_{n-2}\ .\ .\ .\ d_1\ d_0$ represents the value of integer part and $d_{-1}\ d_{-2}\ .\ .\ .d_{-m}$ represents the fractional part.

Also, d_{n-1} is the Most significant bit (MSB) and d_{-m} is the Least significant bit (LSB).

2.6.1 Decimal to other Bases:

2.6.1.1 Decimal to Binary Number:

If we need to convert a decimal number to binary, simply divide it by two.

Example 1: Convert $(35)_{10}$ to binary number.

Operation	Output	Remainder
35 ÷ 2	17	1(LSB)
17 ÷ 2	8	1
8 ÷ 2	4	0
4 ÷ 2	2	0
2 ÷ 2	1	0
1 ÷ 2	1	1(MSB)

Therefore, from the above table, we can write,

$(35)_{10} = (100011)_2$

2.6.1.2 Decimal to Octal Number:

To convert a decimal to an octal number, divide the given original value by 8 such that base 10 becomes base 8.

Example 2: Convert 214_{10} to octal number.

Operation	Output	Remainder
214÷8	26	6(LSB)
26÷8	3	2
3÷8	0	3(MSB)

Therefore, the equivalent octal number = 326_8

2.6.1.3 Decimal to Hexadecimal:

Again in decimal to hex conversion, we have to divide the given decimal number by 16.

Example 3: Convert 945_{10} to hex.

Operation	Output	Remainder
945÷16	59	1 (LSB)
59÷16	3	11
3÷16	0	3 (MSB)

Therefore, the equivalent hexadecimal number is $3B1_{16}$

2.6.2 Binary to other Bases

2.6.2.1 Binary to Decimal:

In this conversion, binary number to a decimal number, we use multiplication method, in such a way that, if a number with base n has to be converted into a number with base 10, then each digit of the given number is multiplied from MSB to LSB with reducing the power of the base. Let us understand this conversion with the help of an example.

Example 1: Convert $(1101)_2$ into a decimal number.

Given a binary number $(1101)_2$.

Now, multiplying each digit from MSB to LSB with reducing the power of the base number 2.

$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

$= 8 + 4 + 0 + 1$

$= 13$

Therefore, $(1101)_2 = (13)_{10}$

2.6.2.2 Binary to Octal

There are several different forms of numbers in the number system, including binary, octal, decimal, and hexadecimal. To convert binary numbers into octal numbers, follow the instructions below.

- Take the provided binary number.
- Multiply each digit by 2^{n-1}, where n is the digit's location from the decimal.
- The resultant is the equivalent decimal number to the given binary integer. \
- Divide the decimal number by eight.
- Note the remainder.
- Continue the preceding two procedures with the quotient until it reaches zero.
- Write the remaining in reverse order.
- The resulting is the required octal number for the given binary value.

Example:

Given binary number is 11010110_2

First, we convert given binary to decimal

Base 2 to decimal calculation:

$(11010110)_2 = (1 \times 2^7) + (1 \times 2^6) + (0 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) = (214)_{10}$

Now we will convert this decimal to octal form

Operation	**Output**	**Remainder**
214÷8	26	6(LSB)
26÷8	3	2
3÷8	0	3(MSB)

Therefore, the equivalent octal number = 326_8

2.6.2.3 Binary to Hexadecimal

Given binary number is 11010110_2

First, we convert given binary to decimal

Base 2 to decimal calculation:

$(11010110)_2 = (1 \times 2^7) + (1 \times 2^6) + (0 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) = (214)_{10}$

Now we will convert this decimal to hexadecimal form

Operation	Output	Remainder
214÷16	13	6 (LSB)
13÷16	0	13 (MSB)

Therefore, the equivalent hexadecimal number is $D6_{16}$

2.6.3 Octal to other Bases

2.6.3.1 Octal to Decimal

To convert binary numbers into octal numbers, follow the instructions below.

- Count the number of digits present in the given number. Assume the number of digits to be ‘x’

- Multiply each of the numbers with 8x-1, and when the digit is in the xth position from the right end of the number. If the number has a decimal part, multiply each digit in the decimal part by 8-y when the digit is in the yth position from the decimal point.
- Now add all the terms after multiplication.
- The obtained answer is the decimal number.

Example:

Convert 326_8 to decimal number

Base 8 to decimal calculation:

(326)8 = (3 × 82) + (2 × 81) + (6 × 80) = (214)10

2.6.3.2 Octal to Binary

There are 2 methods to convert an octal number to a binary number:

- Method 1: Octal to Decimal and Then Decimal to Binary
- Method 2: Direct Method of Octal to Binary Conversion

Direct method of octal to binary conversion

There is a straightforward way to convert an octal number to a binary number. Because the octal number system has just eight symbols (i.e., 0, 1, 2, 3, 4, 5, 6, 7), the base (i.e., 8) is identical to 23 = 8. As a result, we may represent each digit of the octal number as a group of three bits in binary.

This method is straightforward and also serves as the inverse of binary to octal conversion. The steps are outlined below:

Step 1: Use the provided octal number as input.

Step 2: Use the octal to binary table to convert each digit of the octal number to its binary equivalent.

Step 3: Generate the equivalent binary number.

Example:

Convert $(4321)_8 = (?)_2$

Binary representation of each digit of 4321 is:

4 – 100

3 – 011

2 – 010

1 – 001

Therefore, $(4321)_8$=$(100011010001)^2$

Octal to Hexadecimal

Two steps are required to change an octal number to hexadecimal.

Step 1: We first convert the octal number into binary.

In order to convert an octal number to binary, we must write each octal digit's 3 bit binary equivalent in the same order.

Step 2: The binary number is changed to hexadecimal in the following step.

The binary number divides into groups of four bits starting at the binary point. When dealing with whole numbers, we move to the left, and when dealing with fractions, we move to the right.

Example:

Covert the octal number $(56)_8$ to a hexadecimal number.

Fist convert $(56)_8$ into binary number

$(56)_8$

$= (101)\ (110)$

$= (101110)_2$

Now convert $(101110)_2$ in hexadecimal

$(101110)_2$

$= (10)\ (1110)$

$= (2)\ (14)$

$= (2E)_{16}$

2.6.4 Hexadecimal to other Bases

2.6.4.1 Hexadecimal to Decimal

The base number 16 is used to convert hexadecimal to decimal. To change a hexadecimal number to a decimal one:

Step 1: Write each number's digit's decimal equivalent in hexadecimal form.

Step 2: Starting with the digit on the right, multiply the numbers from right to left using exponents of 16^0, 16^1, 16^2, ...

Step 3: Next, add all the products. The resultant sum is the number in the decimal system

Example:

Convert $(A4E)_{16}$ into Decimal

Base 16 to decimal calculation:

$(A4E)_{16} = (10 \times 16^2) + (4 \times 16^1) + (14 \times 16^0) = (2638)_{10}$

2.6.4.2 Hexadecimal to Binary

There are only 16 digits from 0 to 7 and A to F in the hexadecimal number system, so we can represent any digit of the hexadecimal number system using only 4 bits as follows below.

Step 1: Divide the given hexadecimal number into individual digits.

Step 2: Assign binary equivalents to each hexadecimal digit. It’s important to keep track of leading zeros when converting hexadecimal to binary to maintain the correct number of digits.

Hexa	0	1	2	3	4	5	6	7
Binary	0000	0001	0010	0011	0100	0101	0110	0111

Hexa	8	9	A	B	C	D	E	F
Binary	1000	1001	1010	1011	1100	1101	1110	1111

Step 3: Replace each hexadecimal digit with its binary equivalent using the conversion table.

Example:

hexadecimal number “2A” to binary:

“2” in hexadecimal is equivalent to “0010” in binary.

“A” in hexadecimal is equivalent to “1010” in binary.

Therefore, “2A” in hexadecimal is equivalent to “00101010” in binary.

2.6.4.3 Hexadecimal to Octal

This method is straightforward and the steps are outlined below:

1. For each given hexadecimal number digit, write the equivalent binary number. If any of the binary equivalents are less than 4 digits, add 0’s to the left side.
2. Combine and make the groups of binary digits from right to left, each containing 3 digits. Add 0’s to the left if there are less than 3 digits in the last group.
3. Find the octal equivalent of each binary group.

Example: convert $1BC_{16}$ to Octal

Given, $1BC_{16}$ is a hexadecimal number.

1 → 0001, B → 1011, C →1100

Now group them from right to left, each having 3 digits.

000, 110, 111, 100

000→0, 110 →6, 111→7, 100→4

Hence, $1BC_{16} = 674_8$

2.7 APPLICATIONS OF NUMBER SYSTEMS:

Number systems are fundamental to digital electronics, underpinning various applications that range from data representation to arithmetic operations.

1. **Data Representation**
   Digital systems use binary (base-2) to represent data because it aligns perfectly with the two states of electronic circuits—on (1) and off (0). Each piece of data, whether a character, number, or instruction, is encoded in binary format for processing and storage. For example, ASCII characters are represented in binary, allowing computers to interpret text.
2. **Memory Addressing**
   Hexadecimal (base-16) is commonly used in computer memory addressing. Since one hexadecimal digit represents four binary digits, it simplifies the representation of large binary addresses. This makes it easier for programmers and engineers to work with memory locations, as hexadecimal provides a more compact and readable format compared to binary.
3. **Arithmetic Operations**
   Number systems are crucial for performing arithmetic operations in digital circuits. Binary arithmetic, including addition, subtraction, multiplication, and division, is fundamental in computer processing. Arithmetic Logic Units (ALUs) in CPUs perform calculations using binary numbers, which are then interpreted or converted as needed into decimal or other formats for output.
4. **Digital Communication**
   In digital communication systems, data is often transmitted in binary form. Protocols such as Ethernet and USB rely on binary encoding to ensure reliable data transfer. Furthermore, error detection and correction methods, such as parity bits and checksums, utilize binary numbers to identify and correct errors during transmission.
5. **Control Systems**
   Octal (base-8) and hexadecimal systems are often used in control systems, especially in programming and configuring devices. For instance, octal is used in Unix file permissions, where each octal digit represents a set of access rights. Similarly, hexadecimal is used in configuring microcontrollers, allowing for efficient setting of control registers and memory addresses.
6. **Color Representation in Graphics**
   Hexadecimal is widely used in web design and graphic applications to represent colors. In this context, colors are often expressed as RGB (Red, Green, Blue) values, with each color component ranging from 00 to FF in hexadecimal, facilitating a straightforward way to specify colors in digital graphics.
7. **Embedded Systems**
   Embedded systems often utilize binary and hexadecimal representations for efficient programming and debugging. Many programming languages and assembly languages allow developers to write and read code in hexadecimal, making it easier to interface with hardware components and manage memory.

2.8 SUMMARY:

This chapter explores the various number systems used in digital electronics, emphasizing their importance in data representation and processing. It covers the binary, decimal, octal, and hexadecimal systems, detailing how each system functions and how to convert between them. The binary system, with its two digits (0 and 1), forms the foundation of digital computing, while octal and hexadecimal systems offer more compact representations of binary data. Understanding these number systems is crucial for designing and working with digital circuits, as they directly influence how data is stored, transmitted, and processed.

2.9 TECHNICAL TERMS:

Number system, decimal, binary, octal, hexadecimal

2.10 SELF ASSESSMENT QUESTIONS:

Essay Questions:

1) Explain the significance of the binary number system in digital electronics.
2) Discuss its advantages and applications in modern computing.
3) Compare and contrast the hexadecimal and octal number systems
4) Describe the process of converting a decimal number to binary and vice versa with examples
5) Analyze the role of different number systems in computer programming and data storage.

Short Questions:

1) What are the main digits used in the octal number system?
2) How does the hexadecimal system relate to binary in terms of data representation?
3) Convert the binary number 11010 to decimal.
4) What is the primary advantage of using the hexadecimal system over the binary system in programming?
5) Explain how to convert the decimal number 25 to binary. What is the resulting binary value?

2.11 FURTHER READINGS:

1) M.Morris Mano, “Computer System Architecture”, 3rd Edition, Pearson Education (2008).
2) V. Rajaraman, T. Radha Krishnan, “Computer Organization and Architecture”, PHI
3) BehroozParhami, “Computer Architecture”, Oxford (2007).
4) ISRD group, “Computer Organization”, ace series, TMH (2007).
5) William Stallings, “Computer Organization and Architecture-Designing for Performance”, Pearson Education (2005).

Dr. U. Surya Kameswari

LESSON-3
COMBINATIONAL CIRCUITS

OBJECTIVES:

After going through this lesson, you will be able to

- Understand the role and importance of Integrated Circuits (ICs) in modern digital electronics
- Learn how encoders operate and their importance
- Describe the function and applications of Decoders and Multiplexers:
- Identify the differences between half adders and full adders
- Explore the function and implementation of Binary Counters and Memory Units.

STRUCTURE OF THE LESSION:

3.1 INTEGRATED CIRCUITS:

Integrated circuits are used to build digital circuits. An integrated circuit, or IC, is a chip, which is a small silicon semiconductor crystal that holds the electronic parts for the digital gates. Inside the chip, the different gates are linked together to make the circuit that is needed. The chip is put in a clay or plastic case, and thin gold wires connect to pins on the outside to make the integrated circuit. In a small IC package, there may be 14 pins. In a bigger package, there may be 100 pins or more. The package of each integrated circuit has a number written on it so that it can be found. Every company that makes ICs puts out a data book or catalogue with full specifications and all the important details about those products.

The number of gates that can fit on a single chip has grown a lot as IC technology has gotten better. An ordinary way to tell the difference between chips with a few internal gates and chips with hundreds or thousands of gates is to call the box a small, medium, or large-scale integration device.

Small-Scale Integration (SSI): Small-Scale Integration (SSI) refers to the integration of a few transistors, typically up to hundreds, onto a single chip. This early stage of integrated circuit technology, prominent in the 1960s, allowed for the creation of basic logic gates and simple functions, such as flip-flops and multiplexers. SSI circuits were foundational in the development of early computers and electronic devices, significantly reducing the physical size and power consumption compared to discrete component designs.

Medium-Scale Integration (MSI): Medium-Scale Integration (MSI) technology integrates hundreds to a few thousand transistors onto a single chip. Emerging in the late 1960s and 1970s, MSI enabled more complex functions such as simple processors, memory units, and digital adders. This advancement allowed for the creation of more sophisticated electronic systems and was instrumental in the evolution of microprocessors and memory chips, facilitating the development of early computing and telecommunications equipment.

Large-Scale Integration (LSI): Large-Scale Integration (LSI) involves integrating thousands to tens of thousands of transistors on a single chip. This technology, which gained prominence in the 1970s and 1980s, enabled the production of entire processors, memory arrays, and complex logic circuits on a single chip. LSI significantly advanced the capabilities of electronic devices, paving the way for the development of personal computers, video game consoles, and a wide range of consumer electronics by providing greater processing power and efficiency.

Very Large-Scale Integration (VLSI): Very Large-Scale Integration (VLSI) is the process of creating integrated circuits by combining millions of transistors onto a single chip. This technology, which began to emerge in the late 1980s and continues to evolve, revolutionized the electronics industry by enabling the creation of highly complex and powerful microprocessors, memory chips, and application-specific integrated circuits (ASICs). VLSI technology is fundamental to modern computing, telecommunications, and a myriad of digital devices, driving innovation in fields such as artificial intelligence, mobile communications, and high-performance computing.

Ultra Large-Scale Integration (ULSI): Ultra Large-Scale Integration (ULSI) pushes the boundaries of integration by incorporating billions of transistors on a single chip. This state-of-the-art technology, which has been developing since the 2000s, supports the creation of extremely powerful and compact integrated circuits. ULSI is essential for the development of cutting-edge processors, advanced system-on-chip (SoC) solutions, and high-density memory, enabling the exponential growth in performance, functionality, and energy efficiency of modern electronic devices. It is a cornerstone of advancements in fields such as cloud computing, Internet of Things (IoT), and next-generation artificial intelligence applications.

3.2 COMBINATIONAL CIRCUITS:

A combinational circuit is a connected arrangement of logic gates with a set of inputs and outputs. At any given time, the binary values of the outputs are a function of the binary combination of the inputs. A block diagram of a combinational circuit is shown in Fig. 3.1

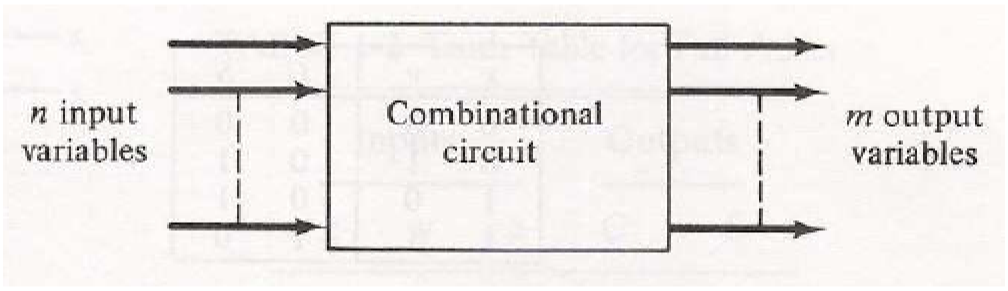


Figure 3.1 block diagram of a combinational circuits

The n binary input variables come from an external source, the m binary output variables go to an external destination, and in between there is an interconnection of logic gates. A combinational circuit transforms binary information from the given input data to the required output data. Combinational circuits are employed in digital computers for generating binary control decisions and for providing digital components required for data processing.

A combinational circuit can be described by a truth table showing the binary relationship between the n input variables and the m output variables. The truth table lists the corresponding output binary values for each of the 2^n input combinations. A combinational circuit can also be specified with m Boolean functions, one for each output variable. Each output function is expressed in terms of the n input variables.

3.3 DECODERS:

Discrete quantities of information are represented in digital computers with binary codes. A binary code of n bits is capable of representing up to 2^n distinct elements of the coded information. A decoder is a combinational circuit that converts binary information from the n coded inputs to a maximum of 2^n unique outputs. If the n-bit coded information has unused bit combinations, the decoder may have less than 2^n outputs.

The decoders presented in this section are called n-to-m-line decoders, where $m \leq 2^n$. Their purpose is to generate the 2^n (or fewer) binary combinations of the n input variables. A decoder has n inputs and m outputs and is also referred to as an n x m decoder.

Figure 3.2 illustrates the logic diagram of a 3-to-8-line decoder. The three binary input variables, Ao, A1, and A, are decoded into eight outputs, each of which represents one of the combinations. The complement of the inputs is provided by the three inverters, and each of the eight AND gates generates one of the binary combinations. A binary-to-octal conversion is a specific application of this decoder. The outputs represent the eight integers of the octal number system, while the input variables represent a binary number. Nevertheless, a 3-to-8-

line decoder can be employed to decode any 3-bit code, resulting in eight outputs, one for each binary code combination.

Commercial decoders include one or more enable inputs to control the operation of the circuit. The decoder of Fig. 3.2 has one enable input, E. The decoder is enabled when E is equal to 1 and disabled when E is equal to 0.

The operation of the decoder can be clarified using the truth table listed in Figure 3.3 . When the enable input E is equal to 0, all the outputs are equal to 0 regardless of the values of the other three data inputs. The three x's in the table designate don't-care conditions. When the enable input is equal to I, the decoder operates in a normal fashion. For each possible input combination, there are seven outputs that are equal to 0 and only one that is equal to I. The output variable whose value is equal to 1 represents the octal number equivalent of the binary number that is available in the input data lines.

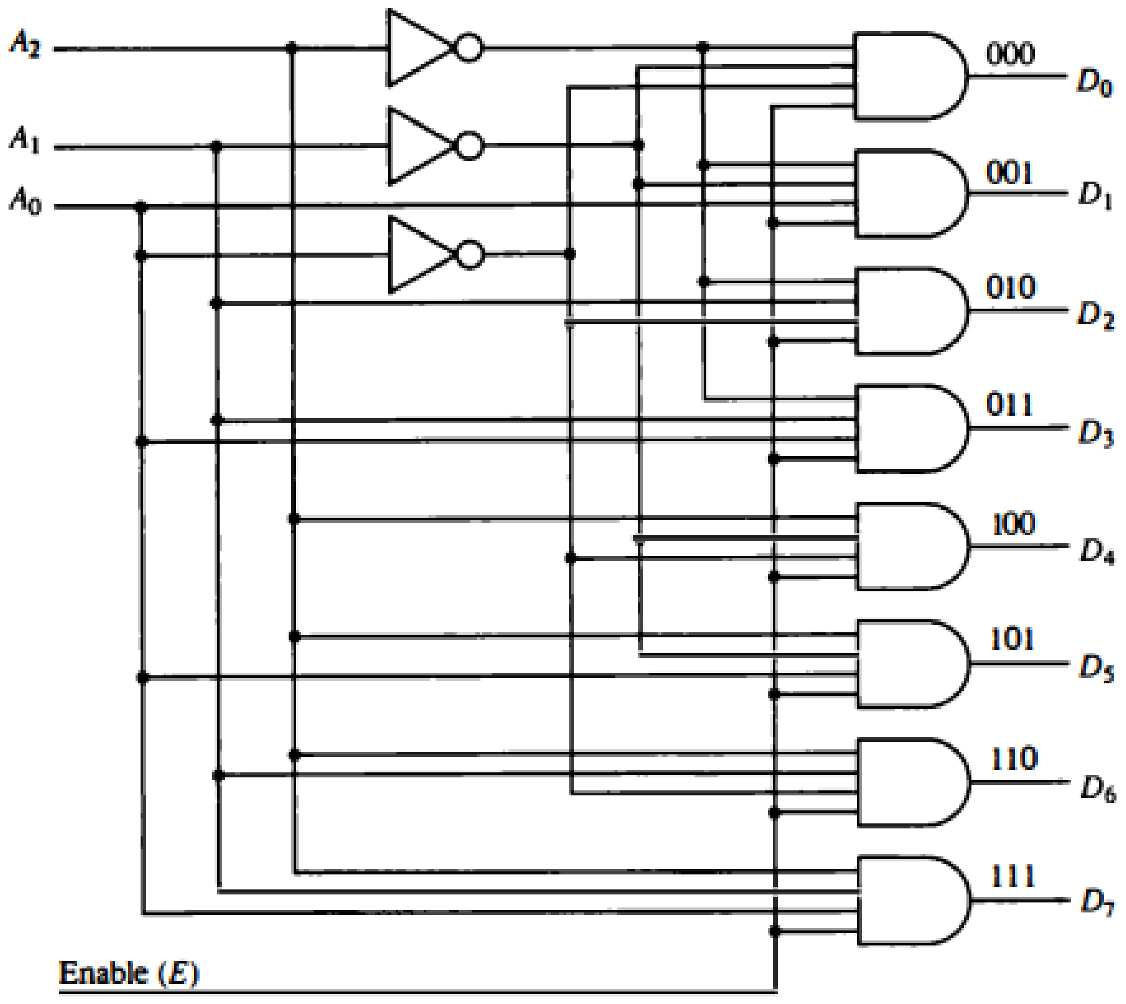


Figure 3.2: 3-to 8 line decoder

Enable	Inputs			Outputs							
E	A_2	A_1	A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	×	×	×	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

Figure 3.3. Truth Table for 3-to-8-Line Decoder

3.4 ENCODERS:

Encoders are digital circuits that convert information from one format or code to another, typically transforming a set of inputs into a binary code. They are used to reduce the number of data lines needed to represent information, thus simplifying the design and operation of digital systems. An encoder has multiple input lines and fewer output lines, where each output combination represents a unique input line.

A common type of encoder is the binary encoder, which takes 2^n input lines and produces an nnn-bit binary output. For example, a 4-to-2 encoder has four input lines and two output lines. Encoders are widely used in applications such as data compression, digital signal processing, and when converting analog signals into digital formats. Additionally, priority encoders handle scenarios where multiple inputs are active simultaneously, providing an output corresponding to the highest-priority active input.

There are different types of Encoders which are mentioned below.

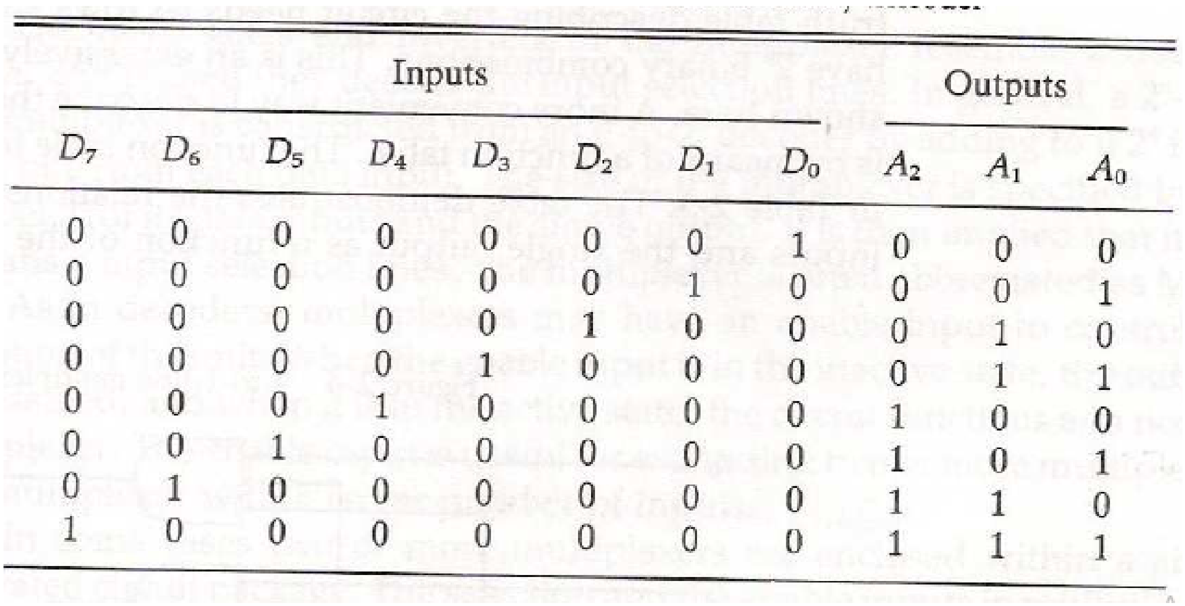
- 4 to 2 Encoder
- Octal to Binary Encoder (8 to 3 Encoder)
- Decimal to BCD Encoder
- Priority Encoder

3.4.1 Octal to binary encoder

An encoder is a digital circuit that performs the inverse operation of a decoder. An encoder has 2n (or less) input lines and n output lines. The output lines generate the binary code corresponding to the input value. An example of an encoder is the octalto-binary encoder, whose truth table is given below.

It is assumed that only one input has a vaue of 1 at any given time; otherwise, the circuit has no meaning.

Table 3.1 Truthtable for binary to octal encoder



Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

The encoder can be implemented with OR gates whose inputs are determined directly from the truth table. Output A0 =1 if the input octal digit is 1 or 3 or 5 or 7. Similar conditions apply for other two outputs.

These conditions can be expressed by the following Boolean functions:

A0 = D1 + D3 + D5 + D7

A1 = D2 + D4 + D6 + D7

A2 = D4 + D5 + D6 + D7

The encoder can be implemented with three OR gates.

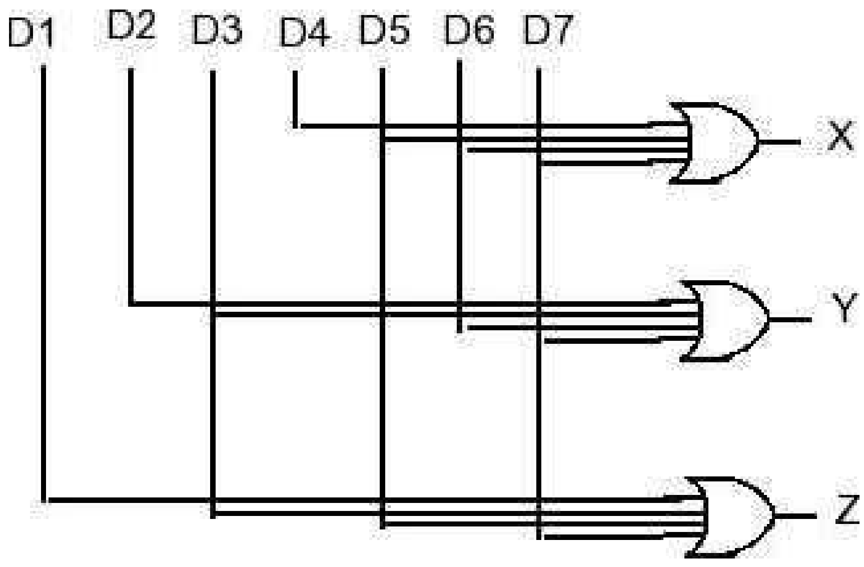


Figure 3.4. 3 to 8 encoder

3.5 MULTIPLEXERS:

A multiplexer is a combinational circuit that receives binary information from one of 2^ninput data lines and directs it to a single output line. The selection of a particular input data line for the output is determined by a set of selection inputs. A 2^n -to-1 multiplexer has 2^ninput data lines and n input selection lines whose bit combinations determine which input data are selected for the output.

A 4-to-1-line multiplexer is shown in Fig. 3.3. Each of the four data inputs I0 through I, is applied to one input of an AND gate. The two selection inputs S_1 and S_0 are decoded to select a particular AND gate. The outputs of the AND gates are applied to a single OR gate to provide the single output. To demonstrate the circuit operation, consider the case when S_1S_0 = 10. The AND gate associated with input I, has two of its inputs equal to 1. The third input of the gate is connected to I3. The other three AND gates have at least one input equal to 0, which makes their outputs equal to 0. The OR gate output is now equal to the value of I_2, thus providing a path from the selected input to the output.

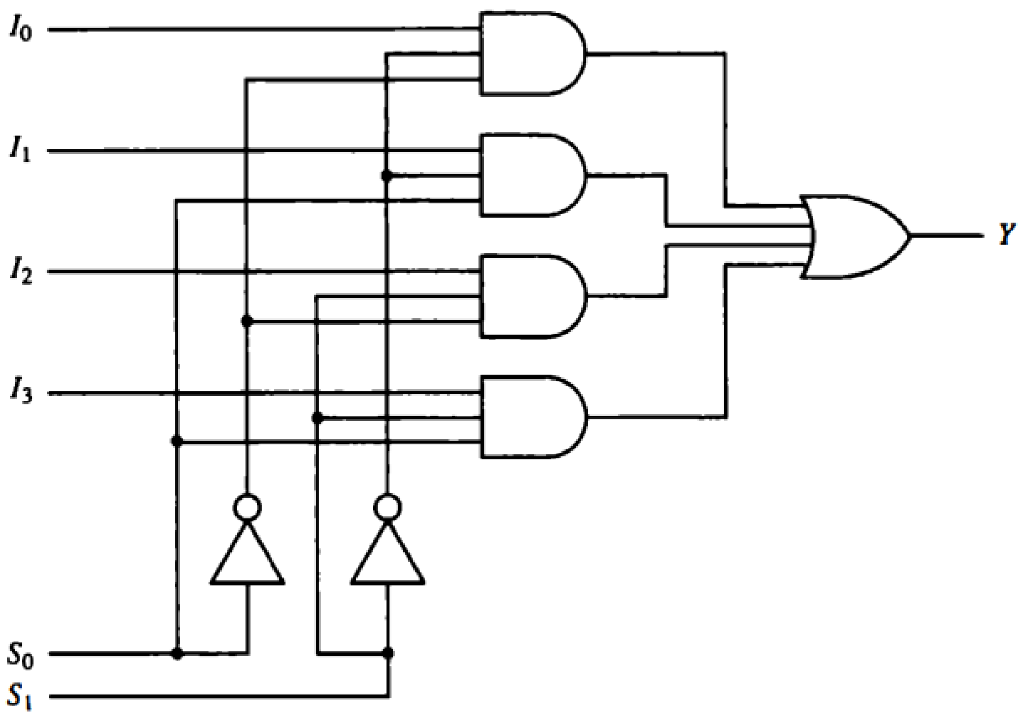


Figure 3.5: 4-to 1 line multiplexer

The 4-to-1 line multiplexer of Fig. 3.3 has six inputs and one output. A truth table describing the circuit needs 64 rows since six input variables can have 2^6 binary combinations. This is an excessively long table and will not be shown here. A more convenient way to describe the operation of multiplexers is by means of a function table. The function table for the multiplexer is shown in Figure 3.3. The table demonstrates the relationship between the four data inputs and the single output as a function of the selection inputs S_1 and $S_{0.}$

Select		Output
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

Figure 3.6: Function table for 4- to-1 line multiplexer

The AND gates and inverters in the multiplexer resemble a decoder circuit, and indeed they decode the input selection lines. In general, a 2^n -to-1- line multiplexer is constructed from an n-to-2^n decoder by adding to it 2^n input lines, one from each data input. The size of the multiplexer is specified by the number 2" of its data inputs and the single output. It is then implied that it also contains n input selection lines. The multiplexer is often abbreviated as MUX. As in decoders, multiplexers may have an enable input to control the operation of the unit. When the enable input is in the inactive state, the outputs are disabled, and when it is in the active state, the circuit functions as a normal multiplexer. The enable input is useful for expanding two or more multiplexers to a multiplexer with a larger number of inputs.

3.6 HALF ADDER:

An adder is a digital logic circuit in electronics that is extensively used for the addition of numbers.

The half adder adds two binary digits and produces two outputs as sum and carry. It consists of one XOR logic gate producing “SUM” and one AND gate producing “CARRY” as outputs.

There are two inputs and two outputs in a Half Adder. Inputs are named as A and B, and the outputs are named as Sum (S) and Carry (C).

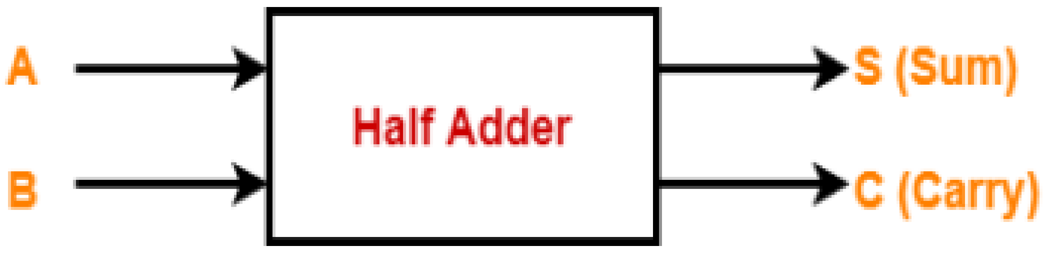


Figure 3.7 Block Diagram of an half adder

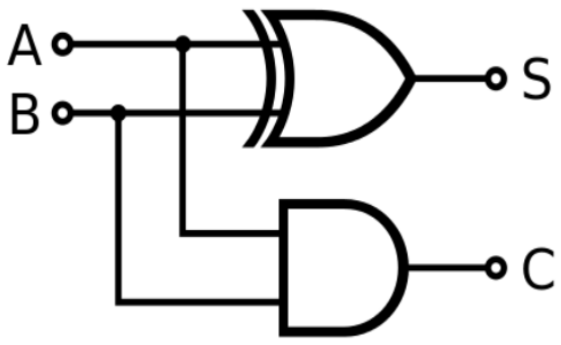


Figure 3.8 Half adder Logic Circuit

INPUTS		OUTPUTS	
A	B	S(Sum)	C(Carry)
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

Figure 3.9: Half adder Truth Table

The output S represents the least significant bit of sum.

The Boolean functions for the two outputs can be obtained directly from the truth table

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

$$C = xy$$

The major drawback of HALF ADDER circuit is that it can only add two inputs and if there is any carry it is neglected. Thus, the process is incomplete. To overcome this drawback Full Adder is designed.

3.7 FULL ADDER:

A full-adder is a combinational circuit that forms the arithmetic sum of three input bits. It consists of three inputs and two outputs. Two of the input variables, denoted by x and y, represent the two significant bits to be added. The third input, z, represents the carry from the previous lower significant position.

Two outputs are necessary because the arithmetic sum of three binary digits ranges in value from 0 to 3, and binary 2 or 3 needs two digits. The two outputs are designated by the symbols S (for sum) and C (for carry). The binary variable S gives the value of the least significant bit of the sum. The binary variable C gives the output carry. The truth table of the full-adder is shown in Table 3.2. The eight rows under the input variables designate all possible combinations that the binary variables may have. The value of the output variables are determined from the arithmetic sum of the input bits. When all input bits are 0, the output is 0. The S output is equal to 1 when only one input is equal to 1 or when all three inputs are equal to 1. The C output has a carry of 1 if two or three inputs are equal to 1.

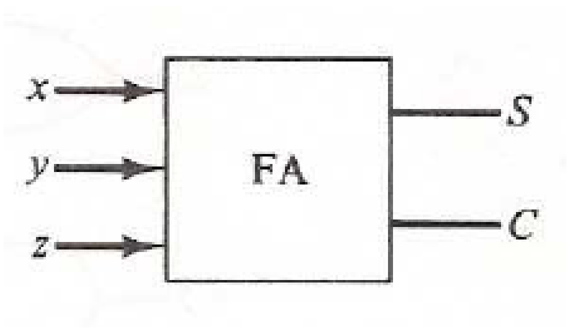


Figure 3.10 Block diagram

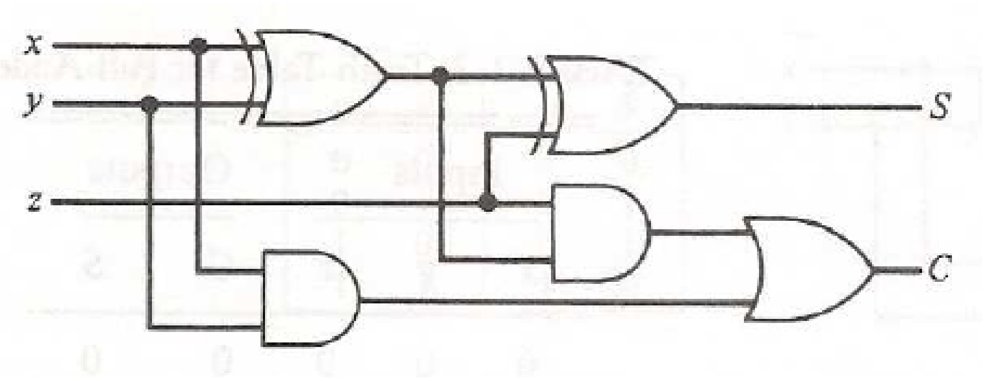


Figure 3.11 Logic diagram

Inputs			Outputs	
x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Figure 3.12 Truth table for full adder

The Boolean functions for the two outputs can be obtained directly from the truth table

$$S = x \oplus y \oplus z$$

$$C = xy + (xy' + x'y)\, z$$

$$C = xy + (x \oplus y) z$$

Applications of half adder and full adder

There are many applications of Half Adder and Full Adder. These circuits are used basically in building of any digital electronics device. Half adder and Full adder can be used to make other arithmetical and logical circuits like ALU. These circuits are mainly designed for the addition of binary number, but they can be used in various other applications like binary code decimal, address decoding, table index calculation, etc.

3.8 MEMORY UNIT:

A memory unit is a collection of storage cells together with associated circuits needed to transfer information in and out of storage. The memory stores binary information in groups of bits called words. A word in memory is an entity of bits that move in and out of storage as a unit. A memory word is a group of 1's and 0's and may represent a number, an instruction code, one or more alphanumeric characters, or any other binary-coded information.

A group of byte eight bits is called a byte. Most computer memories use words whose number of bits is a multiple of 8. Thus a 16-bit word contains two bytes, and a 34-bit word is made up of four bytes. The capacity of memories in commercial computers is usually stated as the total number of bytes that can be stored.

The internal structure of a memory unit is specified by the number of words it contains and the number of bits in each word. Special input lines called address lines select one particular word.

Each word in memory is assigned an identification number, called an address, starting from 0 and continuing with 1, 2, 3, up to 2k - 1 where k is the number of address lines. The selection of a specific word inside the memory is done by applying the k-bit binary address to the address lines.

A decoder inside the memory accepts this address and opens the paths needed to select the bits of the specified word. Computer memories may range from 1024 words, requiring an address of 10 bits, to 2^{32} words, requiring 32 address bits.

4 Bits	1 Nibble	-
8 Bits	1 Byte	-
1024 Bytes	1 KB	2^{10} bytes
1024 KB	1 MB	2^{20} bytes
1024 MB	1 GB	2^{30}bytes
1024 GB	1 TB	2^{40}bytes
1024 TB	1 PB	2^{50}bytes
1024 PB	1 EB	2^{60}bytes
1024 EB	1 ZB	2^{70}bytes
1024 ZB	1 YB	2^{80}bytes

Two major types of memories are used in computer systems: random access memory (RAM) and read-only memory (ROM).

Random-Access Memory

In random-access memory (RAM) the memory cells can be accessed for information transfer from any desired random location.

That is, the process of locating a word in memory is the same and requires an equal amount of time no matter where the cells are located physically in memory: thus the name "random access."

Communication between a memory and its environment is achieved through data input and output lines, address selection lines, and control lines that specify the direction of transfer.

A block diagram of a RAM unit is shown in Figure 3.13. The n data input lines provide the information to be stored in memory, and the n data output lines supply the information coming out of memory.

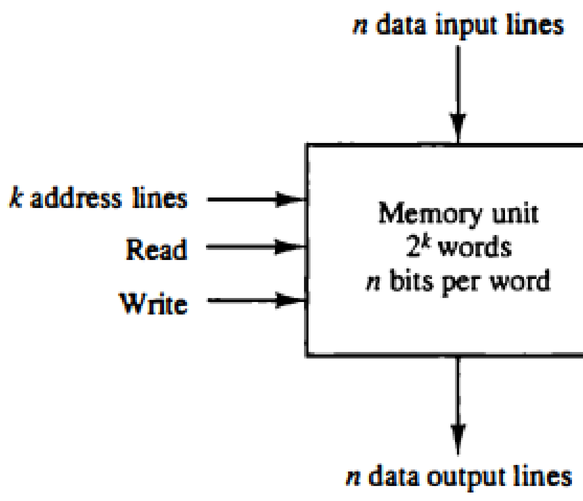


Figure 3.13 Block diagram of random access memory (RAM)

The k address lines provide a binary number of k bits that specify a particular word chosen among the 2k available inside the memory. The two control inputs specify the direction of transfer desired.

The two operations that a random-access memory can perform are the write and read operations.

The write signal specifies a transfer-in operation and the read signal specifies a transfer-out operation. On accepting one of these control signals, the internal circuits inside the memory provide the desired function.

The steps that must be taken for the purpose of transferring a new word to be stored into memory are as follows:

- Apply the binary address of the desired word into the address lines
- Apply the data bits that must be stored in memory into the data input lines.
- Activate the write input.

The memory unit will then take the bits presently available in the input data lines and store them in the word specified by the address lines.

The steps that must be taken for the purpose of transferring a stored word out of memory are as follows:

- Apply the binary address of the desired word into the address lines.
- Activate the read input.

The memory unit will then take the bits from the word that has been selected by the address and apply them into the output data lines. The content of the selected word does not change after reading.

Read-Only Memory:

As the name implies, a read-only memory (ROM) is a memory unit that performs the read operation only; it does not have a write capability.

This implies that the binary information stored in a ROM is made permanent during the hardware production of the unit and cannot be altered by writing different words into it.

Whereas a RAM is a general-purpose device whose contents can be altered during the computational process, a ROM is restricted to reading words that are permanently stored within the unit. The binary information to be stored, specified by the designer, is then embedded in the unit to form the reqttired interconnection pattern.

ROMs come with special internal electronic fuses that can be "programmed" for a specific configuration. Once the pattern is established, it stays within the unit even when power is turned off and on again. An m x n ROM is an array of binary cells organized into m words of n bits each.

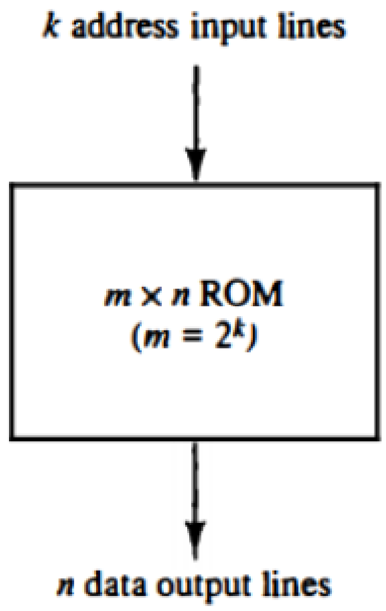


Figure 3.14: Block diagram of read only memory (ROM).

As shown in the block diagram of Figure 3.14, a ROM has k address input lines to select one of 2k = m words of memory, and n output lines, one for each bit of the word.

An integrated circuit ROM may also have one or more enable inputs for expanding a number of packages into a ROM with larger capacity.

The ROM does not need a read-control line since at any given time, the output lines automatically provide the n bits of the word selected by the address value. Because the outputs are a function of only the present inputs (the address lines), a ROM is classified as a combinational circuit. In fact, a ROM is constructed internally with decoders and a set of OR gates.

There is no need for providing storage capabilities as in a RAM, since the values of the bits in the ROM are permanently fixed. ROMs find a wide range of applications in the design of digital systems. Basically, a ROM generates an input-output relation specified by a truth table.

As such, it can implement any combinational circuit with k inputs and n outputs. When employed in a computer system as a memory unit, the ROM is used for storing fixed programs that are not to be altered and for tables of constants that are not subject to change.

ROM is also employed in the design of control units for digital computers. As such, they are used to store coded information that represents the sequence of internal control variables needed for enabling the various operations in the computer.

A control unit that utilizes a ROM to store binary control information is called a microprogrammed control unit.

Types of ROM

Masked ROM (MROM): Masked ROM is a type of ROM where the data is written during the manufacturing process. This makes it highly reliable and cost-effective for mass production, but it cannot be altered once fabricated. MROM is used in applications where the data remains constant, such as embedded systems in consumer electronics.

Programmable ROM (PROM): Programmable ROM is a type of ROM that is programmed after the manufacturing process. Using a special device called a PROM programmer, data can be written once. PROM provides flexibility over MROM by allowing customization of the data after production, but like MROM, it cannot be reprogrammed or erased once written. It's used in applications where the data needs to be set once and never changed.

Erasable Programmable ROM (EPROM): EPROM allows data to be erased and reprogrammed multiple times using ultraviolet (UV) light. The chip has a small quartz window through which the UV light passes to erase the data. After erasing, new data can be programmed using an EPROM programmer. EPROMs are used in development and prototyping environments where the firmware needs to be updated frequently.

Electrically Erasable Programmable ROM (EEPROM): EEPROM is similar to EPROM but allows data to be erased and reprogrammed electrically. This makes it more convenient and versatile than EPROM, as it does not require UV light for erasure. EEPROM can be reprogrammed in-circuit, making it ideal for applications requiring frequent updates, such as BIOS chips in computers and firmware in embedded systems.

Flash Memory: Flash memory is a type of EEPROM that allows for block-wise erasure and reprogramming, offering higher densities and faster access times. Flash memory is widely used in applications requiring high storage capacity and frequent data updates, such as USB flash drives, solid-state drives (SSDs), and memory cards. Its robustness and efficiency make it a popular choice for a broad range of consumer electronics and industrial applications.

3.9 SUMMARY:

This chapter covers essential concepts and components of combinational logic, starting with an introduction to integrated circuits, which form the basis for implementing these systems. It explores various types of combinational circuits, including decoders and encoders, which are used to convert information between different formats. Multiplexers are discussed as devices that select one of many input signals and forward the chosen input into a single line. The chapter also delves into arithmetic circuits, such as half adders and full adders, which perform basic addition operations. Lastly, it touches on the memory unit, providing a foundation for understanding how combinational circuits contribute to the overall functionality of digital memory systems.

3.10 TECHNICAL TERMS:

Integrated Circuit (IC), Decoders, Encoders, half adder, full adder, Multiplexers, Memory Unit, RAM, ROM

3.11 SELF ASSESSMENT QUESTIONS:

Essay questions:

1) Explain encoder with the help of diagram
2) Describe the functionality of decoders
3) What is a multiplexer, and how does it operate? Explain with an example.
4) Write about ROM and RAM Chips. And also design of the main memory of capacity 1024 X 12 words with RAM Chips size as 128X12 words and ROM chip size as 512X12 words and show how these ICs are interconnected with CPU.

Short Questions:

1) How does a binary decoder operate, and what are its typical applications?
2) Explain the function of a multiplexer and how it differs from a demultiplexer.
3) Differentiate between half adders and full adders in terms of their structure and functionality
4) What are the primary differences between RAM and ROM?

3.12 FURTHER READINGS:

1) M.Morris Mano, “Computer System Architecture”, 3rd Edition, Pearson Education (2008).
2) V. Rajaraman, T. Radha Krishnan, “Computer Organization and Architecture”, PHI.
3) Behrooz Parhami, “Computer Architecture”, Oxford (2007).
4) ISRD group, “Computer Organization”, ace series, TMH (2007).
5) William Stallings, “Computer Organization and Architecture-Designing for Performance”, Pearson Education (2005).

Dr. U. Surya Kameswari

LESSON-4
LOGIC FAMILIES

OBJECTIVES:

After going through this lesson, you will be able to

- Understand the Fundamentals of Digital Logic Families
- Analyze the Characteristics of Logic Families
- Explore Bipolar and Unipolar Logic Families
- Learn techniques for interfacing different logic families

STRUCTURE OF THE LESSION:

4.1 INTRODUCTION:

Digital logic families are the foundational building blocks of digital electronics, encompassing a range of technologies used to implement logic gates and circuits. These families are categorized based on the underlying electronic components and circuitry, which define their electrical characteristics and performance. Digital logic families are crucial in the design and operation of various digital devices, including computers, communication systems, and embedded systems, where they perform operations such as logic processing, arithmetic calculations, and data storage.

The primary classification of digital logic families includes bipolar and unipolar families. Bipolar families, such as Transistor-Transistor Logic (TTL) and Emitter-Coupled Logic (ECL), utilize bipolar junction transistors (BJTs) to implement logic functions. These families are known for their high speed but typically consume more power. On the other hand, unipolar families, such as Complementary Metal-Oxide-Semiconductor (CMOS), rely on field-effect transistors (FETs) and are known for their low power consumption and high noise immunity. The choice of a specific logic family depends on the application requirements, balancing factors like speed, power consumption, and cost.

Understanding the various digital logic families is essential for engineers and designers to select the appropriate technology for their projects. Each family has distinct characteristics that make it suitable for particular applications. For instance, CMOS is widely used in battery-operated devices due to its low power consumption, while TTL might be chosen for applications requiring faster switching speeds. As technology evolves, newer logic families emerge, offering improved performance and expanding the possibilities for digital system design.

Discrete circuit components were used to create the first logic circuit. These intricate circuits can be created on a tiny piece of silicon or another semiconductor material by using sophisticated techniques to miniaturize them. The term integrated circuit (IC) refers to such a circuit. All digital circuits are now available in integrated circuit (IC) form. Various circuit layouts and manufacturing technologies are employed in the production of digital integrated circuits. This leads to a certain logic family.

This design ensures that every logic family has the same electrical properties, including noise margin, power dissipation, operating speed, and supply voltage range. We go over the types and importance of logic families in this module. There is also discussion of the importance of both positive and negative reasoning. Furthermore, a detailed discussion is held on various attributes that are crucial in determining the logic family for any given circuit design.

4.1.1 Importance of digital families:

The majority of electronic devices rely on various digital systems to function. Digital ICs of some type are used in all digital systems. All digital circuits (ICs) utilized in the design process should belong to the same logic family for the benefit of design simplicity and compatibility while building any complicated digital system.

Connecting various logic circuits together is required for the system's expansion. One needs a circuit with comparable properties in order to connect the output of one logic circuit to the input of another. To keep digital logic ICs compatible, an interface circuit must be designed if the electrical input/output (IO) characteristics of these logic circuits differ. The electrical properties of the logic circuits will be compatible with this interface circuit. This guarantees compatibility for the circuit's correct operation. This is the fundamental idea of the logic circuit family. Members of a logic family share electrical traits with one another. The compatibility of various logic families with respect to various family traits and parameters must be taken into account while designing digital logic circuits.

4.2 CLASSIFICATION OF LOGIC FAMILIES:

Families of logic are the logic circuits with same electrical properties. This collection of compatible ICs performs different logic operations using the same logic levels and supply voltages. They are created utilizing a certain circuit arrangement sometimes known as a Logic family. Every logic family has a basic gate whose circuit design is the same.

Designed with consideration for the fundamental electronic components—resistors, diodes, transistors, and MOSFET—or combinations of any one of these components—the logic family is consequently, families of logic are categorized according to the building of the fundamental logical circuits. Table 1 lists many several logic families of digital ICs that have been brought commercially.

Logic families are classified according to the principle type of electronic components used in their circuitry as shown in Figure 4.1.

They are

a. Bipolar ICs: which uses diodes and transistors (BJT)

b. Unipolar ICs: which uses MOSFETs

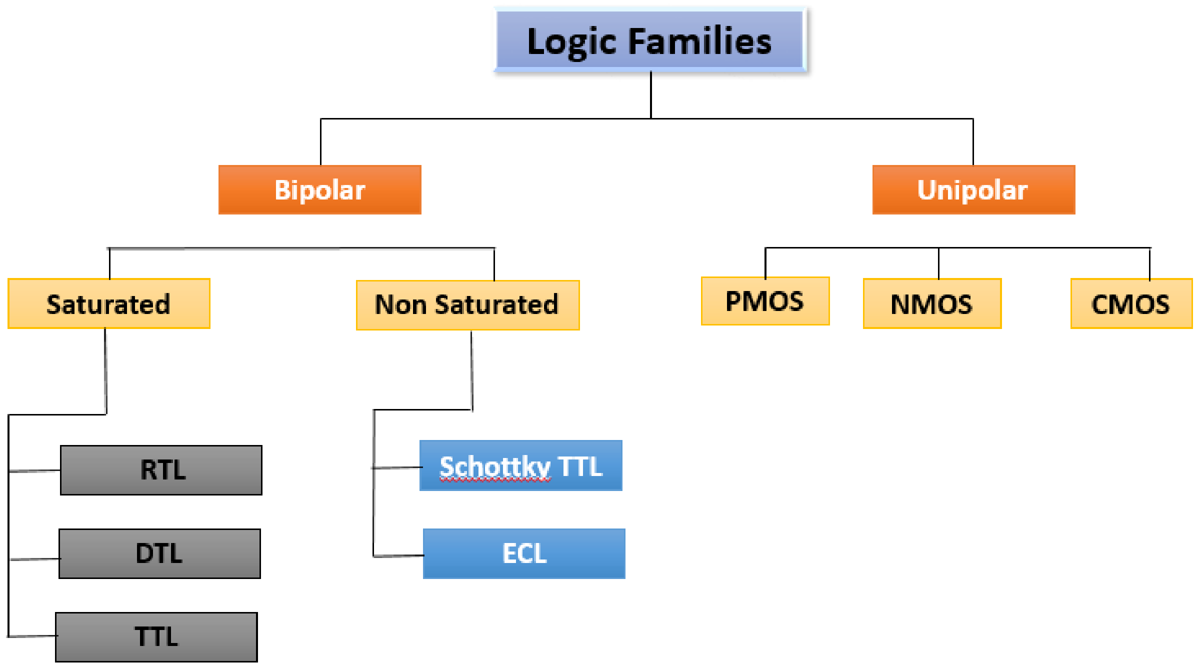


Figure 4.1 classification of logic families

4.2.1 Bipolar Logic Families

Bipolar logic families utilize bipolar junction transistors (BJTs) as their primary switching elements. These families are known for their faster switching speeds but generally have higher power consumption compared to unipolar families. The most common bipolar logic families include:

- Transistor-Transistor Logic (TTL): Known for its robustness and widespread use in industrial and commercial applications. Variants include Standard TTL, Low Power TTL (LPTTL), and High-Speed TTL (HSTTL).
- Emitter-Coupled Logic (ECL): Characterized by its extremely fast switching speeds and is used in high-frequency applications. ECL circuits operate with differential amplifiers to minimize noise and increase speed.

4.2.2Unipolar Logic Families

Unipolar logic families use metal-oxide-semiconductor field-effect transistors (MOSFETs) as their switching elements. These families are favored for their lower power consumption and high input impedance, making them suitable for battery-powered and high-density applications. The main unipolar logic families include:

- Complementary Metal-Oxide-Semiconductor (CMOS): Offers a balance of speed and power efficiency, making it the most widely used logic family in modern electronics. It is highly scalable and forms the basis for most integrated circuits today.
- NMOS and PMOS Logic: These are earlier forms of MOS logic families, where NMOS uses n-type MOSFETs and PMOS uses p-type MOSFETs. They are simpler than CMOS but less power-efficient and are mostly obsolete in favor of CMOS technology.

Each of these logic families is selected based on the specific requirements of the application, such as speed, power consumption, noise immunity, and cost.

4.3 CHARACTERISTICS OF DIGITAL LOGIC FAMILIES:

1. HIGH-level input current, I_{IH}.

This is the current flowing into (taken as positive) or out of (taken as negative) an input, when a HIGH-level input voltage equal to the minimum HIGH-level output voltage specified for the family is applied. In the case of bipolar logic families such as TTL, the circuit design is such that this current flows into the input pin and is therefore specified as positive. In the case of CMOS logic families, it could be either positive or negative, and only an absolute value is specified in this case.

2. LOW-level input current, I_{IL}.

The LOW-level input current is the maximum current flowing into (taken as positive) or out of (taken as negative) the input of a logic function, when the voltage applied at the input equals the maximum LOW-level output voltage specified for the family. In the case of bipolar logic families such as TTL, the circuit design is such that this current flows out of the input pin and is therefore specified as negative. In the case of CMOS logic families, it could be either positive or negative. In this case, only an absolute value is specified.

3. HIGH-level output current, I_{OH}.

This is the maximum current flowing out of an output when the input conditions are such that the output is in the logic HIGH state. It is normally shown as a negative number. It informs us about the current sourcing capability of the output. The magnitude of IOH determines the number of inputs the logic function can drive when its output is in the logic HIGH state. For example, for the standard TTL family, the minimum guaranteed IOH is −400 µA, which can drive 10 standard TTL inputs with each requiring 40 µA in the HIGH state.

4. LOW-level output current, I_{OL}.

This is the maximum current flowing into the output pin of a logic function when the input conditions are such that the output is in the logic LOW state. It informs us about the current sinking capability of the output. The magnitude of IOL determines the number of inputs the logic function can drive when its output is in the logic LOW state. For example, for the standard TTL family, the minimum guaranteed IOL is 16 mA, which can drive 10 standard TTL inputs with each requiring 1.6mA in the LOW state.

5. HIGH-level input voltage, V_{IH}.

This is the minimum voltage level that needs to be applied at the input to be recognized as a legal HIGH level for the specified family. For the standard TTL family, a 2 V input voltage is a legal HIGH logic state.

6. LOW-level input voltage, V_{IL}.

This is the maximum voltage level applied at the input that is recognized as a legal LOW level for the specified family. For the standard TTL family, an input voltage of 0.8 V is a legal LOW logic state.

7. HIGH-level output voltage, V_{OH}.

This is the minimum voltage on the output pin of a logic function when the input conditions establish logic HIGH at the output for the specified family. In the case of the standard TTL family of devices, the HIGH level output voltage can be as low as 2.4V and still be treated as a legal HIGH logic state. It may be mentioned here that, for a given logic family, the VOH specification is always greater than the VIH specification to ensure output-to-input compatibility when the output of one device feeds the input of another.

8. LOW-level output voltage, $V_{OL.}$

This is the maximum voltage on the output pin of a logic function when the input conditions establish logic LOW at the output for the specified family. In the case of the standard TTL family of devices, the LOW-level output voltage can be as high as 0.4V and still be treated as a legal LOW logic state. It may be mentioned here that, for a given logic family, the VOL specification is always smaller than the VIL specification to ensure output-to-input compatibility when the output of one device feeds the input of another.

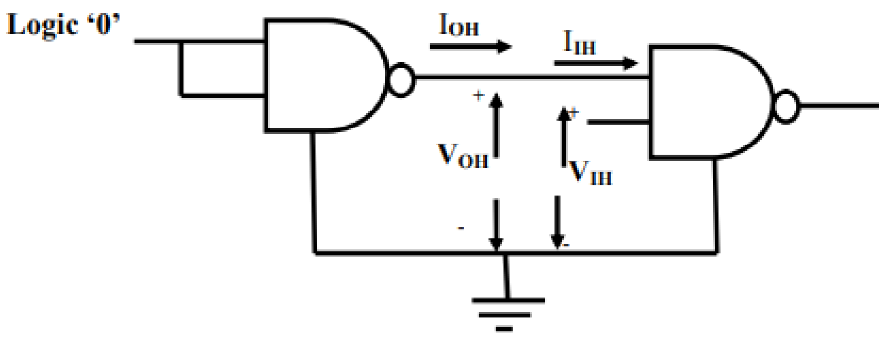


Figure 4.2 High level Input and output current and voltage requirements

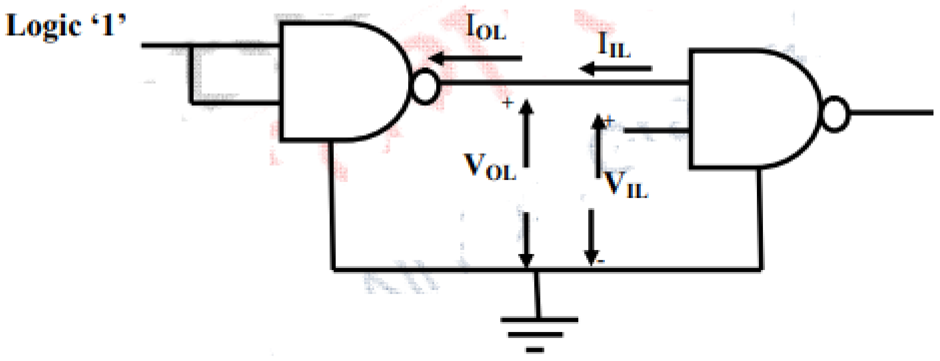


Figure 4.3: Low level Input and output current and voltage requirements

9. Supply current, I_{CC}. The supply current when the output is HIGH, LOW and in the high-impedance state is respectively designated as I_{CCH}, I_{CCL} and I_{CCZ}.

10. Fan out: It specifies the number of standard loads that the output of the gate can drive without affecting its normal operation. A standard load is usually defined as the amount of current needed by an input of another gate in the same family as shown in Figure 6. Sometimes, the term' loading' is also used instead of 'fan out'. This term is derived from the fact that the output of the gate can supply a limited amount of current above which it ceases to operate properly and is said to be overloaded. The output of the gate is usually connected to the inputs of similar gates.

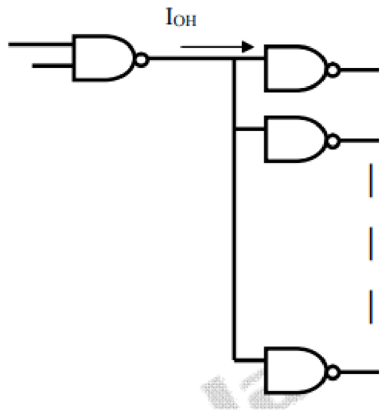


Figure 4.4: Fan out of logic gate is decided by number of loads connected to output of gate of same logic family

Each input consumes a certain amount of power from gate input so that, each additional connection adds to load the gate. These loading rules are listed for family of standard digital circuits. The rules specify the maximum amount of loading allowed for each output in the circuit. Exceeding the specified maximum load may cause a malfunction because the circuit cannot supply the power demanded from it. Thus fan out is the maximum number of inputs that can be connected to the output of the gate and is expressed by a number. The fanout capabilities of a gate must be considered while simplifying Boolean functions. One should use non inverting amplifiers or buffers to provide additional driving capabilities for heavy loads.

11. Fan in: This is the number of inputs of a logic gate. It is decided by the input current sinking capability of a logic gate as shown in Figure 7.

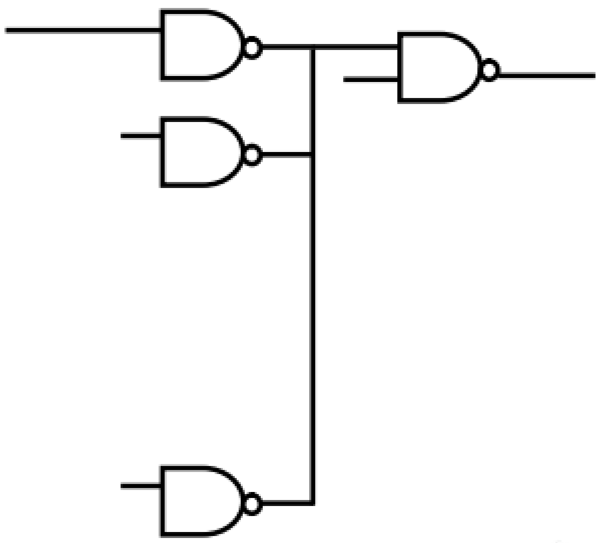


Figure 4.5: Fan in of logic gate

12. Power Dissipation: This is the power supplied required to operate the gate. It is expressed in milli-watt (mW) and represents actual power dissipated in the gate. It is the number that represents power delivered to gate from the power supply. The total power dissipated in the digital system is sum of power dissipated in each digital IC.

13. Rise time, tr: This is the time that elapses between 10% and 90 % of the final signal level when the signal makes a transition from logic LOW to logic HIGH.

14. Fall time, tf: This is the time that elapses between 90 and 10 % of the signal level when the signal makes a transition from logic LOW to logic HIGH

15. Propagation delay tp: The propagation delay is the time delay between the occurrences of change in the logical level at the input and before it is reflected at the output. It is the time delay between the specified voltage points on the input and output waveforms. Propagation delays are separately defined for LOW-to-HIGH and HIGH-to-LOW transitions at the output.

In addition, we also define enable and disable time delays that occur during transition between the high-impedance state and defined logic LOW or HIGH states.

16. Power dissipation. The power dissipation parameter for a logic family is specified in terms of power consumption per gate and is the product of supply voltage VCC and supply current I_{CC}. The supply current is taken as the average of the HIGH-level supply current I_{CCH} and the LOW-level supply current I_{CCL}.

17. Noise margin: This is the maximum noise voltage added to the input signal of digital circuit that does not cause an undesirable change in the circuit output. There are two types of noise to be considered here

a. DC noise: This is caused by a drift in the voltage levels of a signal

b. AC noise: This is caused by random pulse that may be created by other switching signals.

4.4. BIPOLAR LOGIC FAMILY:

Bipolar logic families are a class of digital logic circuits that use bipolar junction transistors (BJTs) as their primary switching elements. These families are known for their high-speed operation due to the rapid switching capabilities of BJTs. The most prominent bipolar logic families are Transistor-Transistor Logic (TTL) and Emitter-Coupled Logic (ECL). TTL is widely used in various digital systems due to its robustness, ease of use, and availability in multiple variants such as Standard TTL, Low-Power TTL, and High-Speed TTL. These variants cater to different needs, balancing speed, power consumption, and cost.

4.4.1 RTL

Resistor-Transistor Logic (RTL) is one of the earliest types of digital logic families. It uses resistors and transistors to implement logical functions such as AND, OR, and NOT gates. In an RTL circuit, resistors are used to limit the input current, and transistors are used as switching devices to perform logical operations.

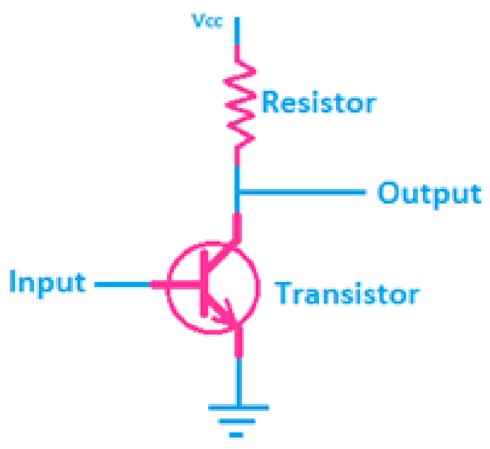


Figure 4.6 RTL

Key Characteristics of RTL:

- **Technology:** RTL circuits consist of resistors for input networks and transistors for switching.
- **Simplicity:** The design is simple, but the performance is limited due to the use of resistors.
- **Power Consumption:** RTL circuits have relatively high power consumption because resistors dissipate power continuously.
- **Switching Speed:** The switching speed of RTL is slow compared to more advanced logic families like TTL (Transistor-Transistor Logic) or CMOS (Complementary Metal-Oxide-Semiconductor) due to the resistor networks.
- **Noise Margin:** RTL circuits have a low noise margin, making them susceptible to noise and signal degradation.

Applications:

- RTL was used in the early days of digital systems, such as in the construction of early computers.
- Due to its limitations in speed and power consumption, RTL has largely been replaced by more advanced logic families like TTL and CMOS.

Overall, RTL played a significant role in the development of early digital logic systems but is rarely used in modern designs due to its inefficiency.

4.4.2 DTL

Diode-Transistor Logic (DTL) is an early type of digital logic family that uses diodes and transistors to perform logic functions like AND, OR, and NOT gates. It improves upon Resistor-Transistor Logic (RTL) by using diodes for input logic and transistors for amplification, leading to better performance in terms of speed and reliability.

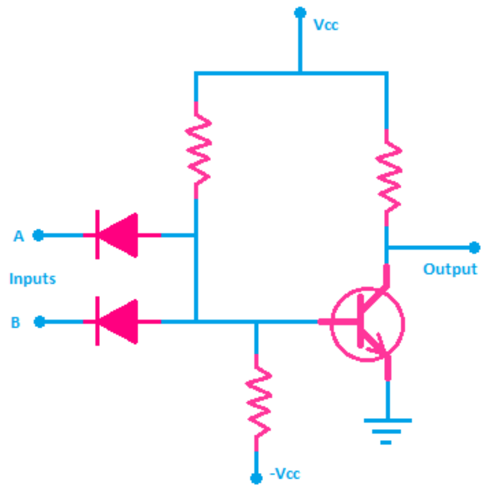


Figure 4.7 DTL

Key Characteristics of DTL:

- **Technology:** DTL circuits use diodes for logical operations (AND or OR functions) and transistors for switching and amplification.
- **Improved Speed:** DTL is faster than RTL because diodes have better performance in controlling signal flow compared to resistors. However, it's still slower than later families like Transistor-Transistor Logic (TTL).
- **Moderate Power Consumption:** DTL consumes less power than RTL but still more than CMOS logic. Power consumption mainly depends on the transistor switching and the number of diodes in the circuit.
- **Noise Immunity:** DTL has better noise immunity compared to RTL due to the use of diodes, but it still lags behind other logic families like TTL and CMOS.
- **Switching Speed:** While faster than RTL, DTL's switching speed is relatively slower than TTL and Emitter-Coupled Logic (ECL).

Circuit Operation:

- Diodes in DTL circuits are used to implement AND or OR logic gates. For example, in an AND gate, diodes are arranged in series so that current flows only when all inputs are high (logic 1).
- Transistors amplify the signals and act as inverters (NOT gates) to produce the required logic output.

Applications of DTL:

- Early Digital Circuits: DTL was widely used in the 1960s for early computing systems, control systems, and digital logic applications.
- Computers and Electronics: Used in early computers, DTL circuits played a crucial role before the development of TTL.

Diode-Transistor Logic represents an important step in the evolution of digital logic families, bridging the gap between the simpler RTL and more efficient logic families like TTL. However, it has been largely replaced by TTL and CMOS due to better performance and efficiency.

4.4.3 TTL

Transistor-Transistor Logic (TTL) is a widely used logic family in digital circuits. It utilizes transistors for both the logic operations and amplification, which results in faster switching speeds and improved efficiency compared to earlier logic families like Resistor-Transistor Logic (RTL) and Diode-Transistor Logic (DTL). TTL became the dominant technology in digital circuit design for many years before the rise of CMOS technology.

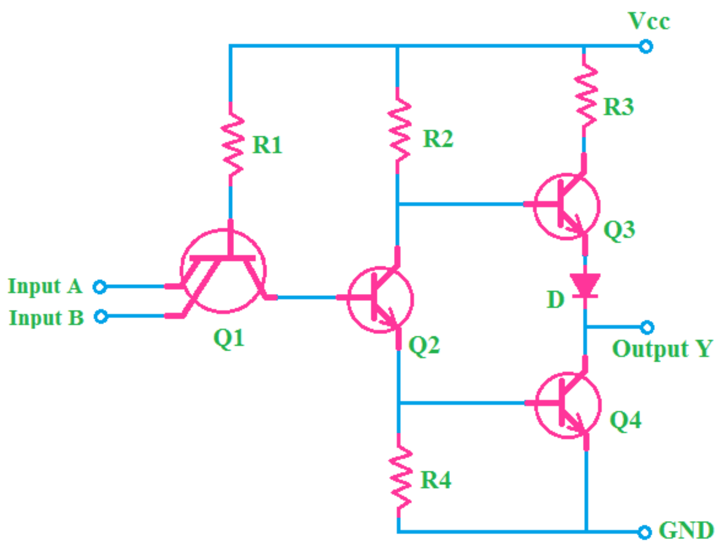


Figure 4.8 TTL

Key Characteristics of TTL:

- **Technology:** TTL circuits are built using bipolar junction transistors (BJTs). Each transistor performs a dual role in both logic operation and amplification, which distinguishes it from earlier families where separate components handled these tasks.
- **High Speed:** TTL circuits are known for their high-speed operation, especially when compared to RTL and DTL. This is due to the efficiency of transistor-based logic, which allows faster signal processing and switching.
- **Moderate Power Consumption:** While TTL offers faster switching speeds, it consumes more power than CMOS, making it less suitable for low-power applications. However, its power consumption is still lower than that of ECL.
- **Noise Immunity:** TTL has moderate noise immunity, which allows it to handle noise and signal degradation better than earlier logic families like DTL. However, it is not as noise-immune as CMOS circuits.
- **Fan-out Capability:** TTL has a relatively high fan-out, meaning that a single output can drive several inputs of other gates without significant performance degradation. Standard TTL gates can typically drive 10 other gates.

Circuit Operation

In TTL circuits, transistors are used to create logic gates (AND, OR, NOT) and perform **amplification:**

- Transistor-Based Gates: Logic operations are handled by BJTs, which act as switches that turn on or off based on the input conditions.
- Inverters: The transistors in TTL are arranged to invert signals as needed, allowing for flexible logic designs.

Subtypes of TTL:

- **Standard TTL:** The basic version of TTL, providing moderate speed and power efficiency.
- **Low Power TTL (LTTL):** Designed to reduce power consumption, but with slightly slower performance.
- **High-Speed TTL (HSTTL):** Optimized for faster operation but at the cost of higher power consumption.
- **Schottky TTL:** Utilizes Schottky diodes to reduce switching times, further improving speed while controlling power dissipation.

Applications of TTL

- **Microprocessors:** Early microprocessors used TTL due to its balance of speed and power efficiency.
- **Control Systems:** TTL is widely used in industrial and automation control systems where moderate speed and reliability are required.
- **Computing Systems:** TTL circuits were a fundamental component in early computers and digital logic systems.

4.4.4 ECL

Emitter-Coupled Logic (ECL) is one of the fastest logic families used in digital circuits, characterized by its use of differential amplifiers instead of switching transistors. Unlike other logic families, ECL avoids the saturation state in transistors, leading to faster switching times. It is primarily used in high-speed applications, where speed is more critical than power efficiency.

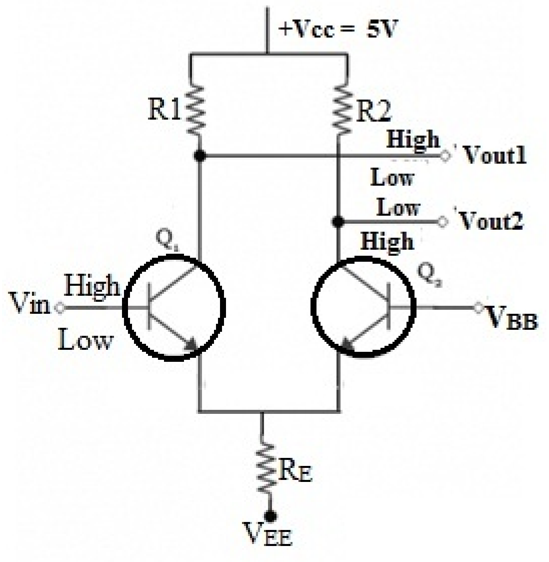


Figure 4.9 ECL

Key Characteristics of ECL

- Technology: ECL circuits use bipolar junction transistors (BJTs) in a differential amplifier configuration. The transistors are operated in the active region, avoiding saturation, which significantly increases switching speed.
- High Speed: ECL is the fastest logic family, as it operates with transistors in the active region. The elimination of the transistor's saturation and cut-off stages reduces delay and enables switching in the nanosecond range, making it ideal for high-speed applications.
- High Power Consumption: One major drawback of ECL is its high power consumption. Since current flows continuously through the transistors, ECL consumes significantly more power compared to families like CMOS or TTL.
- **Low Noise Immunity:** ECL has relatively low noise immunity compared to other logic families. This is due to the small voltage swing (around 0.8V) used in ECL circuits, which can make them more susceptible to noise. However, their differential nature can provide some inherent noise resistance.
- **Negative Power Supply:** ECL circuits typically use a negative power supply (often -5.2V), unlike most other logic families, which use a positive supply. This negative voltage supply helps maintain the transistors in the active region for faster switching.

Circuit Operation

In ECL, the transistors are arranged as differential pairs, where the output is determined by which of the two transistors is conducting. Since the transistors do not enter saturation, the switching is fast, and the output levels are stable.

- **Emitter Follower Configuration:** The output is taken from the emitter of one of the transistors, which allows the circuit to drive a low-impedance load.
- **Differential Amplifier:** The basic ECL gate consists of a differential pair of transistors, where the inputs control the current through the two transistors.

Applications of ECL

- **Supercomputing:** ECL is used in supercomputers where processing speed is the highest priority.
- **High-Speed Data Communication:** It is employed in high-speed data links, such as in fiber optic communication systems, where rapid data transmission is essential.
- **Telecommunication Systems:** ECL is utilized in telecom switches and routers, particularly in the backbone infrastructure where speed is critical.
- **Radar Systems:** ECL is used in radar systems due to its ability to handle extremely high frequencies.

- **Military Applications:** Many high-speed military applications, including missile guidance systems, use ECL for its fast response times.

4.5 UNIPOLAR LOGIC FAMILY:

Unipolar logic families are digital circuits that use metal-oxide-semiconductor field-effect transistors (MOSFETs) as their primary switching components. The most prominent unipolar logic family is Complementary Metal-Oxide-Semiconductor (CMOS), which employs both n-type and p-type MOSFETs to create logic gates. CMOS technology is highly favored for its low power consumption, high noise immunity, and scalability, making it ideal for use in battery-operated devices and large-scale integrated circuits. Due to these advantages, CMOS has become the dominant technology in modern digital electronics, including microprocessors, memory devices, and application-specific integrated circuits (ASICs).

4.4.1 MOS

In the IC design, the basic and most essential component is the transistor. So MOSFET is one kind of transistor used in many applications. The formation of this transistor can be done like a sandwich by including a semiconductor layer, generally a wafer, a slice from a single crystal of silicon; a layer of silicon dioxide & a metal layer. These layers allow the transistors to be formed within the semiconductor material. A good insulator like Sio2 has a thin layer with a hundred molecules thickness.

The transistors which we use polycrystalline silicon (poly) instead of metal for their gate sections. The Polysilicon gate of FET can be replaced almost using metal gates in large scale ICs. Sometimes, both polysilicon & metal FET’s are referred to as IGFET’s which means insulated gate FETs, because the Sio2 below the gate is an insulator.

4.4.1 CMOS

The term CMOS stands for “Complementary Metal Oxide Semiconductor”. This is one of the most popular technology in the computer chip design industry and it is broadly used today to form integrated circuits in numerous and varied applications. Today’s computer memories, CPUs, and cell phones make use of this technology due to several key advantages. This technology makes use of both P channel and N channel semiconductor devices. One of the most popular MOSFET technologies available today is the Complementary MOS or CMOS technology. This is the dominant semiconductor technology for microprocessors, microcontroller chips, memories like RAM, ROM, EEPROM and application-specific integrated circuits (ASICs).

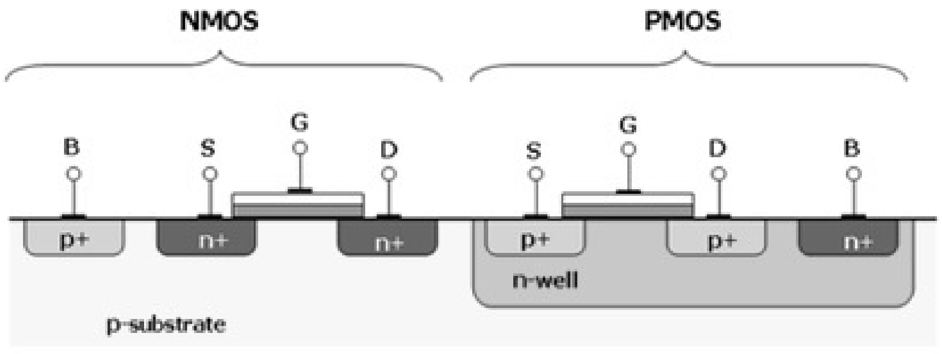


Figure 4.10 CMOS

CMOS Applications:

Complementary MOS processes were widely implemented and have fundamentally replaced NMOS and bipolar processes for nearly all digital logic applications. CMOS technology has been used for the following digital IC designs.

- Computer memories, CPUs
- Microprocessor designs
- Flash memory chip designing
- Used to design application-specific integrated circuits (ASICs)

4.4.2 PMOS

P- channel MOSFET consists of P-type Source and Drain diffused on an N-type substrate. The majority of carriers are holes. When a high voltage is applied to the gate, the PMOS will not conduct. When a low voltage is applied to the gate, the PMOS will conduct. The PMOS devices are more immune to noise than NMOS devices.

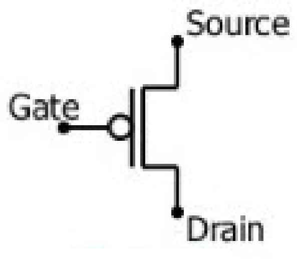


Figure 4.11: PMOS

4.4.3 NMOS

NMOS is built on a p-type substrate with n-type source and drain diffused on it. In NMOS, the majority of carriers are electrons. When a high voltage is applied to the gate, the NMOS will conduct. Similarly, when a low voltage is applied to the gate, NMOS will not conduct. NMOS is considered to be faster than PMOS, since the carriers in NMOS, which are electrons, travel twice as fast as the holes.

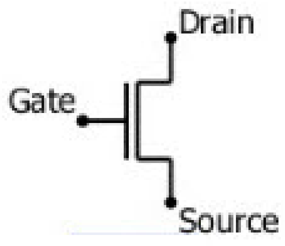


Figure 4.12 NMOS

4.6 SUMMARY:

This chapter explores the various types of logic families used in digital electronics, focusing on their classification into bipolar (e.g., TTL, ECL) and unipolar (e.g., CMOS, NMOS, PMOS) categories. It delves into their key characteristics, such as logic levels, noise margins, propagation delays, power consumption, and fan-out capabilities, which influence the performance and application of these families. The chapter highlights the differences between these families, offering insights into their advantages, limitations, and suitable use cases. Additionally, it addresses the challenges and methods of interfacing different logic families, emphasizing the importance of proper selection based on application needs. Emerging technologies and recent developments in logic families are also discussed, providing a comprehensive understanding of their evolution and current trends.

4.7 TECHNICAL TERMS:

Flip Flop, latch, sequential circuit, timing diagram, characteristic equation.

4.8 SELF ASSESSMENT QUESTIONS:

Essay questions:

1) Discuss the operational principles, advantages, and disadvantages of TTL and CMOS logic families
2) Explain the Importance of Key Characteristics in Digital Logic Families
3) Analyze how advancements in semiconductor technology have influenced the evolution of digital logic families
4) What are the common issues encountered when interfacing TTL, CMOS, and ECL logic families?

Short Questions:

1) What are the two main categories of digital logic families?
2) Define the term 'noise margin' in the context of digital logic families.
3) What is the primary advantage of CMOS logic over TTL logic?
4) Name one example of a bipolar logic family and one example of a unipolar logic family.
5) Why is propagation delay an important factor in the performance of digital circuits?

4.9 FURTHER READINGS:

1) M.Morris Mano, “Computer System Architecture”, 3rd Edition, Pearson Education (2008).
2) V. Rajaraman, T. Radha Krishnan, “Computer Organization and Architecture”, PHI.
3) BehroozParhami, “Computer Architecture”, Oxford (2007).
4) ISRD group, “Computer Organization”, ace series, TMH (2007).
5) William Stallings, “Computer Organization and Architecture-Designing for Performance”, Pearson Education (2005).

Dr. U. Surya Kameswari

LESSON-5
FLIP FLOPS

OBJECTIVES:

After going through this lesson, you will be able to

- Understand the Fundamental Concepts
- Explore Types of Flip-Flops
- Analyze Characteristic Equations
- Examine Timing Diagrams
- Application in Sequential Logic

STRUCTURE OF THE LESSION:

5.1 INTRODUCTION:

The combinational circuit does not use any memory. Hence the previous state of input does not have any effect on the present state of the circuit. But sequential circuit has memory so output can vary based on input. This type of circuits uses previous input, output, clock and a memory element.

Flip-flops are fundamental building blocks in digital electronics, serving as essential components for data storage and processing in various electronic devices. These bistable devices can exist in one of two stable states, making them ideal for holding binary information. The ability of flip-flops to maintain their state until explicitly changed allows them to function as memory elements, which are critical in sequential logic circuits.

At the core of flip-flop operation is the concept of triggering. Flip-flops are typically controlled by clock signals, which synchronize the state changes in digital systems. This synchronization is crucial in ensuring that data is processed and stored at the right times, enabling the reliable functioning of complex systems. The clock signal provides a uniform timing mechanism, allowing multiple flip-flops in a circuit to work in harmony.

In addition to serving as memory elements, flip-flops play a pivotal role in the design of state machines. These state machines are used in various applications, from simple control systems to complex microprocessors. By enabling the representation of different states within a system, flip-flops facilitate the implementation of algorithms and processes that require sequential logic. This capability is vital for tasks such as counting, timing, and data handling.

Another important aspect of flip-flops is their role in implementing registers and memory units. Registers are groups of flip-flops that work together to store multi-bit data. This capability is crucial in microcontrollers and processors, where data needs to be temporarily held for processing. The organization of flip-flops into registers allows for efficient data manipulation and retrieval, which is fundamental to computing operations.

The versatility of flip-flops extends beyond basic memory functions; they can also be employed in various applications, including frequency dividers, digital counters, and shift registers. Their ability to be combined in different configurations enables engineers to create complex digital systems tailored to specific requirements. This adaptability makes flip-flops indispensable in the realm of digital design and engineering.

5.2. FLIP FLOP Vs LATCH:

Latches and flip-flops are the basic elements for storing information. One latch or flip-flop can store one bit of information. The main difference between latches and flip-flops is that for latches, their outputs are constantly affected by their inputs as long as the enable signal is asserted. In other words, when they are enabled, their content changes immediately when their inputs change. Flip-flops, on the other hand, have their content change only either at the rising or falling edge of the enable signal. This enable signal is usually the controlling clock signal. After the rising or falling edge of the clock, the flip-flop content remains constant even if the input changes. There are basically four main types of latches and flip-flops: SR, D, JK, and T. The major differences in these flip-flop types are the number of inputs they have and how they change state. For each type, there are also different variations that enhance their operations.

Latches are level-sensitive devices, meaning they respond to input signals as long as the control signal (enable or gate) is active. This allows them to continuously monitor their inputs and change state whenever the input conditions are met. In contrast, flip-flops are edge-sensitive devices; they change state only at specific moments dictated by a clock signal, typically on the rising or falling edge. This makes flip-flops more suited for synchronous applications.

Flip flop = latch + clock signal

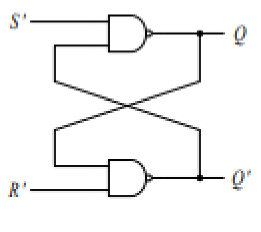
Latches tend to be simpler in terms of design and implementation compared to flip-flops. They can be constructed using fewer gates and require less power. Flip-flops, while more complex, provide better control over data flow and timing, making them more suitable for advanced digital systems that require reliability and precision.

The Latch's Characteristics:

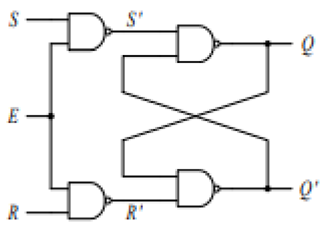
- Latches can simultaneously store one or more bits of binary data.
- They work by "latching" onto a specific binary value and holding it there until the input changes.
- Various gates can be used to construct latches, including NOR and NAND.
- There are several types of latches: SR Latch, D Latch, JK Latch, etc

The Flip-Flops characteristics:

- Flip-flops are binary devices with two stable states: high and low.
- Numerous logic gates, such as NAND, NOR, and JK, can construct them.
- Flip-flops are used in digital electronic devices like computers, calculators, and cell phones and are crucial in digital circuits.
- The performance of flip-flops affects the general functionality and speed of digital circuits.
- Flip-flops are used in various sectors, such as signal processing, control systems, and telecommunication.
- Flip-flops can switch between positive (rising edge) and negative (falling edge) states in response to a clock signal.
- In digital circuits, flip-flops are devices used for storing binary data, and there are four main types.



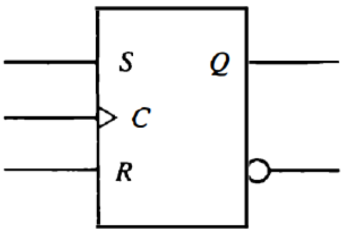
SR Latch



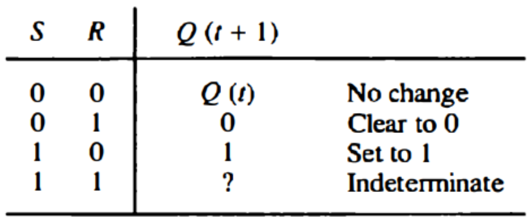
SR Flip Flop

5.3 SR FLIP-FLOP:

S-R flip-flop is a set-reset flip-flop. It is used in MP3 players, Home theatres, Portable audio docks, and etc. S-R can be built with NAND gate or with NOR gate. Either of them will have the input and output complemented to each other.



(a) Graphic symbol



S	R	$Q(t+1)$	
0	0	$Q(t)$	No change
0	1	0	Clear to 0
1	0	1	Set to 1
1	1	?	Indeterminate

(b) Characteristic table

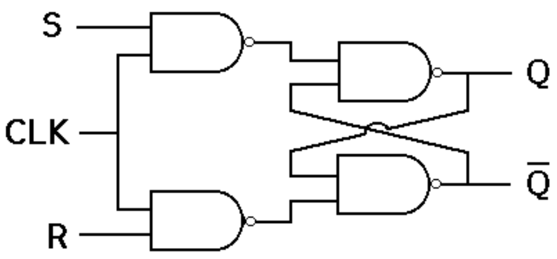


Figure 5.1 SR Flip Flop

5.3.1 Operation:

The operation of the SR flip-flop is as follows. If there is no signal at the clock input C, the output of the circuit cannot change irrespective of the values at inputs S and R. Only when the clock signal changes from 0 to 1 can the output be affected according to the values in inputs S and R. If S = 1 and R = 0 when C changes from 0 to 1, output Q is set to 5. If S = 0 and R = 1 when changes from 0 to 1, output Q is cleared to 0. If both S and R are 0 during the clock transition, the output does not change. When both 5 and R are equal to 1, the output is unpredictable and may go to either 0 or 1, depending on internal timing delays that occur within the circuit.

5.3.2 Timing Characteristics:

The characteristic table shown in Fig. 5.2(b) summarizes the operation of the SR flip-flop in tabular form. The S and R columns give the binary values of the two inputs. Q(t) is the binary state of the Q output at a given time (referred to as present state). Q(t + 1) is the binary state of the Q output after the occurrence of a clock transition (referred to as next state). If S = R = 0, a clock transition produces no change of state [i.e., Q(t + 1) = Q(t)] . If S = 0 and R = 1, the flip-flop goes to the 0 (clear) state. If S = 1 and R = 0, the flip-flop goes to the 1 (set) state. The SR flip-flop should not be pulsed when S = R = 1 since it produces an indeterminate next state. This indeterminate condition makes the SR flip-flop difficult to manage and therefore it is seldom used in practice.

Characteristic equation is ***Q (t+1) = S + (Q (t) • R')***

5.3.3 Applications:

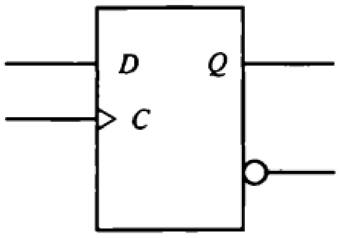
SR flip-flops are commonly used in applications that require basic memory storage, such as data latches, and in circuits where control signals are needed to manage operations. They can also be found in state machines, counters, and other sequential logic devices, providing a simple means to implement basic state changes.

5.3.4. Limitations

While the SR flip-flop is straightforward and useful, it has limitations. The indeterminate state that occurs when both inputs are high is a significant drawback in many designs. To address this issue, more advanced flip-flops, like JK or D flip-flops, have been developed, which provide a more robust approach to state management and eliminate the invalid state condition.

5.4 D FLIP FLOP:

The D (data) flip-flop is a slight modification of the SR flip-flop. An SR flip-flop is converted to a D flip-flop by inserting an inverter between S and R and assigning the symbol D to the single input. The D input is sampled during the occurrence of a clock transition from 0 to 5. If D = 1, the output of the flip-flop goes to the 1 state, but if D = 0, the output of the flip-flop goes to the 0 state



(a) Graphic symbol

D	$Q(t+1)$	
0	0	Clear to 0
1	1	Set to 1

(b) Characteristic table

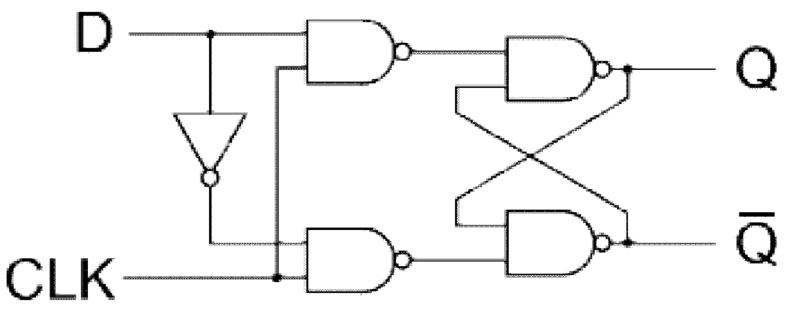


Figure 5.2 D Flip Flop

5.4.1 Operation:

The D flip-flop operates on the principle of edge-triggering. When the clock signal transitions-typically on the rising edge (from low to high)-the current value of the D input is captured and transferred to the Q output. This means that Q takes on the value of D at that specific moment. If the D input is high (1) at the clock edge, Q will be set to high; if D is low (0), Q will be reset to low. Between clock cycles, the output remains stable, effectively "holding" the last value of D.

5.4.2 Timing Characteristics:

From the characteristic table we note that the next state Q(t + 1) is determined from the D input. The relationship can be expressed by a characteristic equation:

$$Q(t + 1) = D$$

This means that the Q output of the flip-flop receives its value from the D input every time that the clock signal goes through a transition from 0 to 5.

no input condition exists that will leave the state of the D flip-flop unchanged. Although a D flip-flop has the advantage of having only one input (excluding C), it has the disadvantage that its characteristic table does not have a "no change" condition Q(t + 1) = Q(t).

5.4.3 Applications

D flip-flops are commonly used in various applications, including data storage, shift registers, and memory devices. They play a significant role in creating digital systems such as microcontrollers and digital signal processors, where data needs to be captured and processed in a controlled manner. The D flip-flop’s ability to delay data until a specific clock event also makes it useful in timing applications.

5.4.4 Advantages

One of the primary advantages of the D flip-flop is its simplicity and reliability. It eliminates the indeterminate states associated with other flip-flop types (like the SR flip-flop), providing a clear and unambiguous output based on the input at the clock edge. This feature enhances its robustness in digital design, making it a preferred choice in many applications.

5.5 JK FLIP FLOP:

A JK flip-flop is a refinement of the SR flip-flop in that the indeterminate condition of the SR type is defined in the JK type. Inputs J and K behave like inputs S and R to set and clear the flip-flop, respectively. When inputs J and K are both equal to 1, a clock transition switches the outputs of the flip-flop to their complement state.

The graphic symbol and characteristic table of the JK flip-flop are shown in Figure 5.3.

The J input is equivalent to the S (set) input of the SR flip-flop, and the K input is equivalent to the R (clear) input.

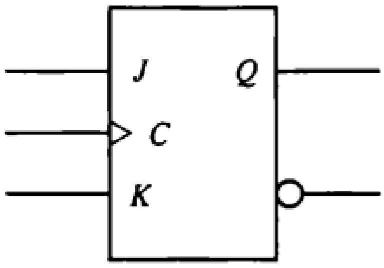
5.5.1 Operation

The JK flip-flop operates based on the values of its J and K inputs at the moment of a clock edge, typically the rising edge. The behavior of the JK flip-flop is defined by the following rules:

- If J is high (1) and K is low (0), the flip-flop sets Q to high (1).
- If J is low (0) and K is high (1), it resets Q to low (0).

- If both J and K are low (0), the output Q retains its previous state.
- If both J and K are high (1), the flip-flop toggles its state (if Q was 0, it becomes 1, and vice versa).

This toggle capability is a significant advantage of the JK flip-flop, making it ideal for counting applications.



(a) Graphic symbol

J	K	$Q(t+1)$	
0	0	$Q(t)$	No change
0	1	0	Clear to 0
1	0	1	Set to 1
1	1	$Q'(t)$	Complement

(b) Characteristic table

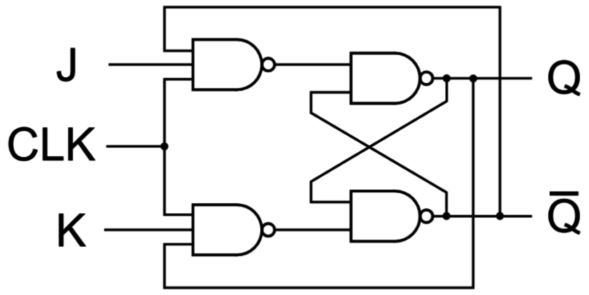


Figure 5.3 JK Flip Flop

5.5.2 Timing Characteristics

The JK flip-flop is an edge-triggered device, meaning it only changes its output state at the moment of a clock edge. This feature allows for precise timing in synchronous circuits, where multiple JK flip-flops can be coordinated to operate together. The edge-triggered behavior ensures that state changes occur predictably, making it suitable for complex digital systems.

The characteristic function of JK Flip Flop is$Q\ (t+1) = J + (Q\ (t) \cdot K')$

5.5.3 Applications

The flexibility of the JK flip-flop makes it ideal for a variety of applications, including:

- **Counters**: The toggle feature allows JK flip-flops to function as binary counters, where the state changes with each clock pulse.
- **Shift Registers**: They can be used in shift register designs to store and manipulate data serially.
- **State Machines**: JK flip-flops are often used in finite state machines to represent and transition between different states.

5.5.4 Advantages

One of the primary advantages of the JK flip-flop is its ability to toggle, which eliminates the ambiguous state present in the SR flip-flop. This feature makes it more versatile for applications where changing states is required. Additionally, its robust performance and predictable behavior in synchronous designs enhance its reliability in various electronic systems.

5.6 T FLIP FLOP:

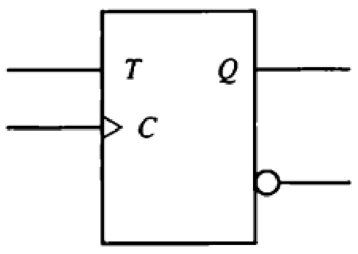
The T (Toggle) flip-flop is a type of bistable multivibrator widely used in digital electronics. It is a simplified version of the JK flip-flop, designed specifically for toggling its output state based on a single input. The T flip-flop has one input, labeled T (Toggle), and two outputs: Q and $\overline{Q}$ (the complement of Q). Its straightforward operation makes it particularly useful in applications such as counters and frequency dividers.

5.6.1 Operation

The T flip-flop changes its output state based on the value of the T input at the moment of a clock edge, typically the rising edge. The key characteristic of the T flip-flop is that:

- If T is high (1), the flip-flop toggles its output: if Q is currently 0, it becomes 1, and if Q is currently 1, it becomes 0.
- If T is low (0), the output Q remains in its current state, effectively holding the value.

This toggling behavior is what distinguishes the T flip-flop from other types of flip-flops.



(a) Graphic symbol

T	$Q(t+1)$	
0	$Q(t)$	No change
1	$Q'(t)$	Complement

(b) Characteristic table

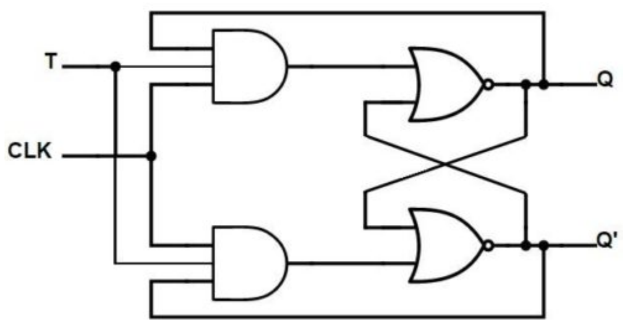


Figure 5.4 T Flip Flop

5.6.2 Timing Characteristics

The T flip-flop therefore has only two conditions. When T = 0 (J = K = 0) a clock transition does not change the state of the flip-flop. When T = 1 (J = K = 1) a clock transition complements the state of the flip-flop. These conditions can be expressed by a characteristic equation:

$$Q(t+1) = Q(t) \oplus T$$

5.6.3 Applications

The T flip-flop is particularly useful in various applications, including:

- **Counters**: It is commonly used in binary counters, where the T flip-flop toggles its state with each clock pulse, effectively counting the number of pulses received.
- **Frequency Dividers**: T flip-flops can be used to create frequency dividers, reducing the frequency of an input clock signal by half for each T flip-flop in the chain.
- **State Machines**: They are often employed in simple state machine designs where toggling between states is necessary.

5.6.4 Advantages

The simplicity of the T flip-flop is one of its main advantages. With only one input, it reduces design complexity compared to other flip-flops like JK and SR. This makes it easier to implement in various digital systems. Additionally, the toggling feature allows for efficient counting and frequency division, making it a versatile choice in many applications.

5.7 SUMMARY:

This chapter focuses on flip-flops, the fundamental building blocks of sequential logic circuits in digital electronics. It covers the various types of flip-flops-SR, D, JK, and T-highlighting their unique operational characteristics, applications, and characteristic equations. The chapter emphasizes the importance of timing diagrams in illustrating the relationship between inputs and outputs and explores the roles flip-flops play in data storage, counting, and state machine design. Understanding flip-flops is crucial for mastering sequential logic and its applications in modern electronic systems.

5.8 TECHNICAL TERMS:

Flip Flop, latch, sequential circuit, timing diagram, characteristic equation.

5.9 SELF ASSESSMENT QUESTIONS:

Essay questions:

1) Explain the Role of Flip-Flops in Digital Electronics
2) Compare and Contrast Different Types of Flip-Flops

3) Analyze the Characteristic Equations of Flip-Flops
4) Describe the Implementation of Flip-Flops in Integrated Circuits
5) Explain various types of flip flops with neat diagrams.

Short Questions:

1) What is the primary function of a flip-flop in digital electronics?
2) How does a D flip-flop differ from an SR flip-flop?
3) What is the characteristic equation for a JK flip-flop?
4) Describe the toggle behavior of a T flip-flop.
5) In what applications are flip-flops commonly used?

5.10 FURTHER READINGS:

1) M.Morris Mano, “Computer System Architecture”, 3rd Edition, Pearson Education (2008).
2) V. Rajaraman, T. Radha Krishnan, “Computer Organization and Architecture”, PHI
3) Behrooz Parhami, “Computer Architecture”, Oxford (2007).
4) ISRD group, “Computer Organization”, ace series, TMH (2007).
5) William Stallings, “Computer Organization and Architecture-Designing for Performance”, Pearson Education (2005).

Dr. Neelima Guntupalli

LESSON-6
REGISTERS AND COUNTERS

OBJECTIVES:

After going through this lesson, you will be able to

- Understand the role and importance of sequential circuits
- Explain the purpose and operation of Registers
- Know what are the various types of registers
- Discuss the concept and applications of Shift Registers
- Explore the function and implementation of Binary Counters and Memory Units.

STRUCTURE OF THE LESSION:

6.1 SEQUENTIAL CIRCUITS:

Sequential circuits are a fundamental component in computer organization, characterized by their ability to store information and operate based on both current inputs and past states. Unlike combinational circuits, which produce outputs solely based on the current inputs, sequential circuits incorporate memory elements, such as flip-flops or latches, to retain state information. This memory aspect allows sequential circuits to implement complex functions like counting, storing data, and managing state transitions, which are essential in creating finite state machines (FSMs) and various timing-dependent operations within a computer system.

These circuits play a crucial role in various components of a computer, including registers, counters, and control units. Registers use sequential circuits to temporarily hold data that the processor needs to access quickly, while counters help keep track of occurrences or operations within the system. The control unit, which orchestrates the operation of the CPU by directing the flow of data and the execution of instructions, relies heavily on sequential circuits to ensure accurate sequencing and timing of operations. By leveraging the ability to store and manipulate state information, sequential circuits enable computers to perform a wide array of tasks efficiently and reliably.

Table 6.1 Differences between Combinational and sequential circuits

Combinational	Sequential
In this output depends only upon present input.	In this output depends upon present as well as past input.
Speed is fast.	Speed is slow.
It is designed easy.	It is designed tough as compared to combinational circuits.
There is no feedback between input and output.	There exists a feedback path between input and output.
This is time independent.	This is time dependent.
Elementary building blocks: Logic gates	Elementary building blocks: Flip-flops
Used for arithmetic as well as boolean operations.	Mainly used for storing data.
Don't have capability to store any state.	Have capability to store any state or to retain earlier state.
These circuits do not have any memory element.	These circuits have memory element.
It is easy to use and handle.	It is not easy to use and handle

6.2 REGISTERS:

A register is basically an array or a group of flip-flops used for storing binary information. As we know that the flip-flop is the basic block of storage, one flip-flop can store one bit of information. For storing n-bit in register, it will require 'n' number of flip-flops. Registers are mainly used for storing and shifting information.

Registers finds its application in many other digital devices like serial adder, microprocessors , multipliers , dividers etc.

There is a wide variety of commercially available registers. The most basic register is one that is composed solely of flip-flops and lacks any external gates. Such a register is illustrated in Figure 6.1, which is composed of four D flip-flops. The binary data available at the four inputs is transferred into the 4-bit register, and the common clock input activates all flip-flops on the rising edge of each pulse. The binary information contained in the register can be obtained by sampling the four outputs at any given moment. Each flip-flop has a unique terminal that receives the clear input. Asynchronously, all flip-flops are reset when this input reaches 0. The clear input is beneficial for erasing the register to all O's before it is clocked. The clear input must be maintained at logic 1 during normal clocked operation. It should be noted that the D input is enabled by the clock signal, whereas the clear. input is not affected by the clock.

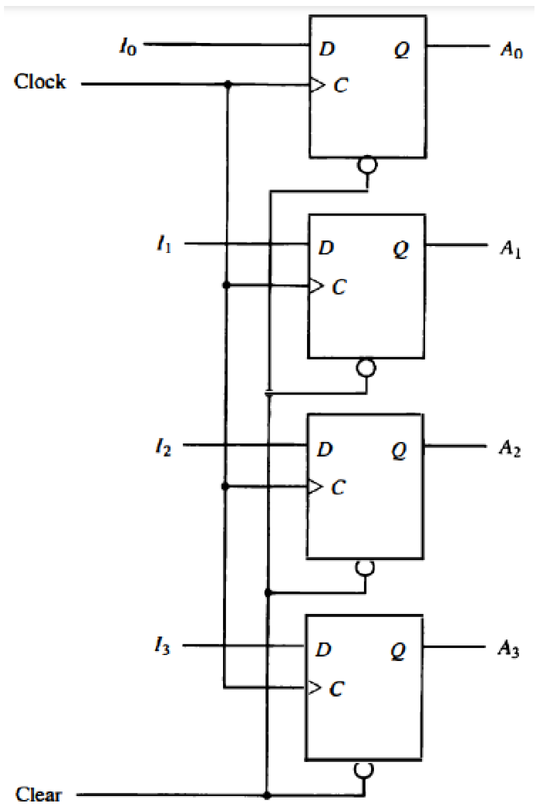


Figure 6.1: 4 Bit Register

The transfer of new information into a register is referred to as loading the register. If all the bits of the register are loaded simultaneously with a common clock pulse transition, we say that the loading is done in parallel. A clock transition applied to the C inputs of the register of Fig. 6.1 will load all four inputs 10 through 13 in parallel. In this configuration, the clock must be inhibited from the circuit if the content of the register must be left unchanged.

The majority of digital systems are equipped with a master clock generator that generates a consistent stream of clock pulses. The clock pulses are applied to all registers and flip-flops in the system. A constant rhythm is supplied to all components of the system by the master clock, which functions as a pump. In order to determine which specific clock pulse will impact a specific register, a distinct control signal must be employed.

Register with parallel load

Fig. 6.2 illustrates a 4-bit register with a load control input that is directed through gates and into the D inputs. Clock pulses are consistently transmitted to the C inputs. The power requirement is reduced by the buffer gate in the port input. The output's subsequent state is determined by the D input with each clock pulse. In order to maintain the output's current value, it is imperative to set the D input to the output's present value.

Please be advised that the clock pulses are consistently applied to the C inputs. The next pulse's acceptance of new information or preservation of the information in the register is contingent upon the load input. During a single pulse transition, all four bits are simultaneously transferred from the inputs to the register.

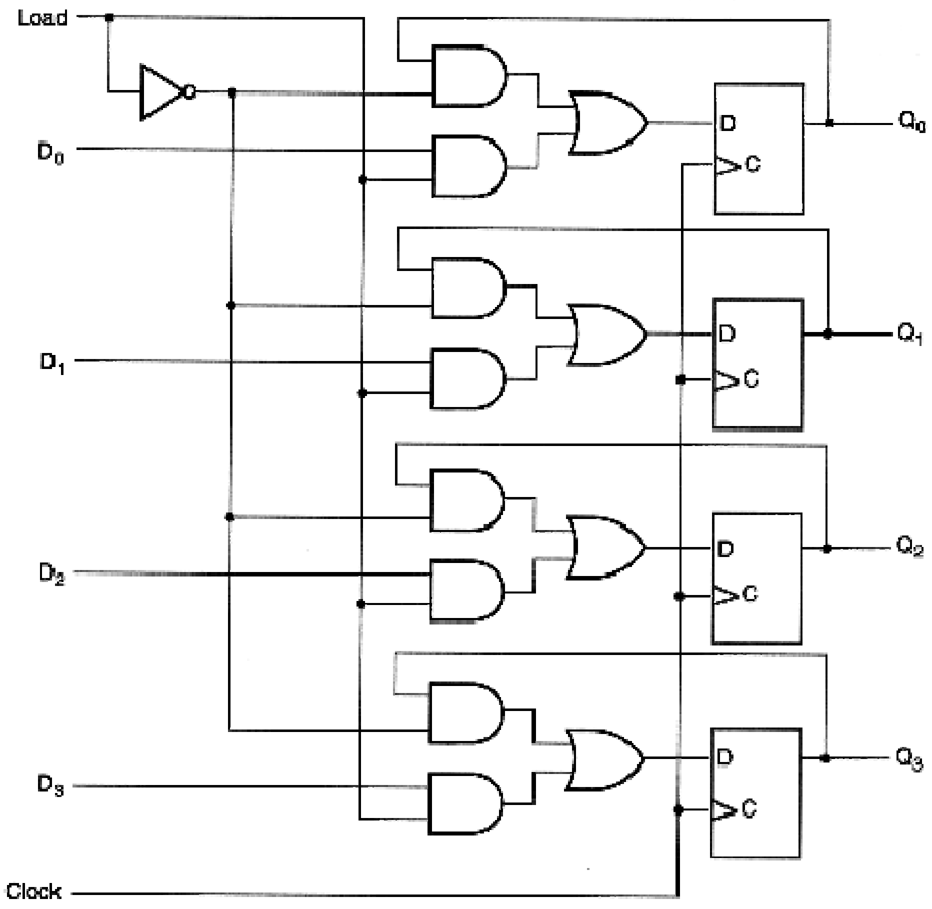


Figure 6.2: 4 bit register with parallel load

6.3 SHIFT REGISTERS:

A shift register is a register that has the capacity to modify its binary information in either direction. The logical configuration of a shift register is a cascade of flip-flops, with the output of one flip-flop connected to the input of the next. The transition from one stage to the next is initiated by common clock pulses that are received by all flip-flops. As illustrated in Figure 4-8, the most basic shift register is one that exclusively employs flip-flops. The D input of the flip-flop located to the right of the provided flip-flop is connected to its output. All flip-flops share a timepiece. What is placed in the leftmost position during the transfer is determined by the serial input. The serial output is obtained from the output of the rightmost flip-flop.

Sometimes, it is necessary to regulate the shift so that it occurs in conjunction with specific clock frequencies but not with others. The clock can be inhibited from the register's input to prevent it from shifting. The shift can be regulated by connecting the clock to the input of an

AND gate when the shift register of Fig 4-7 is employed. Subsequently, the shift can be regulated by inhibiting the clock at a second input of the AND gate. Nevertheless, it is also feasible to incorporate additional circuits to regulate the shift operation by utilizing the D inputs of the flip-flops, rather than the clock input.

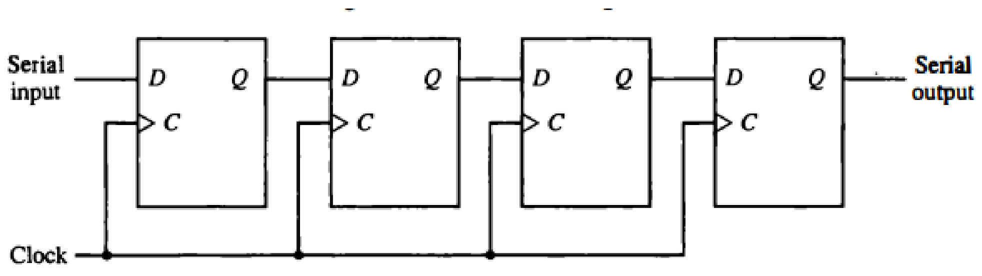


Figure 6.3: 4 bit register with parallel load

6.3.1 Types of Shift Registers

Shift registers are classified into the following types based on the how the data is stored or shifted.

1. Serial-In Serial-Out (SISO) - This type of shift register accepts data serially i.e, one bit at a time. For entering the n-bit information in the register, n flip-flops will be required and 'n' number of clock pulses. And for retrieving the data (or shifting the data out from the register) stored in the register, again 'n' number of clock pulses will be required because data is shifting serially i.e, one bit at a time (clock pulse).

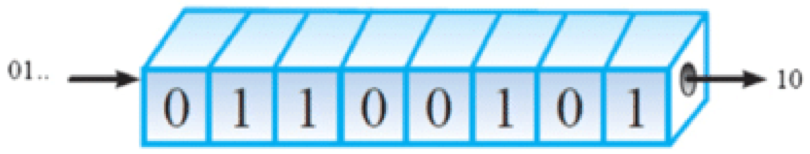


Figure 6.4 bit transferring in SISO register

2. Serial-In Parallel-Out (PISO) - In this type of flip-flop, data is entered serially into register but the output from all the flip flop are taken out in parallel (all bit at a time). Therefore for entering the n- bit data, 'n' clock pulse will be required. But for retrieving the data (or shifting the data out from the register) all bits will be available simultaneously. But shifting the data out from register can only be done when the data is completely stored i.e, each bit should appear on its respective output line.

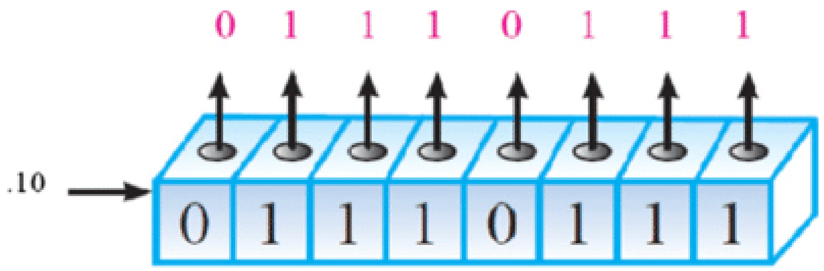


Figure 6.5 bit transferring in PISO register

3. Parallel-In Serial-Out (PISO): As the name suggests, in this type of register the bits are added/entered simultaneously into their respective flip-flop. And the shifting the data out from the register is done serially i.e., on bit by bit basis.

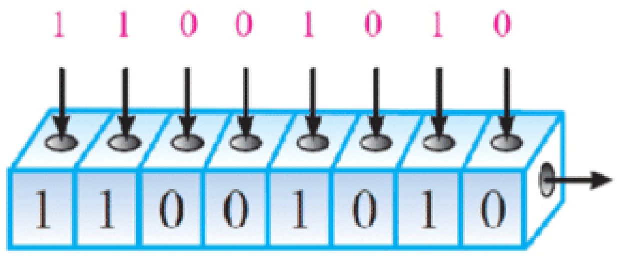


Figure 6.6 Bit transferring in SISO register

4. Parallel-In Parallel-Out (PIPO) - In this type of register, the data is entered into the register simultaneously to their respective flip-flop and the respective bits will be generated at the output because of the parallel output. Since there is not serial shifting of data, so there is no interconnection between the successive flip-flop only clock is applied simultaneously to each flip-flop.

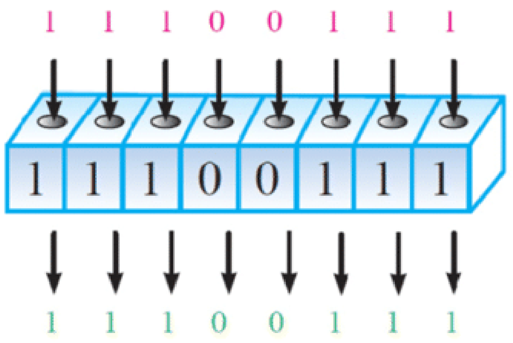


Figure 6.7 Bit transferring in SISO register

6.3.2 Bidirectional Shift Register with Parallel Load

A register capable of shifting in one direction only is called aunl directional shift register. A register that can shift in both directions is called a bidirectional shift register. Some shift registers provide the necessary input and output terminals for parallel transfer

A 4-bit bidirectional shift register with parallel load is shown in Fig. 6.8. Each stage consists of a D flip-flop and a 4 x 1 multiplexer. The two selection inputs S_1 and S_0 select one of the multiplexer data inputs for the D flip-flop.

The selection lines control the mode of operation of the register according to the function table shown in Figure 6.9.

When the mode control $S_1S_0 = 00$, data input 0 of each multiplexer is selected. This condition forms a path from the output of each flip-flop into the input of the same flip-flop. The next clock transition transfers into each flip-flop the binary value it held previously, and no change of state occurs.

When S_1S_0 = 01, the terminal marked 1 in each multiplexer has a path to the D input of the corresponding flip-flop. This causes a shift-right operation, with the serial input data transferred into flip-flop A_0 and the content of each flip-flop A_{i-1}, transferred into flip-flop A_i for i = 1, 2,3.

When S_1S_0 = 10 a shift-left operation results, with the other serial input data going into flip-flop A_3, and the content of flip-flop A_{i+1}, transferred into flip-flop A_i for i = 0, I, 6.

When S_1S_0 = 11, the binary information from each input I_0 through I_3, is transferred into the corresponding flip-flop, resulting in a parallel load operation.

Note that the way the diagram is drawn, the shift-right operation shifts the contents of the register in the down direction while the shift left operation causes the contents of the register to shift in the upward direction

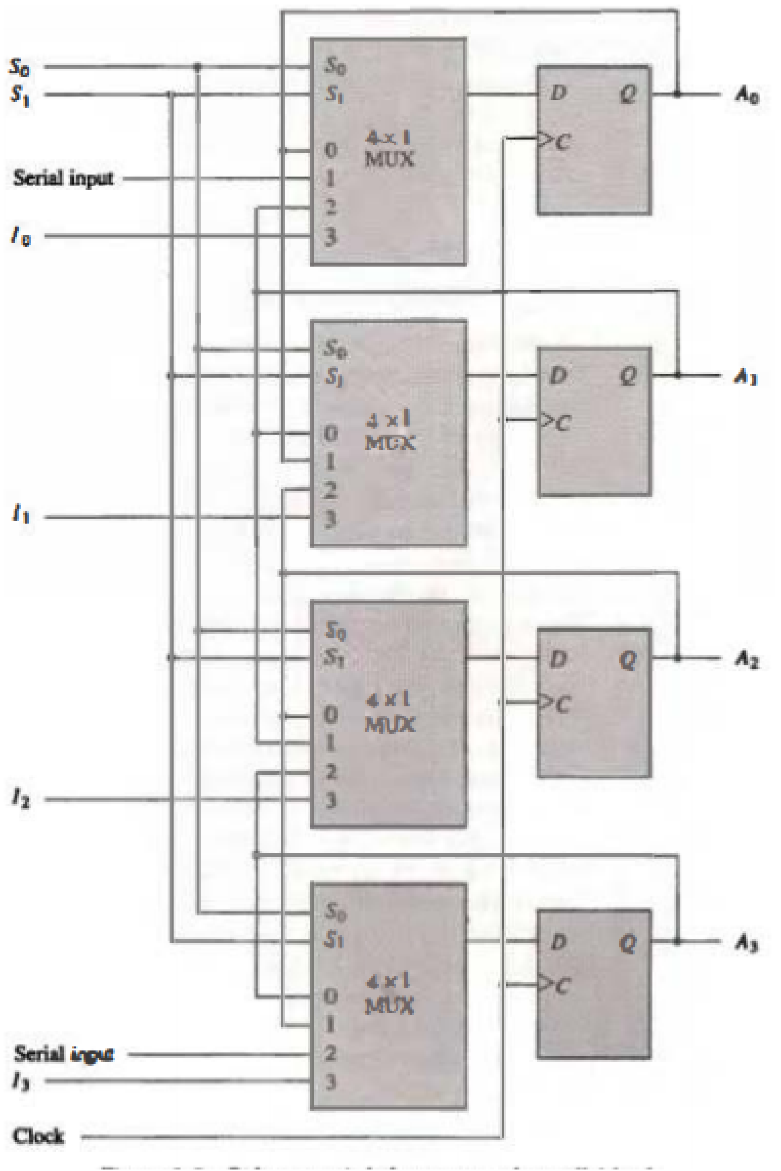


Figure 6.8: Bi directional shift register with parallel load

Mode control		
S_1	S_0	Register operation
0	0	No change
0	1	Shift right (down)
1	0	Shift left (up)
1	1	Parallel load

Figure 6.9: Function table for the bi directional shift register with parallel load

6.4 BINARY COUNTERS:

A register that goes through a predetermined sequence of states upon the application of input pulses is called a counter. The input pulses may be clock pulses or may originate from an external source. They may occur at uniform intervals of time or at random. Counters are found in almost all equipment containing digital logic. They are used for counting the number of occurrences of an event and are useful for generating timing signals to control the sequence of operations in digital computers. Of the various sequences a counter may follow, the straight binary sequence is the simplest and most straightforward. A counter that follows the binary number sequence is called a binary counter. An n-bit binary counter is a register of n flip-flops and associated gates that follows a sequence of states according to the binary count of n bits, from 0 to 2^n - 1.

A simpler alternative design procedure may be carried out from a direct inspection of the sequence of states that the register must undergo to achieve a straight binary count. Going through a sequence of binary numbers such as 0000, 0001, 0010, 0011, and so on, we note that the lower-order bit is complemented after every count and every other bit is complemented from one count to the next if and only if all its lower-order bits are equal to 1.

For example, the binary count from 0111 (7) to 1000 (8) is obtained by (a) complementing the low-order bit, (b) complementing the second-order bit because the first bit of 0111 is 1, (c) complementing the third-order bit because the first two bits of 0111 are 1's, and (d) complementing the fourth-order bit because the first three bits of 0111 are ali i's.

A counter circuit will usually employ flip-flops with complementing capabilities. Both T and JK flip-flops have this property. Remember that a JK flip-flop is complemented if both its J and K inputs are 1 and the clock goes through a positive transition. The output of the flip-flop does not change if J = K = 0.

In addition, the counter may be controlled with an enable input that turns the counter on or off without removing the clock signal from the flip-flops. Synchronous binary counters have a regular pattern, as can be seen from the 4-bit binary counter shown in Figure 6.10. The C inputs of all flip-flops receive the common clock.

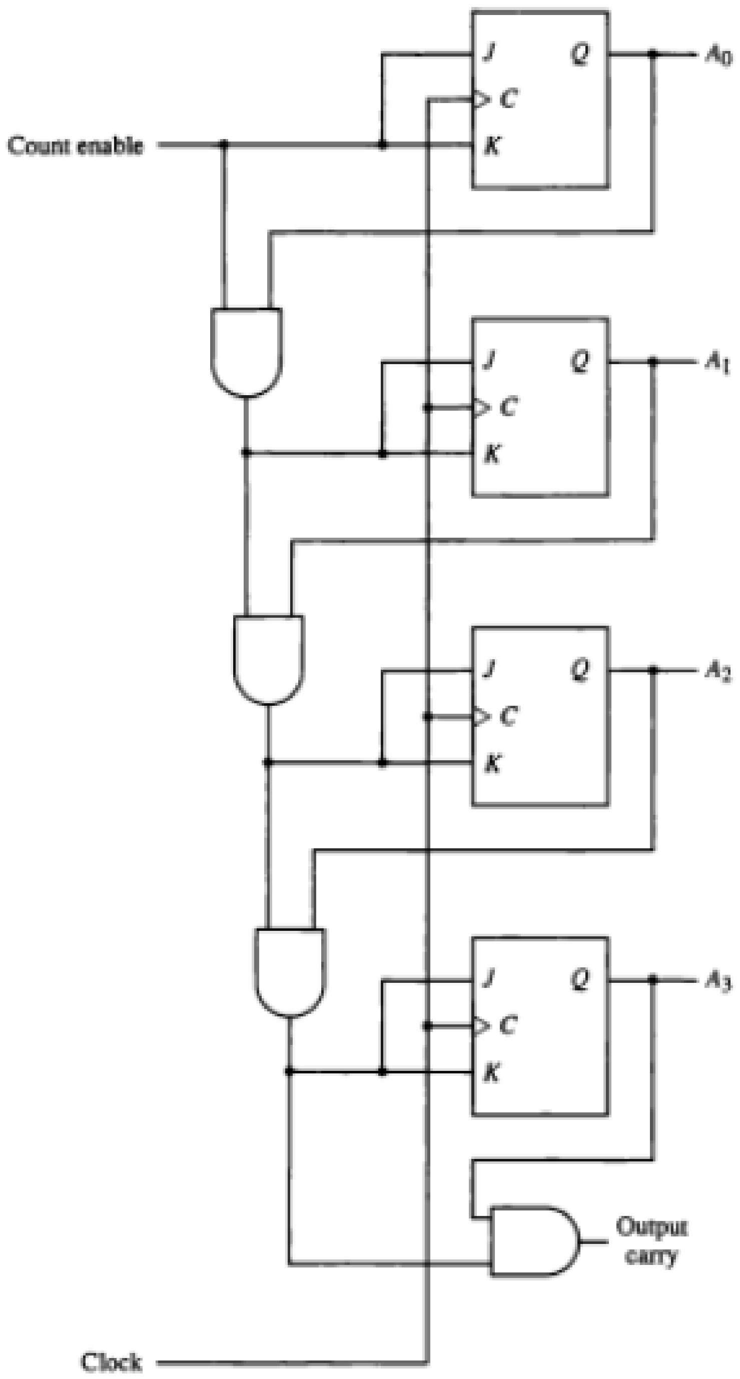


Figure 6.10: 4-bit synchronous binary counter

If the count enable is 0, all J and K inputs are maintained 4-bit-shift-registerat 0 and the output of the counter does not change. The first stage A0 is complemented when the counter is enabled and the clock goes through a positive transition. Each of the other three flip-flops are complemented when all previous least significant flip-flops are equal to 1 and the count is enabled. The chain of AND gates generate the required logic for the J and K inputs. The output carry can be used to extend the counter to more stages, with each stage having an additional flip-flop and an AND gate.

6.4.1 Binary Counter with Parallel Load

Counters employed in digital systems quite often require a parallel load capability for transferring an initial binary number prior to the count operation.

Figure 6.11 shows the logic diagram of a binary counter that has a parallel load capability and can also be cleared to 0 synchronous with the clock. When equal to 1, the clear input sets all the K inputs to 1, thus clearing all flip-flops with the next clock transition. The input load control when equal to 1, disables the count operation and causes a transfer of data from the four parallel inputs into the four flip-flops (provided that the clear input is 0). If the clear and load inputs are both 0 and the increment input is 1, the circuit operates as a binary counter.

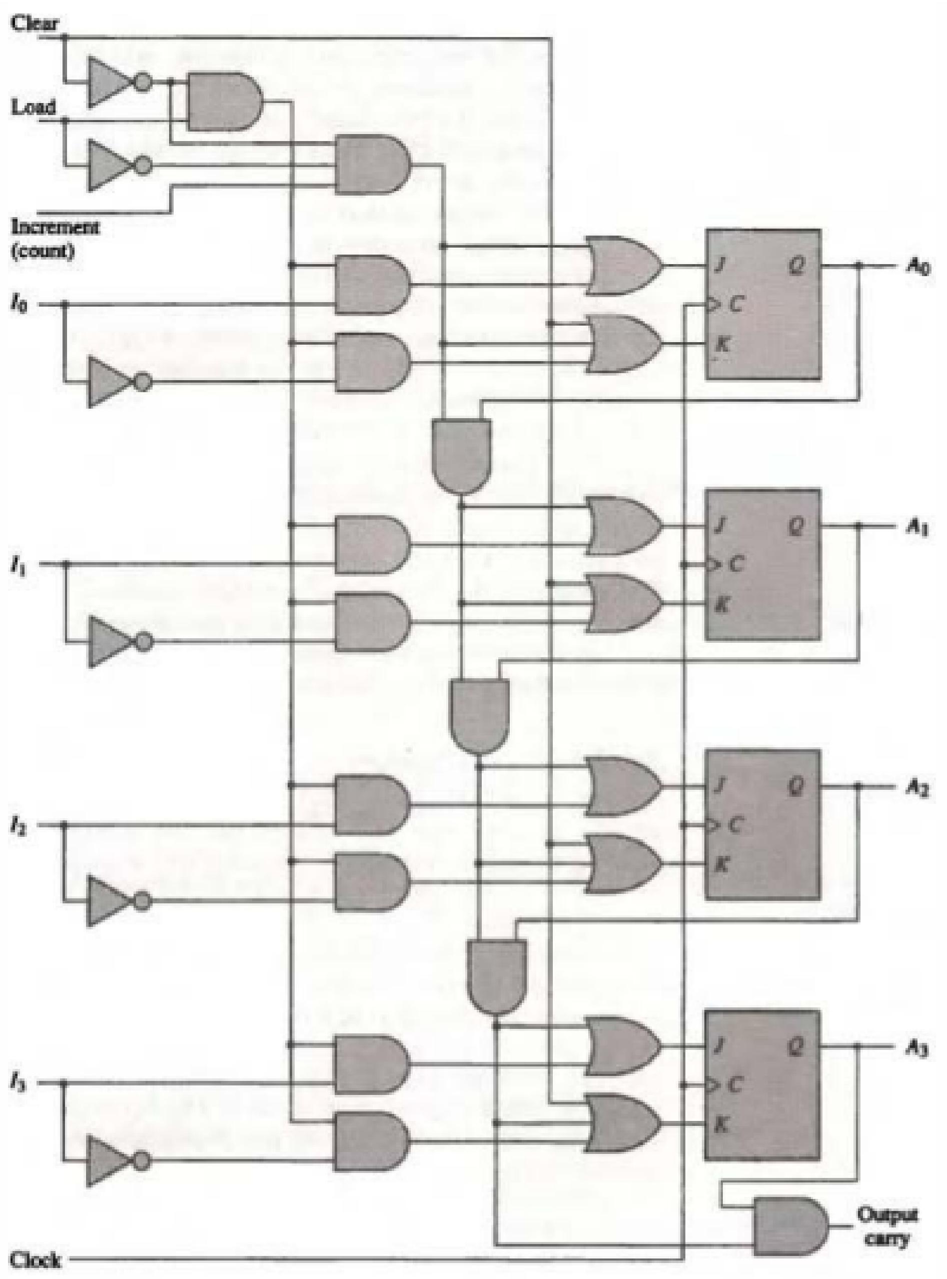


Figure 6.11 4-bit binary counter with parallel load and synchronous clear

The operation of the circuit is summarized in Table below:

Clock	Clear	Load	Increment	Operation
↑	0	0	0	No change
↑	0	0	1	Increment count by 1
↑	0	1	×	Load inputs I_0 through I_3
↑	1	×	×	Clear outputs to 0

With the clear, load, and increment inputs all at 0, the outputs do not change even when pulses are applied to the C terminals. If the clear and load inputs are maintained at logic 0, the increment input controls the operation of the counter and the outputs change to the next binary count for each positive transition of the clock. The input data are loaded into the flip-flops when the load control input is equal to 1 provided that the clear is disabled, but the increment input can be 0 or 1. The register is cleared to 0 with the clear control regardless of the values in the load and increment inputs.

Counters with parallel load are very useful in the design of digital computers. In subsequent chapters we refer to them as registers with load and increment operations. The increment operation adds one to the content of a register. By enabling the count input during one clock period, the content of the register can be incremented by one.

6.5 SUMMARY:

This chapter delves into the essential components of sequential circuits, focusing on their role in computer systems. It begins with an overview of sequential circuits, highlighting their ability to store and process data based on both current inputs and previous states. The chapter then explores registers, which are crucial for temporary data storage, and shift registers, which can shift data either left or right, with types including serial-in-serial-out (SISO), serial-in-parallel-out (SIPO), parallel-in-serial-out (PISO), and parallel-in-parallel-out (PIPO). Finally, it discusses binary counters, which are used to count events or operations, showcasing their importance in timing and control applications within digital systems.

6.6 TECHNICAL TERMS:

Sequential circuits, Registers, Shift Registers, Binary Counters, Memory Unit.

6.7 SELF ASSESSMENT QUESTIONS:

Essay questions:

1) Explain about sequential circuits and differentiate them with combinational circuits.
2) What is Shift Register and explain its types with neat diagrams.

3) Explain 4-bit register with parallel load with a neat diagram

4) Discuss about 4-bit binary counter with parallel load

Short Questions:

1) Define Registers

2) Write the functionalities of shift registers

3) What are binary counters, and how are they used in digital electronics?

4) What are the applications of counters.

6.8 FURTHER READINGS

1) M.Morris Mano, “Computer System Architecture”, 3rd Edition, Pearson Education (2008).

2) V. Rajaraman, T. Radha Krishnan, “Computer Organization and Architecture”, PHI.

3) Behrooz Parhami, “Computer Architecture”, Oxford (2007).

4) ISRD group, “Computer Organization”, ace series, TMH (2007).

5) William Stallings, “Computer Organization and Architecture-Designing for Performance”, Pearson Education (2005).

Dr. Neelima Guntupalli

LESSON-7

DATA REPRESENTATION

OBJECTIVES:

After going through this lesson, you will be able to

- Understand Data Types.
- Apply complements to perform arithmetic operations, particularly subtraction, in binary systems.
- Describe the fixed-point and floating-point representations of numbers.
- Identify and explain various binary codes.
- Understand the importance of error detection codes.

STRUCTURE OF THE LESSION:

7.1 DATA TYPES:

Alphanumeric Representation

Many applications of digital computers require the handling of data that consist not only of numbers, but also of the letters of the alphabet and certain special characters.

An alphanumeric character set is a set of elements that includes the 10 decimal digits, the 26 letters of the alphabet and a number of special characters, such as $, + , and =.

Such a set contains between 32 and 64 elements (if only uppercase letters are included) or between 64 and 128 (if both uppercase and lowercase letters are included).

In the first case, the binary code will require six bits and in the second case, seven bits. The standard alphanumeric binary code is the ASCII (American Standard Code for Information Interchange), which uses seven bits to code 128 characters.

The binary code for the uppercase letters, the decimal digits, and a few special characters is listed in Table below.

Note that the decimal digits in ASCII can be converted to BCD by removing the three high-order bits, 011.

Binary codes play an important part in digital computer operations. The codes must be in binary because registers can only hold binary information. One must realize that binary codes merely change the symbols, not the meaning of the discrete elements they represent.

The operations specified for digital computers must take into consideration the meaning of the bits stored in registers so that operations are performed on operands of the same type.

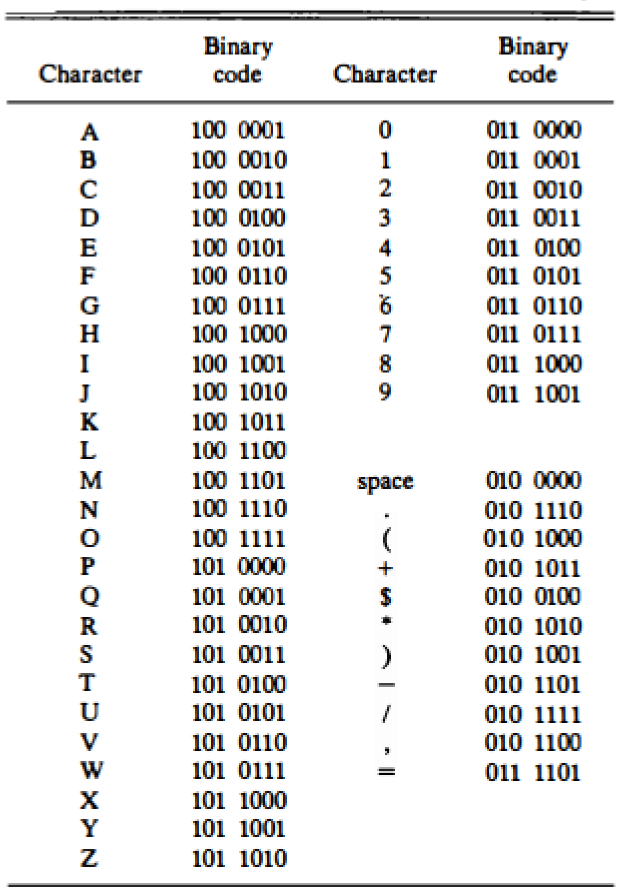


TABLE American Standard Code for Information Interchange (ASCII)

Character	Binary code	Character	Binary code
A	100 0001	0	011 0000
B	100 0010	1	011 0001
C	100 0011	2	011 0010
D	100 0100	3	011 0011
E	100 0101	4	011 0100
F	100 0110	5	011 0101
G	100 0111	6	011 0110
H	100 1000	7	011 0111
I	100 1001	8	011 1000
J	100 1010	9	011 1001
K	100 1011		
L	100 1100		
M	100 1101	space	010 0000
N	100 1110	.	010 1110
O	100 1111	(	010 1000
P	101 0000	+	010 1011
Q	101 0001	$	010 0100
R	101 0010	*	010 1010
S	101 0011	)	010 1001
T	101 0100	−	010 1101
U	101 0101	/	010 1111
V	101 0110	,	010 1100
W	101 0111	=	011 1101
X	101 1000		
Y	101 1001		
Z	101 1010		

7.2 COMPLEMENTS:

Complements are used in digital computers for simplifying the subtraction operation and for logical manipulation. There are two types of complements for each base r system: the r's complement and the (r - 1)'s complement

When the value of the base r is substituted in the name, the two types are referred to as the 2's and 1's complement for binary numbers and the 10's and 9's complement for decimal numbers.

7.2.1 (r - 1)'s Complement:

Given a number N in base r having n digits, the (r - 1)'s complement of N is defined as $(r^n - 1) - N$.

9’s complement

For decimal numbers r = 10 and r - 1 = 9, so the 9's complement of N is $(10^n - 1) - N$.

Now, 10^n represents a number that consists of a single 1 followed by n 0's.

$10^n - 1$ is a number represented by n 9's.

For example, with n = 4 we have $10^4 = 10000$ and $10^4 - 1 = 9999$.

It follows that the 9's complement of a decimal number is obtained by subtracting each digit from 9.

For example, the 9's complement of 546700 is 999999 - 546700 = 453299 and the 9's complement of 12389 is 99999 - 12389 = 87610.

1's complement

For binary numbers, r = 2 and r - 1 = 1, so the 1's complement of N is $(2^n - 1) - N$. Again, 2^n is represented by a binary number that consists of a 1 followed by n 0's. $2^n - 1$ is a binary number represented by n 1's.

For example, with n = 4, we have $2^4 = (10000)_2$ and $2^4 - 1 = (1111)_2$. Thus the 1's complement of a binary number is obtained by subtracting each digit from 1.

However, the subtraction of a binary digit from 1 causes the bit to change from 0 to 1 or from 1 to 0. Therefore, the 1's complement of a binary number is formed by changing 1's into 0's and 0's into 1's.

For example, the 1's complement of 1011001 is 0100110 and the 1's complement of 0001111 is 1110000.

The (r - 1)'s complement of octal or hexadecimal numbers are obtained by subtracting each digit from 7 or F (decimal 15) respectively.

7.2.2 r's Complement:

The r's complement of an n-digit number N in base r is defined as $r^n - N$ for $N \neq 0$ and 0 for $N = 0$.

Comparing with the (r - 1)'s complement, we note that the r's complement is obtained by adding 1 to the (r - 1)'s complement since $r^n - N = [(r^n - 1) - N] + 1$.

10's complement:

Thus the 10's complement of the decimal 2389 is 7610 + 1 = 7611 and is obtained by adding 1 to the 9' s complement value.

The 2's complement of binary 101100 is 010011 + 1 = 010100 and is obtained by adding 1 to the 1's complement value.

Since 10^n is a number represented by a 1 followed by n 0's, then 10^n - N, which is the 10's complement of N, can be formed also be leaving all least significant 0's unchanged, subtracting the first nonzero least significant digit from 10, and then subtracting all higher significant digits from 9.

The 10's complement of 246700 is 753300 and is obtained by leaving the two zeros unchanged, subtracting 7 from 10, and subtracting the other three digits from 9.

2's complement:

The 2's complement can be formed by leaving all least significant 0's and the first 1 unchanged, and then replacing 1's by 0's and 0's by 1's in all other higher significant bits.

The 2's complement of 1101100 is 0010100 and is obtained by leaving the two low-order 0's and the first 1 unchanged, and then replacing 1's by 0's and 0's by 1's in the other four most significant bits.

In the definitions above it was assumed that the numbers do not have a radix point.

If the original number N contains a radix point, it should be removed temporarily to form the r's or (r - 1)'s complement.

The radix point is then restored to the complemented number in the same relative position.

It is also worth mentioning that the complement of the complement restores the number to its original value.

The r's complement of N is r^n - N.

The complement of the complement is $r^n - (r^n - N) = N$ giving back the original number.

7.2.3 Subtraction of Unsigned Numbers

The subtraction of two n-digit unsigned numbers M - N ($N \neq 0$) in base r can be done as follows:

Add the minuend M to the r's complement of the subtrahend N. This performs $M + (r^n - N) = M - N + r^n$.

If $M \geq N$, the sum will produce an end carry r^n which is discarded, and what is left is the result M - N.

If $M < N$, the sum does not produce an end carry and is equal to $r^n - (N - M)$, which is the r's complement of (N - M). To obtain the answer in a familiar form, take the r' s complement of the sum and place a negative sign in front.

Consider, for example, the subtraction 72532 - 13250 = 59282. The 10's complement of 13250 is 86750. Therefore:

$$\begin{array}{rr}
M = & 72532 \\
\text{10's complement of } N = & +\underline{86750} \\
\text{Sum} = & 159282 \\
\text{Discard end carry } 10^5 = & -\underline{100000} \\
\text{Answer} = & 59282
\end{array}$$

Now consider an example with M < N. The subtraction 13250 - 72532 produces negative 59282. Using the procedure with complements, we have

$$\begin{array}{rr}
M = & 13250 \\
\text{10's complement of } N = & +\underline{27468} \\
\text{Sum} = & 40718
\end{array}$$

There is no end carry. Answer is negative 59282 = 10's complement of 40718

Since we are dealing with unsigned numbers, there is really no way to get an unsigned result for the second example.

When subtracting with complements, the negative answer is recognized by the absence of the end carry and the complemented result.

Subtraction with complements is done with binary numbers in a similar manner using the same procedure outlined above.

Using the two binary numbers X = 1010100 and Y = 1000011, we perform the subtraction X - Y and Y - X using 2's complements:

$$\begin{array}{rr}
X = & 1010100 \\
\text{2's complement of } Y = & +\underline{0111101} \\
\text{Sum} = & 10010001 \\
\text{Discard end carry } 2^7 = & -\underline{10000000} \\
\text{Answer: } X - Y = & 0010001 \\
 & \\
Y = & 1000011 \\
\text{2's complement of } X = & +\underline{0101100} \\
\text{Sum} = & 1101111
\end{array}$$

There is no end carry. Answer is negative 0010001 = 2's complement of 1101111

7.3 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 hardware 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 a register is complicated by the fact that it is characterized by a position in the register.

There are two ways of specifying the 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 in 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 of the register to make the stored number an integer.

In either 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.

Integer Representation

When an integer binary number is positive, the sign is represented by 0 and the magnitude by a positive binary number. When the number is negative, the sign is represented by 1 but the rest of the number may be represented in one of three possible ways:

1. Signed-magnitude representation
2. Signed-1's complement representation
7. Signed 2's complement representation

The signed-magnitude representation of a negative number consists of the magnitude and a negative sign. In the other two representations, the negative number is represented in either the 1's or 2's complement of its positive value.

As an example, consider the signed number 14 stored in an 8-bit register. + 14 is represented by a sign bit of 0 in the leftmost position followed by the binary equivalent of 14: 00001110.

Note that each of the eight bits of the register must have a value and therefore 0' s must be inserted in the most significant positions following the sign bit.

Although there is only one way to represent + 14, there are three different ways to represent -14 with eight bits.

In signed-magnitude representation 1 0001110

In signed-1's complement representation 1 1110001

In signed-2's complement representation 1 1110010

The signed-magnitude representation of - 14 is obtained from + 14 by complementing only the sign bit.

The signed-1's complement representation of - 14 is obtained by complementing all the bits of + 14, including the sign bit. The signed-2' s complement representation is obtained by taking the 2' s complement of the positive number, including its sign bit.

The signed-magnitude system is used in ordinary arithmetic but is awkward when employed in computer arithmetic.

Therefore, the signed-complement is normally used. The 1' s complement imposes difficulties because it has two representations of 0 (+0 and - 0).

It is seldom used for arithmetic operations except in some older computers.

The 1's complement is useful as a logical operation since the change of 1 to 0 or 0 to 1 is equivalent to a logical complement operation.

Arithmetic Addition:

The addition of two numbers in the signed-magnitude system follows the rules of ordinary arithmetic.

If the signs are the same, we add the two magnitudes and give the sum the common sign. If the signs are different, we subtract the smaller magnitude from the larger and give the result the sign of the larger magnitude.

For example, (+25) + (-37) = - (37 - 25) = - 12 and is done by subtracting the smaller magnitude 25 from the larger magnitude 37 and using the sign of 37 for the sign of the result.

This is a process that requires the comparison of the signs and the magnitudes and then performing either addition or subtraction. By contrast, the rule for adding numbers in the signed-2's complement system does not require a comparison or subtraction, only addition and complementation.

The procedure can be stated as follows: Add the two numbers, including their sign bits, and discard any carry out of the sign (leftmost) bit position.

Numerical examples for addition are shown below. Note that negative numbers must initially be in 2' s complement and that if the sum obtained after the addition is negative, it is in 2's complement form.

```
 +6   00000110       −6   11111010
+13   00001101      +13   00001101
+19   00010011       +7   00000111

 +6   00000110       −6   11111010
−13   11110011      −13   11110011
 −7   11111001      −19   11101101
```

In each of the four cases, the operation performed is always addition, including the sign bits. Any carry out of the sign bit position is discarded, and negative results are automatically in 2's complement form.

The complement form of representing negative numbers is unfamiliar to people used to the signed-magnitude system.

To determine the value of a negative number when in signed-2's complement, it is necessary to convert it to a positive number to place it in a more familiar form.

For example, the signed binary number 1111 1001 is negative because the leftmost bit is 1.

Its 2's complement is 00000111, which is the binary equivalent of +7. We therefore recognize the original negative number to be equal to -7.

Arithmetic Subtraction:

Subtraction of two signed binary numbers when negative numbers are in 2' s complement form can be stated as follows:

Take the 2's complement of the subtrahend (including the sign bit) and add it to the minuend (including the sign bit). A carry out of the sign bit position is discarded.

This procedure stems from the fact that a subtraction operation can be changed to an addition operation if the sign of the subtrahend is changed. This is demonstrated by the following relationship:

(±A) - (+B) = (±A) + (-B)

(±A) - (-B) = (±A) + (+B)

But changing a positive number to a negative number is easily done by taking its 2's complement.

The reverse is also true because the complement of a negative number in complement form produces the equivalent positive number.

Consider the subtraction of (-6) - (- 13) = +7. In binary with eight bits this is written as 11111010 - 11110011.

The subtraction is changed to addition by taking the 2's complement of the subtrahend (- 13) to give (+ 13). In binary this is 11111010 + 00001101 = 100000111. Removing the end carry, we obtain the correct answer 00000111 (+ 7).

It is worth noting that binary numbers in the signed-2's complement system are added and subtracted by the same basic addition and subtraction rules as unsigned numbers.

Therefore, computers need only one common hardware circuit to handle both types of arithmetic.

The user or programmer must interpret the results of such addition or subtraction differently depending on whether it is assumed that the numbers are signed or unsigned.

Overflow

When two numbers of n digits each are added and the sum occupies n + 1 digits, we say that an overflow occurred.

An overflow is a problem in digital computers because the width of registers is finite. A result that contains n + 1 bits cannot be accommodated in a register with a standard length of n bits. For this reason, many computers detect the occurrence of an overflow, and when it occurs, a corresponding flip-flop is set which can then be checked by the user.

The detection of an overflow after the addition of two binary numbers depends on whether the numbers are considered to be signed or unsigned. When two unsigned numbers are added, an overflow is detected from the end carry out of the most significant position.

In the case of signed numbers, the leftmost bit always represents the sign, and negative numbers are in 2' s complement form.

When two signed numbers are added, the sign bit is treated as part of the number and the end carry does not indicate an overflow.

An overflow cannot occur after an addition if one number is positive and the other is negative, since adding a positive number to a negative number produces a result that is smaller than the larger of the two original numbers.

An overflow may occur if the two numbers added are both positive or both negative. To see how this can happen, consider the following example. Two signed binary numbers, + 70 and + 80, are stored in two 8-bit registers.

The range of numbers that each register can accommodate is from binary + 127 to binary -128. Since the sum of the two numbers is + I50, it exceeds the capacity of the 8-bit register.

This is true if the numbers are both positive or both negative. The two additions in binary are shown below together with the last two carries.

carries:	0 1	carries:	1 0
+70	0 1000110	−70	1 0111010
+80	0 1010000	−80	1 0110000
+150	1 0010110	−150	0 1101010

Note that the 8-bit result that should have been positive has a negative sign bit and the 8-bit result that should have been negative has a positive sign bit. If, however, the carry out of the sign bit position is taken as the sign bit of the result, the 9-bit answer so obtained will be correct.

Since the answer cannot be accommodated within 8 bits, we say that an overflow occurred. An overflow condition can be detected by observing the carry into the sign bit position and the carry out of the sign bit position. If these two carries are not equal, an overflow condition is produced.

This is indicated in the examples where the two carries are explicitly shown. If the two carries are applied to an exclusive-OR gate, an overflow will be detected when the output of the gate is equal to 1.

Decimal Fixed-Point Representation:

The representation of decimal numbers in registers 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

By representing numbers in decimal we are wasting a considerable amount of storage space since the number of bits needed to store a decimal number in a binary code is greater than the number of bits needed for its equivalent binary representation.

Also, the circuits required to perform decimal arithmetic are more complex. However, there are some advantages in the use of decimal representation because computer input and output data are generated by people who use the decimal system.

Some applications, such as business data processing, require small amounts of arithmetic computations compared to the amount required for input and output of decimal data. For this reason, some computers and all electronic calculators perform arithmetic operations directly with the decimal data (in a binary code) and thus eliminate the need for conversion to binary and back to decimal.

Some computer systems have hardware for arithmetic calculations with both binary and decimal data. The representation of signed decimal numbers in BCD is similar to the representation of signed numbers in binary.

We can either use the familiar signed-magnitude system or the signed-complement system. The sign of a decimal number is usually represented with four bits to conform with the 4-bit code of the decimal digits.

It is customary to designate a plus with four 0' s and a minus with the BCD equivalent of 9, which is 1001 .

The signed-magnitude system is difficult to use with computers. The signed-complement system can be either the 9's or the 10's complement, but the 10's complement is the one most often used.

To obtain the 10's complement of a BCD number, we first take the 9's complement and then add one to the least significant digit. The 9's complement is calculated from the subtraction of each digit from 9.

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.

$$\begin{array}{rl} 0\ 375 & (0000\ 0011\ 0111\ 0101)_{BCD} \\ +\underline{9\ 760} & \underline{(1001\ 0111\ 0110\ 0000)}_{BCD} \\ 0\ 135 & (0000\ 0001\ 0011\ 0101)_{BCD} \end{array}$$

The 9 in the leftmost position of the second number indicates that the number is negative. 9760 is the 10's complement of 0240. The two numbers are added and the end carry is discarded to obtain + 137. Of course, the decimal numbers inside the computer must be in BCD, including the sign digits. The addition is done with BCD adders

The subtraction of decimal numbers either unsigned or in the signed-10's complement system is the same as in the binary case. Take the 10's complement of the subtrahend and add it to the minuend.

7.4 FLOATING-POINT REPRESENTATION:

The floating-point representation of a number has two parts. The first part represents a signed, fixed-point number called the mantissa.

The second part designates the position of the decimal (or binary) point and is called the exponent. The fixed-point mantissa may be a fraction or an integer.

For example, the decimal number + 6132.789 is represented in floating-point with a fraction and an exponent as follows: Fraction: +0.6132789; Exponent: +04

The value of the exponent indicates that the actual position of the decimal point is four positions to the right of the indicated decimal point in the fraction.

This representation is equivalent to the scientific notation +0.6132789 X 10^{+4}.

Floating-point is always interpreted to represent a number in the following form: $m * r^e$

Only the mantissa m and the exponent e are physically represented in the register (including their signs). The radix r and the radix-point position of the mantissa are always assumed. The circuits that manipulate the floating-point numbers in registers conform with these two assumptions in order to provide the correct computational results.

A floating-point binary number is represented in a similar manner except that it uses base 2 for the exponent. For example, the binary number + 1001 .11 is represented with an 8-bit fraction and 6-bit exponent as follows: Fraction: 01001110; Exponent: 000100

The fraction has a 0 in the leftmost position to denote positive. The binary point of the fraction follows the sign bit but is not shown in the register.

The exponent has the equivalent binary number +4. The floating-point number is equivalent to $m \times 2^e = +(.1001110)^2 \times 2^{+4}$

A floating-point number is said to be normalized if the most significant digit of the mantissa is nonzero. For example, the decimal number 350 is normalized but 00035 is not.

Regardless of where the position of the radix point is assumed to be in the mantissa, the number is normalized only if its leftmost digit is nonzero. For example, the 8-bit binary number 00011010 is not normalized because of the three leading 0's.

The number can be normalized by shifting it three positions to the left and discarding the leading 0's to obtain 11010000.

The three shifts multiply the number by $2^3 = 8$. To keep the same value for the floating-point number, the exponent must be subtracted by 7.

Normalized numbers provide the maximum possible precision for the floating-point number. A zero cannot be normalized because it does not have a nonzero digit.

It is usually represented in floating-point by all 0's in the mantissa and exponent.

Arithmetic operations with floating-point numbers are more complicated than arithmetic operations with fixed-point numbers and their execution takes longer and requires more complex hardware.

7.5 OTHER BINARY CODES:

7.5.1 Gray Code

Digital systems can process data in discrete form only. Many physical systems supply continuous output data.

The data must be converted into digital form before they can be used by a digital computer. Continuous, or analog, information is converted into digital form by means of an analog-to-digital converter.

The reflected binary or Gray code, shown in Table below, is sometimes used for the converted digital data.

The advantage of the Gray code over straight binary numbers is that the Gray code changes by only one bit as it sequences from one number to the next.

In other words, the change from any number to the next in sequence is recognized by a change of only one bit from 0 to 1 or from 1 to 0. A typical application of the Gray code occurs when the analog data are represented by the continuous change of a shaft position.

The shaft is partitioned into segments with each segment assigned a number. If adjacent segments are made to correspond to adjacent Gray code numbers, ambiguity is reduced when the shaft position is in the line that separates any two segments.

Gray code counters are sometimes used to provide the timing sequences that control the operations in a digital system.

TABLE 4-Bit Gray Code

Binary code	Decimal equivalent	Binary code	Decimal equivalent
0000	0	1100	8
0001	1	1101	9
0011	2	1111	10
0010	3	1110	11
0110	4	1010	12
0111	5	1011	13
0101	6	1001	14
0100	7	1000	15

A Gray code counter is a counter whose flip-flops go through a sequence of states as specified in Table above. Gray code counters remove the ambiguity during the change from one state of the counter to the next because only one bit can change during the state transition.

7.5.2 Other Decimal Codes

Binary codes for decimal digits require a minimum of four bits. Numerous different codes can be formulated by arranging four or more bits in 10 distinct possible combinations. A few possibilities are shown in Table below.

TABLE Four Different Binary Codes for the Decimal Digit

Decimal digit	BCD 8421	2421	Excess-3	Excess-3 gray
0	0000	0000	0011	0010
1	0001	0001	0100	0110
2	0010	0010	0101	0111
3	0011	0011	0110	0101
4	0100	0100	0111	0100
5	0101	1011	1000	1100
6	0110	1100	1001	1101
7	0111	1101	1010	1111
8	1000	1110	1011	1110
9	1001	1111	1100	1010
	1010	0101	0000	0000
Unused	1011	0110	0001	0001
bit	1100	0111	0010	0011
combi-	1101	1000	1101	1000
nations	1110	1001	1110	1001
	1111	1010	1111	1011

The BCD (binary-coded decimal) uses a straight assignment of the binary equivalent of the digit. The six unused bit combinations listed have no meaning when BCD is used, just as the letter H has no meaning when decimal digit symbols are written down.

For example, saying that 1001 1110 is a decimal number in BCD is like saying that 9H is a decimal number in the conventional symbol designation. Both cases contain an invalid symbol and therefore designate a meaningless number.

One disadvantage of using BCD is the difficulty encountered when the 9's complement of the number is to be computed. On the other hand, the 9's complement is easily obtained with the 2421 and the excess-3 codes listed self-complementing in Table above. These two codes have a self-complementing property which means that the 9' s complement of a decimal number, when represented in one of these codes, is easily obtained by changing 1's to 0's and 0's to 1's.

This property is useful when arithmetic operations are done in signed-complement representation. weighted code The 2421 is an example of a weighted code. In a weighted code, the bits are multiplied by the weights indicated and the sum of the weighted bits gives the decimal digit.

For example, the bit combination 1101, when weighted by the respective digits 2421, gives the decimal equivalent of 2 x 1 + 4 x 1 + 2 x 0 + 1 + 1 = 7. The BCD code can be assigned the weights 8421 and for this reason it is sometimes called the 8421 code.

The excess-3 code is a decimal code that has been used in older computers. This is an unweighted code. Its binary code assignment is obtained from the corresponding BCD equivalent binary number after the addition of binary 3 (0011).

From Table in gray code section we note that the Gray code is not suited for a decimal code if we were to choose the first 10 entries in the table. This is because the transition from 9 back to 0 involves a change of three bits (from 1101 to 0000). To overcome this difficulty, we choose the 10 numbers starting from the third entry 0010 up to the twelfth entry 1010.

Now the transition from 1010 to 0010 involves a change of only one bit. Since the code has been shifted up three numbers, it is called the excess-3 Gray. This code is listed with the other decimal codes in Table above.

7.5.3 Other Alphanumeric Codes

The ASCII code (Table below) is the standard code commonly used for the transmission of binary information.

Each character is represented by a 7-bit code and usually an eighth bit is inserted for parity. The code consists of 128 characters. Ninety-five characters represent graphic symbols that include upper- and lowercase letters, numerals zero to nine, punctuation marks, and special symbols.

Twenty-three characters represent format effectors, which are functional characters for controlling the layout of printing or display devices such as carriage return, line feed, horizontal tabulation, and back space. The other 10 characters are used to direct the data communication flow and report its status.

TABLE American Standard Code for Information Interchange (ASCII)

Character	Binary code	Character	Binary code
A	100 0001	0	011 0000
B	100 0010	1	011 0001
C	100 0011	2	011 0010
D	100 0100	3	011 0011
E	100 0101	4	011 0100
F	100 0110	5	011 0101
G	100 0111	6	011 0110
H	100 1000	7	011 0111
I	100 1001	8	011 1000
J	100 1010	9	011 1001
K	100 1011		
L	100 1100		
M	100 1101	space	010 0000
N	100 1110	.	010 1110
O	100 1111	(	010 1000
P	101 0000	+	010 1011
Q	101 0001	$	010 0100
R	101 0010	*	010 1010
S	101 0011	)	010 1001
T	101 0100	−	010 1101
U	101 0101	/	010 1111
V	101 0110	,	010 1100
W	101 0111	=	011 1101
X	101 1000		
Y	101 1001		
Z	101 1010		

Another alphanumeric (sometimes called alphameric) code used in IBM equipment is the EBCDIC (Extended BCD Interchange Code). It uses eight bits for each character (and a ninth bit for parity).

EBCDIC has the same character symbols as ASCII but the bit assignment to characters is different.

When alphanumeric characters are used internally in a computer for data processing (not for transmission purposes) it is more convenient to use a 6-bit code to represent 64 characters.

A 6-bit code can specify the 26 uppercase letters of the alphabet, numerals zero to nine, and up to 28 special characters.

This set of characters is usually sufficient for data-processing purposes.

Using fewer bits to code characters has the advantage of reducing the memory space needed to store large quantities of alphanumeric data.

7.6 ERROR DETECTION CODES:

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 parity bit.

A parity bit is an extra bit included 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 below.

The P(odd) bit is chosen in such a way as to make the sum of 1's (in all four bits) odd. The P(even) bit is chosen to make the sum of all 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 the 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 a 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 incoming 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.

TABLE Parity Bit Generation

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, and only if, an odd function number of variables are equal to 1.

According to this definition, the P(even) 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 (Table above).

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. below, 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 if 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 an 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 networks need an exclusive-NOR at the output to complement the function.

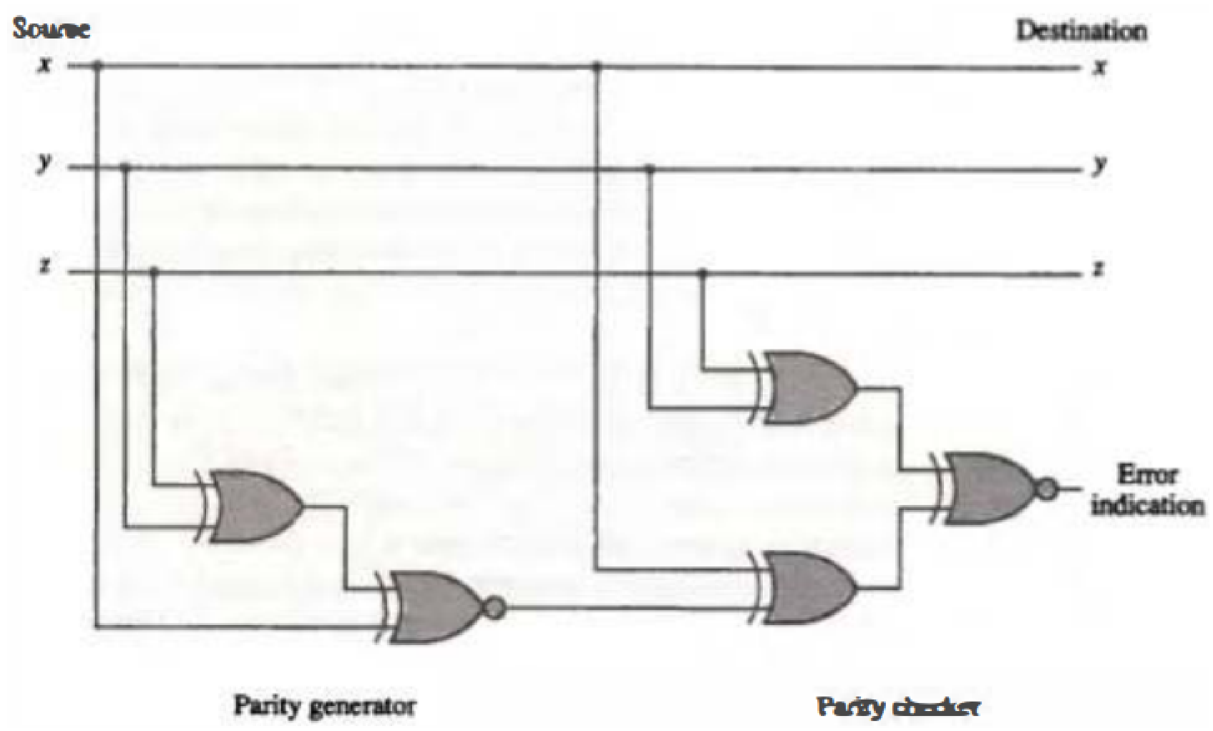


Figure 7.1 Error Detection Code

7.7 SUMMARY:

The chapter on data representation delves into the fundamental concepts of how information is encoded and manipulated within digital systems. It begins with an overview of various data types, including integers, floating-point numbers, and characters, explaining their specific uses and representations. The chapter then explores complements, particularly one's and two's complement, which are essential for performing binary arithmetic and simplifying subtraction operations. Fixed-point representation is examined for its role in representing integers and fractional numbers, while floating-point representation is discussed in the context of handling a wide range of values with precision, adhering to the IEEE 754 standard. Additionally, the chapter covers other binary codes such as Binary Coded Decimal (BCD), Gray code, and ASCII, which are crucial for specific encoding purposes. Lastly, it addresses error detection codes, including parity bits and checksums, which are vital for ensuring data integrity in digital communications by detecting and correcting errors during data transmission.

7.8 TECHNICAL TERMS:

Binary Coded Decimal (BCD), IEEE, Parity bit, complement.

7.9 SELF ASSESSMENT QUESTIONS:

Essay Questions:

1) Describe the process of performing addition and subtraction using fixed-point numbers?
2) Floating Point Representation?
3) What is the need of complement? Explain r-1^s and r^s complement with examples?
4) How does Gray code differ from traditional binary code, and what are its advantages?
5) What is a parity bit, and how is it used for error detection in digital communication?

Short Questions:

1) How is a character typically represented in a computer system, and what encoding standard is commonly used?
2) Explain the concept of two's complement and how it is used for representing signed integers?
3) What is fixed-point representation, and how does it differ from floating-point representation?
4) How do you normalize a floating-point number, and why is normalization important?

7.10 FURTHER READINGS:

1) M.Morris Mano, “Computer System Architecture”, 3rd Edition, Pearson Education (2008).
2) V. Rajaraman, T. Radha Krishnan, “Computer Organization and Architecture”, PHI.
3) BehroozParhami, “Computer Architecture”, Oxford (2007).
4) ISRD group, “Computer Organization”, ace series, TMH (2007).
5) William Stallings, “Computer Organization and Architecture- Designing for Performance”, Pearson Education (2005).

Dr. Neelima Guntupalli

LESSON-8
REGISTER TRANSFER AND MICRO-OPERATIONS

OBJECTIVES:

After going through this lesson, you will be able to

- Understand Register Transfer Languages (RTL)
- Illustrate how data is transferred between different registers within the CPU.
- Comprehend Bus and Memory Transfers
- Describe different arithmetic, logic and shift operations
- Describe how the ALU performs arithmetic, logic, and shift operations.

STRUCTURE OF THE LESSION:

A digital system is an inter connection of digital hardware modules. The modules are registers, decoders, arithmetic elements and control logic. The various modules are interconnected with common data and control paths to form a digital computer system.

Digital modules are best defined by the registers they contain and the operations that are performed on the data stored in them. The operations executed on data stored in registers are called *micro-operations.*

A *micro-operation* is an elementary operation performed on the information stored in one or more registers. The result of the operation may replace the previous binary information of a register or may be transferred to another register.

Examples of micro-operations are shift, count, clear, and load.

The internal hardware organization of a digital computer is best defined by specifying:

1. The set of registers it contains and their function.
2. The sequence of micro-operations performed on the binary information stored in the registers.
3. The control that initiates the sequence of micro-operations.

8.1 REGISTER TRANSFER LANGUAGE:

The symbolic notation used to describe the micro-operation transfer among registers is called RTL (Register Transfer Language).

The use of *symbols* instead of a *narrative explanation* provides an organized and concise manner for listing the micro-operation sequences in registers and the control functions that initiate them.

A register transfer language is a system for expressing in symbolic form the micro-operation sequences among the registers of a digital module.

It is a convenient tool for describing the internal organization of digital computers in concise and precise manner.

Registers:

Computer registers are designated by upper case letters (and optionally followed by digits or letters) to denote the function of the register.

For example, the register that holds an address for the memory unit is usually called a memory address register and is designated by the name *MAR*.

Other designations for registers are PC (for program counter), IR (for instruction register, and *R1* (for processor register).

The individual flip-flops in an n-bit register are numbered in sequence from 0 through n-1, starting from 0 in the rightmost position and increasing the numbers toward the left.

Figure 8 shows the representation of registers in block diagram form.

The most common way to represent a register is by a rectangular box with the name of the register inside, as in Fig. 8.1(a). The individual bits can be distinguished a sin (b). The numbering of bits in a 16-bit register can be marked on top of the box as shown in (c). 16-bit register is partitioned into two parts in (d). Bits 0 through 7 are assigned the symbol L (for low byte) and bits 8 through 15 are assigned the symbol *H* (for high byte).The name of the16-bit register is *PC.* The symbol *PC (0-7)* or *PC (L)* refers to the low-order byte and PC (8-15) or *PC (H)* to the high-order byte**.**

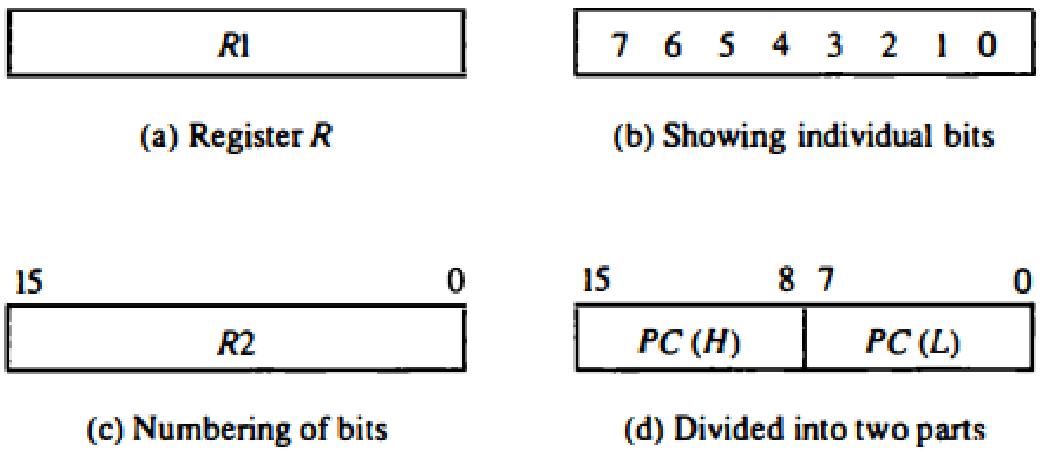


Figure 8.1 Block diagram of a Register

8.2 REGISTER TRANSFER:

Information transfer from one register to another is designated in symbolic form by means of a replacement operator.

The statement R2 $\leftarrow$ R1 denotes a transfer of the content of register R1 into register R2.

It designates a replacement of the content of R2 by the content of R1.

By definition, the content of the source register R 1 does not change after the transfer. If we want the transfer to occur only under a predetermined control condition then it can be shown by an if-then statement.

$$\text{if } (P=1) \text{ then } R2 \leftarrow R1$$

P is the control signal generated by a control section.

We can separate the control variables from the register transfer operation by specifying a Control Function. Control function is a Boolean variable that is equal to 0 or 1.

Control function is included in the statement as

$$P: R2 \leftarrow R1$$

Control condition is terminated by a colon implies transfer operation be executed by the hardware only if P=1.

Every statement written in a register transfer notation implies a hardware construction for implementing the transfer.

Figure 8.2 shows the block diagram that depicts the transfer from R1 to R2.

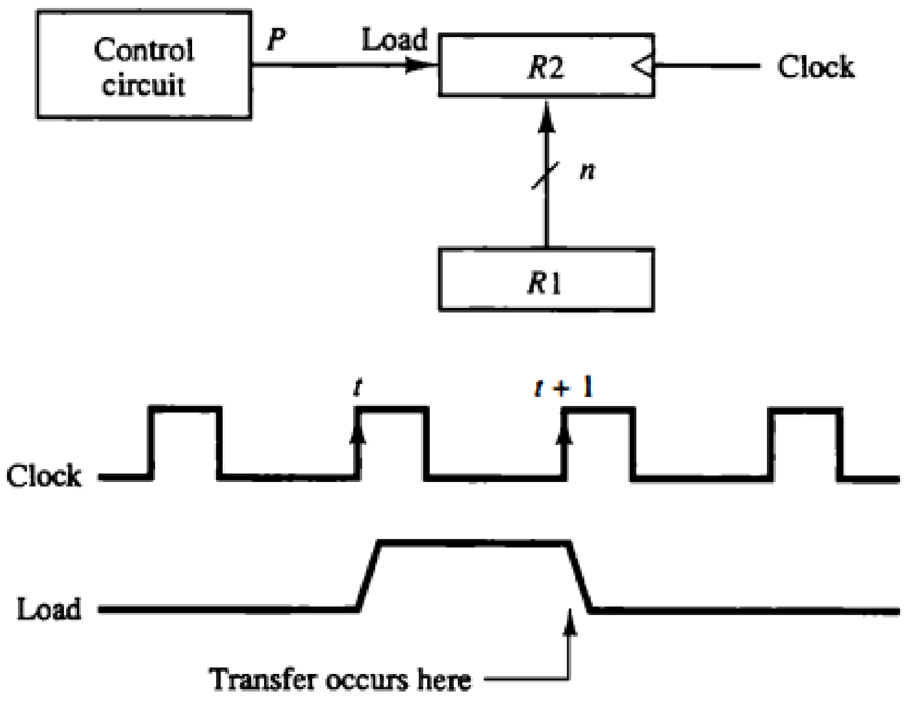


Figure 8.2 Transfer from R1 to R2 when P = 1

The n outputs of register R1 are connected to the n inputs of register R2. The letter n will be used to indicate any number of bits for the register. It will be replaced by an actual number when the length of the register is known. Register R2 has a load input that is activated by the control variable P.

It is assumed that the control variable is synchronized with the same clock as the one applied to the register.As shown in the timing diagram, P is activated in the control section by the rising edge of a clock pulse at time t. The next positive transition of the clock at time t + 1 finds the load input active and the data inputs of R2 are then loaded into the register in parallel.

P may go back to 0 at time t+1; otherwise, the transfer will occur with every clock pulse transition while P remains active. Even though the control condition such as P becomes active just after time t, the actual transfer does not occur until the register is triggered by the next positive transition of the clock at time t +1.

The basic symbols of the register transfer notation are listed in below table

Table 8.1: basic symbols for register transfer

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

A comma is used to separate two or more operations that are executed at the same time.

The statement

$$T : R2 \leftarrow R1,\ R1 \leftarrow R2$$

denotes an operation that exchanges the contents of two registers during one common clock pulse provided that T=1.

8.3 BUS AND MEMORY TRANSFERS:

A more efficient scheme for transferring information between registers in a multiple-register configuration is a Common Bus System. A common bus consists of a set of common lines, one for each bit of a register. Control signals determine which register is selected by the bus during each particular register transfer.

Different ways of constructing a Common Bus System

- Using Multiplexers
- Using Tri-state Buffers

Common bus system is with multiplexers: The multiplexers select the source register whose binary information is then placed on the bus. The construction of a bus system for four registers is shown in below Figure.

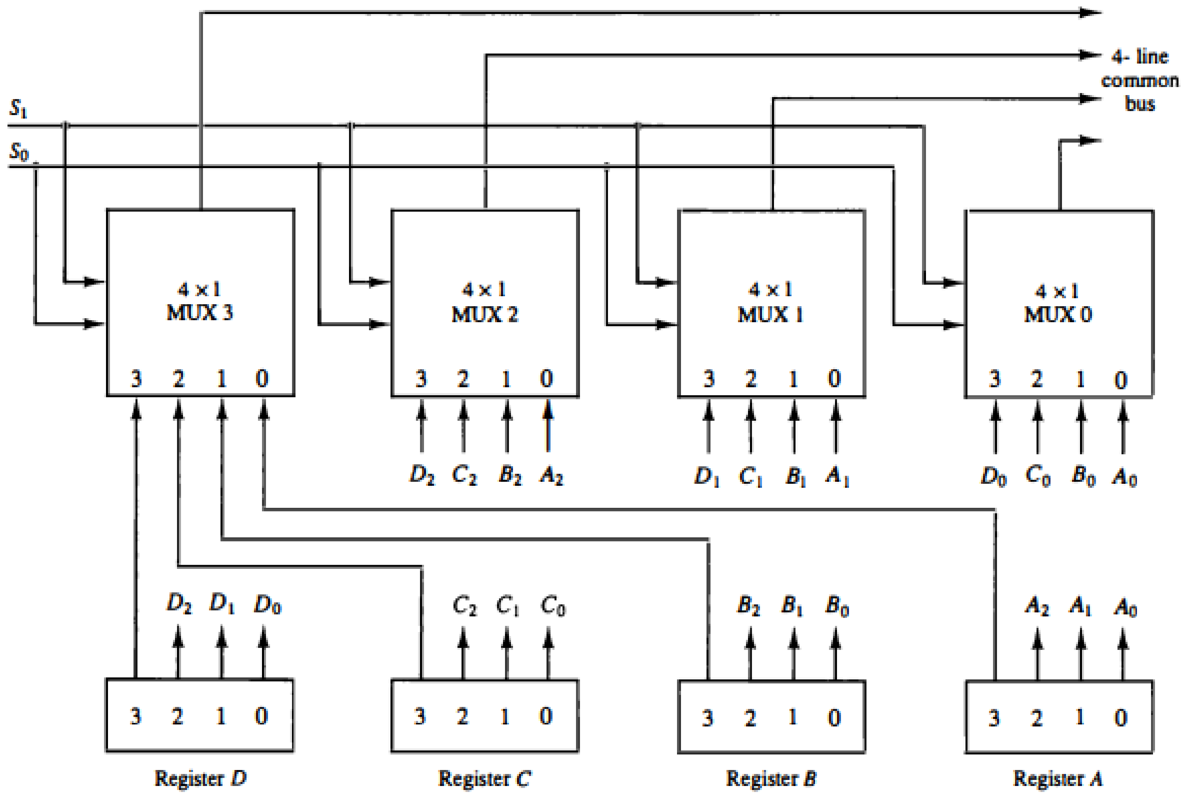


Figure 8.3: Bus system for 4 registers

The bus consists of four 4 x 1 multiplexers each having four data inputs, 0 through 3, and two selection inputs, S_1 and S_0. For example, output 1 of register A is connected to input 0 of MUX1 because this input is labelled A_1. The diagram shows that the bits in the same significant position in each register are connected to the data inputs of one multiplexer to form one line of the bus.

Thus MUX 0 multiplexes the four 0 bits of the registers, MUX 1 multiplexes the four 1 bits of the registers, and similarly for the other two bits. The two selection lines Si and So are connected to the selection inputs of all four multiplexers. The selection lines choose the four bits of one register and transfer them into the four-line common bus.

When S_1S_0 = 00, the 0 data inputs of all four multiplexers are selected and applied to the outputs that form the bus.

This causes the bus lines to receive the content of register A since the outputs of this register are connected to the 0 data inputs of the multiplexers.

Similarly, register B is selected if S_1S_0 = 01, and so on.

Table 8.2 shows the register that is selected by the bus for each of the four possible binary value of the selection lines.

Table 8.2 Function table for the bus

S_1	S_0	Register selected
0	0	A
0	1	B
1	0	C
1	1	D

In general a bus system has multiplex “k” Registers, in each register of “n” bits to produce “n-line bus”, no. of multiplexers required are n, then size of each multiplexer is

k x 1

When the bus is includes in the statement, the register transfer is symbolized as follows:

BUS← C, R1← BUS

The content of register C is placed on the bus, and the content of the bus is loaded into register R1 by activating its load control input. If the bus is known to exist in the system, it may be convenient just to show the direct transfer.

R1← C

Three-State Bus Buffers:

A bus system can be constructed with three-state gates instead of multiplexers. A three-state gate is a digital circuit that exhibits three states. Two of the states are signals equivalent to logic 1 and 0 as in a conventional gate. The third state is a high-impedance state.

The high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have logic significance.

Because of this feature, a large number of three-state gate outputs can be connected with wires to form a common bus line without endangering loading effects.

The graphic symbol of a three-state buffer gate is shown in Fig. 8.8.

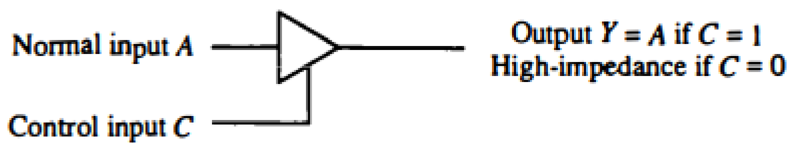


Figure 8.4 Graphic symbols for three state buffers

It is distinguished from a normal buffer by having both a normal input and a control input. The control input determines the output state. When the control input is equal to 1, the output is enabled and the gate behaves like any conventional buffer, with the output equal to the normal input.

When the control input is 0, the output is disabled and the gate goes to a high-impedance state, regardless of the value in the normal input. The construction of a bus system with three-state buffers is shown in Fig. 8.5

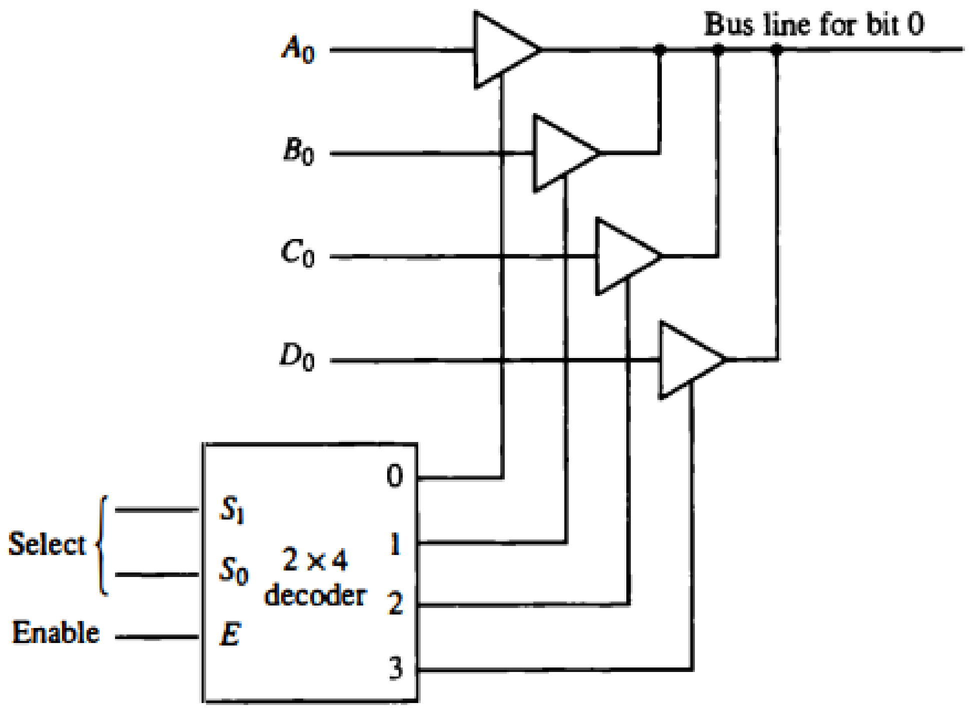


Figure 8.5 bus line with three state-buffers

The outputs of four buffers are connected together to form a single bus line. The control inputs to the buffers determine which of the four normal inputs will communicate with the bus line.

No more than one buffer may be in the active state at any given time. The connected buffers must be controlled so that only one three-state buffer has access to the bus line while all other buffers are maintained in a high impedance state.

One way to ensure that no more than one control input is active at any given time is to use a decoder, as shown in the diagram.

When the enable input of the decoder is 0, all of its four outputs are 0, and the bus line is in a high-impedance state because all four buffers are disabled. When the enable input is active, one of the three-state buffers will be active, depending on the binary value in the select inputs of the decoder.

Memory Transfer:

The transfer of information from a memory word to the outside environment is called a *read* operation. The transfer of new information to be stored into the memory is called a *write* operation.

A memory word will be symbolized by the letter M. The particular memory word among the many available is selected by the memory address during the transfer. It is necessary to specify the address of M when writing memory transfer operations.

This will be done by enclosing the address in square brackets following the letter M. Consider a memory unit that receives the address from a register, called the address register, symbolized by AR.

The data are transferred to another register, called the data register, symbolized by DR.

The read operation can be stated as follows:

Read: DR $\leftarrow$ M [AR]

This causes a transfer of information into DR from the memory word M selected by the address in AR.

The write operation transfers the content of a data register to a memory word M selected by the address. Assume that the input data are in register R1 and the address is in AR.

The write operation can be stated as follows:

Write: M [AR] $\leftarrow$ R1

Types of Micro-operations:

- Register Transfer Micro-operations: Transfer binary information from one register to another.
- Arithmetic Micro-operations: Perform arithmetic operation on numeric data stored in registers.

- Logical Micro-operations: Perform bit manipulation operations on data stored in registers.
- Shift Micro-operations: Perform shift operations on data stored in registers.

Register Transfer Micro-operation doesn’t change the information content when the binary information moves from source register to destination register.

Other three types of micro-operations change the information change the information content during the transfer.

8.4 ARITHMETIC AND MICRO-OPERATIONS:

The basic arithmetic micro-operations are

- Addition
- Subtraction
- Increment
- Decrement
- Shift

The arithmetic Micro-operation defined by the statement below specifies the add micro-operation.

$R3 \leftarrow R1 + R2$

It states that the contents of R1 are added to contents of R2 and sum is transferred to R3.

To implement this statement hardware requires 3 registers and digital component that performs addition

Subtraction is most often implemented through complementation and addition. The subtract operation is specified by the following statement

$R3 \leftarrow R1 + R2 + 1$

instead of minus operator, we can write as

R2 is the symbol for the 1’s complement of R2

Adding 1 to 1’s complement produces 2’s complement

Adding the contents of R1 to the 2's complement of R2 is equivalent to R1-R2.

Binary Adder:

Digital circuit that forms the arithmetic sum of 2 bits and the previous carry is called *Full Adder*. Digital circuit that generates the arithmetic sum of 2 binary numbers of any lengths is called *Binary Adder*.

Figure 8.6 shows the interconnections of four full-adders (FA) to provide a 8.bit binary adder.

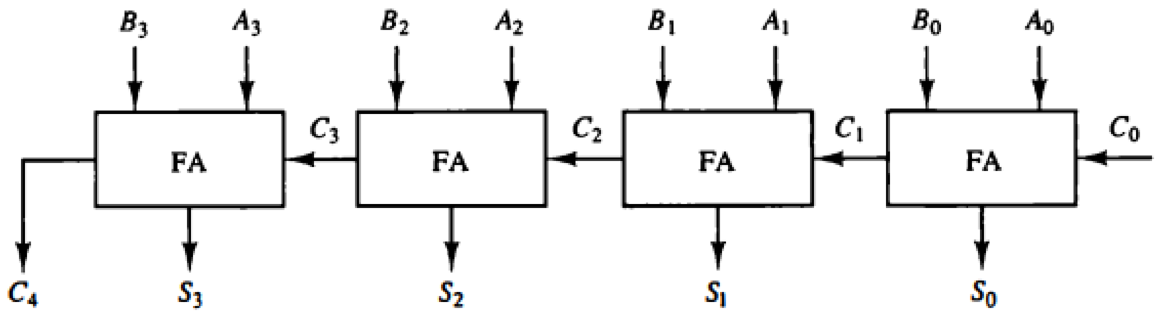


Figure 8.6 8.bit binary adder

The augends bits of A and the addend bits of B are designated by subscript numbers from right to left, with subscript 0 denoting the low-order bit.

The carries are connected in a chain through the full-adders. The input carry to the binary adder is Co and the output carry is C8. The S outputs of the full-adders generate the required sum bits. An n-bit binary adder requires n full-adders

Binary Adder-Subtractor:

The addition and subtraction operations can be combined into one common circuit by including an exclusive-OR gate with each full-adder.

A 8.bit adder-subtractor circuit is shown in Fig. 8.7

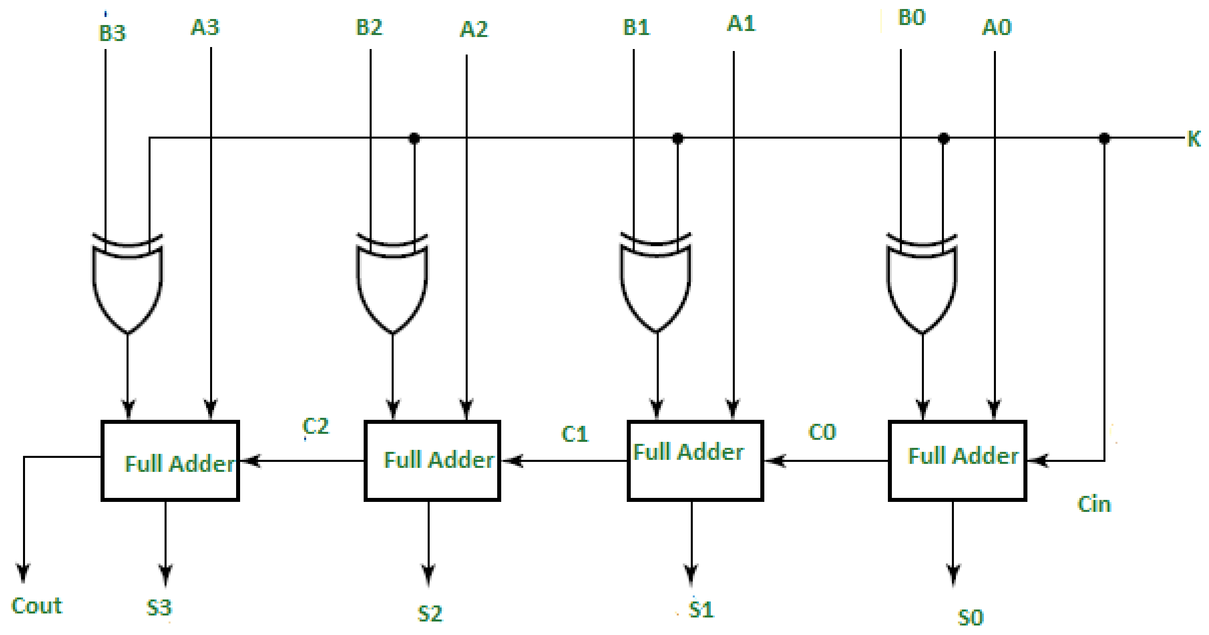


Figure 8.7 8-bit adder subtractor

The mode input M controls the operation. When M = 0 the circuit is an adder and when M = 1 the circuit becomes a subtractor.

Each exclusive-OR gate receives input M and one of the inputs of B

When M = 0, we have B ⊕ 0 = B. The full-adders receive the value of B, the input carry is 0, and the circuit performs A plus B.

When M = 1, we have B ⊕1 = B' and C_0 = 1.

The B inputs are all complemented and a 1 is added through the input carry.

The circuit performs the operation A plus the 2's complement of B.

Binary Incrementer:

The increment micro-operation adds one to a number in a register.

For example, if a 8.bit register has a binary value 0110, it will go to 0111 after it is incremented.

This can be accomplished by means of half-adders connected in cascade.

The diagram of a 8.bit 'combinational circuit incrementer is shown in Fig. 8.8.

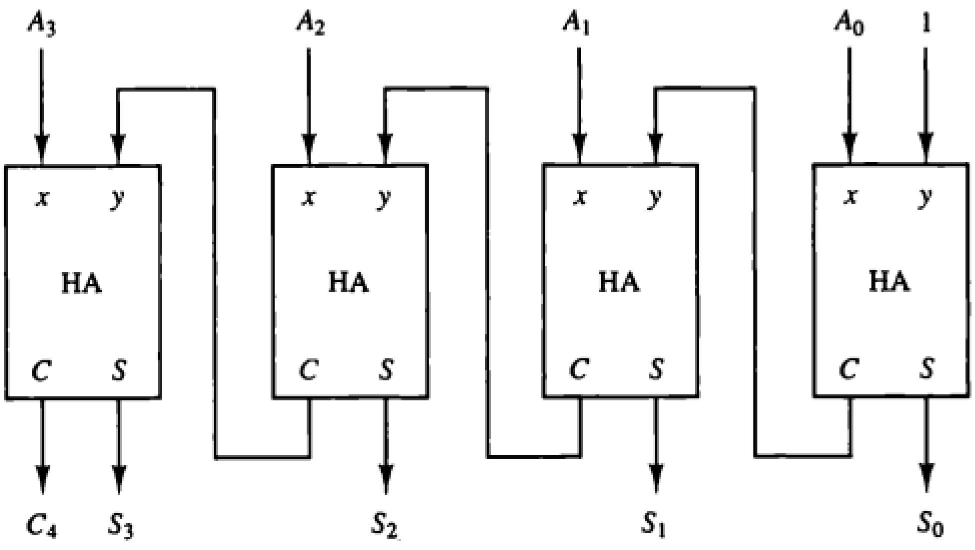


Figure 8.8: 8.bit binary incrementer

One of the inputs to the least significant half-adder (HA) is connected to logic-1 and the other input is connected to the least significant bit of the number to be incremented. The output carry from one half-adder is connected to one of the inputs of the next-higher-order half-adder.

The circuit receives the four bits from A_0 through A_3, adds one to it, and generates the incremented output in S_0 through S_3.The output carry C_4 will be 1 only after incrementing binary 1111. This also causes outputs S_0 through S_3 to go to 0.

The circuit of Fig. 8.8 can be extended to an n -bit binary incrementer by extending the diagram to include n half-adders. The least significant bit must have one input connected to logic-1. The other inputs receive the number to be incremented or the carry from the previous stage.

Arithmetic Circuit:

The basic component of an arithmetic circuit is the parallel adder. By controlling the data inputs to the adder, it is possible to obtain different types of arithmetic operations.

The diagram of a 8.bit arithmetic circuit is shown in Fig. 8.9. It has four full-adder circuits that constitute the 8.bit adder and four multiplexers for choosing different operations.

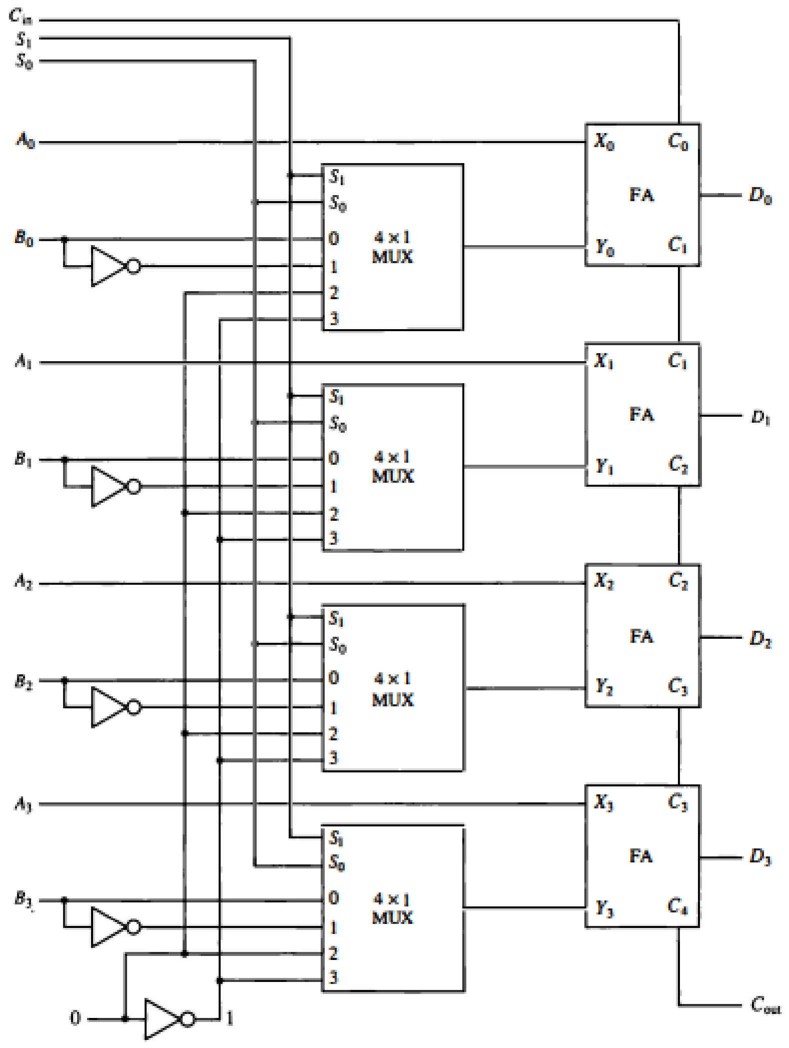


Figure 8.9: 8.bit arithmetic circuit

There are two 8.bit inputs A and B and a 8.bit output D. The four inputs from A go directly to the X inputs of the binary adder. Each of the four inputs from B are connected to the data inputs of the multiplexers. The multiplexer's data inputs also receive the complement of B. The other two data inputs are connected to logic-0 and logic-1.

The four multiplexers are controlled by two selection inputs S_1 and S_0. The input carry C_{in}, goes to the carry input of the FA in the least significant position. The other carries are connected from one stage to the next.

By controlling the value of Y with the two selection inputs S_1 and S_0 and making Cin equal to 0 or 1, it is possible to generate the eight arithmetic micro-operations listed in Table 48.

Addition:

When S1S0= 00, the value of B is applied to the Y inputs of the adder.

- if C_{in}, = 0, the output D =A+B.
- if C_{in} = 1, output D=A+B + 1.

Both cases perform the add micro-operation with or without adding the input carry.

Subtraction:

When S_1S_0 = 01, the complement of B is applied to the Y inputs of the adder.

- if C_{in} = 1, then D = A + B + 1. This produces A plus the 2's complement of B, which is equivalent to a subtraction of A -B.
- When C_{in} = 0 then D = A + B. This is equivalent to a subtract with borrow, that is, A-B-1.

I**ncrement:**

When S_1S_0 = 10, the inputs from B are neglected, and instead, all 0's are inserted into the Y inputs.

- if C_{in} = 0, then D = A + 0 + C_{in}.
- if C_{in} = 1, then D = A + 1

In the first case we have a direct transfer from input A to output D. In the second case, the value of A is incremented by 1.

Decrement:

When S_1S_0= 11, all l's are inserted into the Y inputs of the adder to produce the decrement operation D = A -1 when Cin = 0.

This is because a number with all 1's is equal to the 2's complement of 1 (the 2's complement of binary 0001 is 1111). Adding a number A to the 2's complement of 1 produces F = A + 2's complement of 1 = A - 1. When C_{in} = 1, then D = A -1 + 1=A, which causes a direct transfer from input A to output D.

8.5 LOGIC MICRO-OPERATIONS:

Logic micro-operations specify binary operations for strings of bits stored in registers. These operations consider each bit of the register separately and treat them as binary variables.

For example, the exclusive-OR micro-operation with the contents of two registers RI and R2 is symbolized by the statement

$$P: R1 \leftarrow R1 \oplus R2$$

It specifies a logic micro-operation to be executed on the individual bits of the registers provided that the control variable P = 1.

List of Logic Micro-operations:

There are 16 different logic operations that can be performed with two binary variables. They can be determined from all possible truth tables obtained with two binary variables as shown in Table 8.3.

Table 8.3 Truth table for 16 Functions of Two 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

The 16 Boolean functions of two variables x and y are expressed in algebraic form in the first column of Table 8.3.

The 16 logic micro-operations are derived from these functions by replacing variable x by the binary content of register A and variable y by the binary content of register B.

The logic micro-operations listed in the second column represent a relationship between the binary content of two registers A and B.

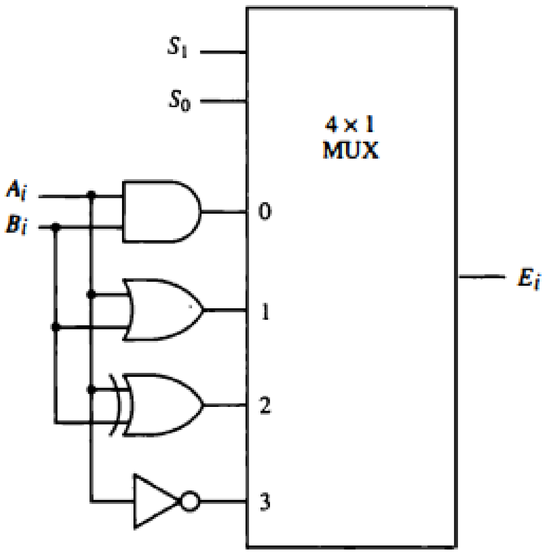
Table 8.4: Sixteen logic micro-operations

Boolean function	Microoperation	Name
$F_0 = 0$	$F \leftarrow 0$	Clear
$F_1 = xy$	$F \leftarrow A \wedge B$	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

Hardware Implementation:

The hardware implementation of logic micro-operations requires that logic gates be inserted for each bit or pair of bits in the registers to perform the required logic function. Although there are 16 logic micro-operations, most computers use only four--AND, OR, XOR (exclusive-OR), and complement from which all others can be derived.

Figure 8.10 shows one stage of a circuit that generates the four basic logic micro-operations.



(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 8.10 one stage of logic circuit

It consists of four gates and a multiplexer. Each of the four logic operations is generated through a gate that performs the required logic. The outputs of the gates are applied to the data inputs of the multiplexer. The two selection inputs S_1 and S_0 choose one of the data inputs of the multiplexer and direct its value to the output.

Some Applications:

Logic micro-operations are very useful for manipulating individual bits or a portion of a word stored in a register.

They can be used to change bit values, delete a group of bits or insert new bits values into a register.

The following example shows how the bits of one register (designated by A) are manipulated by logic micro-operations as a function of the bits of another register (designated by B).

Selective set

The selective-set operation sets to 1 the bits in register A where there are corresponding l's in register B. It does not affect bit positions that have 0's in B. The following numerical example clarifies this operation:

1010	A before
$\underline{1100}$	B (Logic operand)
1110	A After

The OR micro-operation can be used to selectively set bits of a register.

Selective complement

The selective-complement operation complements bits in A where there are corresponding 1's in B. It does not affect bit positions that have 0's in B. For example:

1010	A before
$\underline{1100}$	B (Logic operand)
0110	A After

The exclusive-OR micro-operation can be used to selectively complement bits of a register.

Selective clear

The selective-clear operation clears to 0 the bits in A only where there are corresponding l's in B. For example:

1010	A before
$\underline{1100}$	B (Logic operand)
0010	A After

The corresponding logic micro-operation is

Mask

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. The mask operation is an AND micro operation as seen from the following numerical example:

1010	A before
$\underline{1100}$	B (Logic operand)
1000	A After

Insert

The insert operation inserts a new value into a group of bits. This is done by first masking the bits and then ORing them with the required value.

0110 1010	A before
$\underline{0000\ 1111}$	B (mask)
0000 1010	A After

For example, suppose that an A register contains eight bits, 0110 1010. To replace the four leftmost bits by the value 1001 we first mask the four unwanted bits:

0000 1010	A before
1001 0000	B (mask)
1001 1010	A After insertion

The mask operation is an AND micro-operation and the insert operation is an OR micro-operation.

Clear

The clear operation compares the words in A and B and produces an all 0's result if the two numbers are equal. This operation is achieved by an exclusive-OR micro-operation as shown by the following example

1010	A before
1010	B
0010	$A \leftarrow A \oplus B$

8.6 SHIFT MICRO-OPERATOINS:

Shift micro-operations are used for serial transfer of data. The contents of a register can be shifted to the left or the right. During a shift-left operation the serial input transfers a bit into the rightmost position. During a shift-right operation the serial input transfers a bit into the leftmost position.

There are three types of shifts: logical, circular, and arithmetic.

The symbolic notation for the shift micro-operations is shown in Table 8.5.

Table 8.5: Shift micro-operations

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

Logical Shift:

A logical shift is one that transfers 0 through the serial input. The symbols shl and shr for logical shift-left and shift-right micro-operations.

The micro-operations that specify a 1-bit shift to the left of the content of register R and a 1-bit shift to the right of the content of register R shown in table 8.5. The bit transferred to the end position through the serial input is assumed to be 0 during a logical shift.

$$R1 \leftarrow \text{shl } R1$$

$$R2 \leftarrow \text{shr } R2$$

Circular Shift:

The circular shift (also known as a rotate operation) circulates the bits of the register around the two ends without loss of information. This is accomplished by connecting the serial output of the shift register to its serial input. We will use the symbols *cil* and *cir* for the circular shift left and right, respectively.

Arithmetic Shift:

An arithmetic shift is a micro-operation that shifts a signed binary number to the left or right. An arithmetic shift-left multiplies a signed binary number by 2. An arithmetic shift-right divides the number by 2. Arithmetic shifts must leave the sign bit unchanged because the sign of the number remains the same when it is multiplied or divided by 2.

Hardware Implementation:

A combinational circuit shifter can be constructed with multiplexers as shown in Fig. 8.12.

The 8.bit shifter has four data inputs, A_0 through A_3, and four data outputs, H_0 through H_3. There are two serial inputs, one for shift left (IL) and the other for shift right (IR).

When the selection input S=0 the input data are shifted right (down in the diagram).

When S = 1, the input data are shifted left (up in the diagram).

The function table in Fig. 8.12 shows which input goes to each output after the shift. A shifter with n data inputs and outputs requires n multiplexers. The two serial inputs can be controlled by another multiplexer to provide the three possible types of shifts.

8.7 ARITHMETIC LOGIC SHIFT UNIT:

Instead of having individual registers performing the micro-operations directly, computer systems employ a number of storage registers connected to a common operational unit called an arithmetic logic unit, abbreviated ALU.

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. The shift micro-operations are often performed in a separate unit, but sometimes the shift unit is made part of the overall ALU.

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. 8.13.

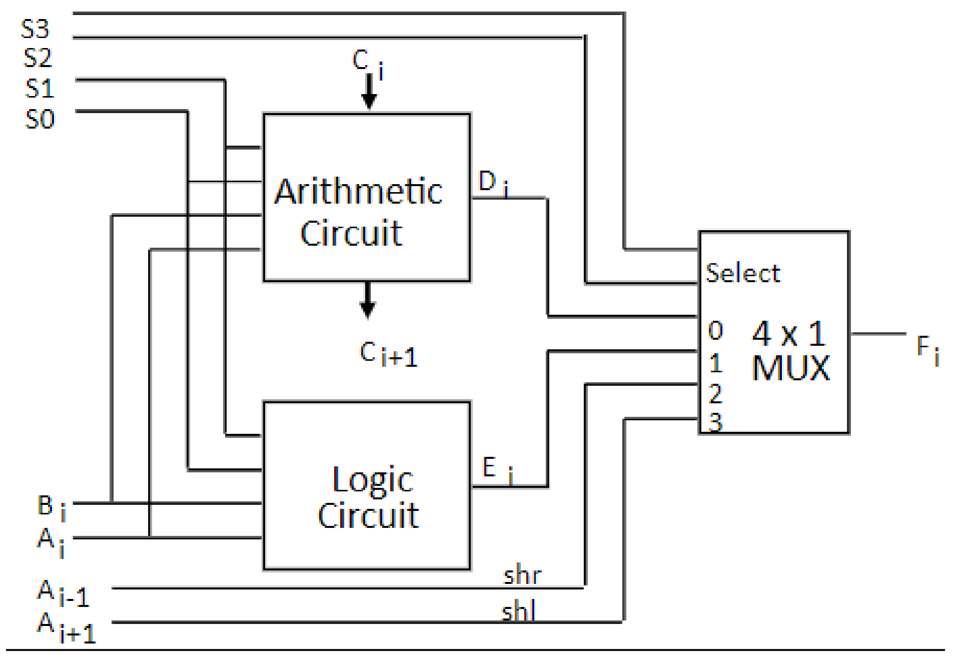


Figure 8.11: One stage of arithmetic logic shift unit

Particular micro-operation is selected with inputs S_1 and S_0. A 4 x 1 multiplexer at the output chooses between an arithmetic output in Di and a logic output in Ei. The data in the multiplexer are selected with inputs S_3 and S_2. The other two data inputs to the multiplexer receive inputs Ai-1 for the shift-right operation and Ai+1 for the shift-left operation.

The circuit whose one stage is specified in Fig. 8.11 provides eight arithmetic operation, four logic operations, and two shift operations. Each operation is selected with the five variables S_3, S_2, S_1, S_0 and Cin. The input carry Cin is used for selecting an arithmetic operation only.

Table 8.6: Function table of Arithmetic logic shift unit

Operation select						
S_3	S_2	S_1	S_0	C_{in}	Operation	Function
0	0	0	0	0	$F = A$	Transfer A
0	0	0	0	1	$F = A + 1$	Increment A
0	0	0	1	0	$F = A + B$	Addition
0	0	0	1	1	$F = A + B + 1$	Add with carry
0	0	1	0	0	$F = A + \overline{B}$	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	×	×	×	$F = \text{shr } A$	Shift right A into F
1	1	×	×	×	$F = \text{shl } A$	Shift left A into F

Table 8.7: lists the 14 operations of the ALU.

The first eight are arithmetic operations and are selected with S_3S_2 = 00. The next four are logic and are selected with S_3S_2 = 01.The input carry has no effect during the logic operations and is marked with don't-care x's. The last two operations are shift operations and are selected with S_3S_2= 10 and 11.The other three selection inputs have no effect on the shift.

8.8 SUMMARY:

The chapter on Register Transfer and Micro-operations delves into the fundamental mechanisms of data movement and transformation within a computer system. It begins with an exploration of Register Transfer Languages (RTL), providing a framework for describing data transfers between registers. The chapter then covers the concept of register transfers, detailing how data is moved between various CPU registers. It discusses bus and memory transfers, explaining the role of different types of buses in facilitating data movement between the CPU, memory, and other components. Arithmetic micro-operations, such as addition and subtraction, are examined alongside logic micro-operations like AND, OR, and NOT, showcasing their importance in data manipulation. Shift micro-operations, including logical and arithmetic shifts, are explained for their role in altering data positioning. Finally, the chapter introduces the Arithmetic Logic Shift Unit (ALU), illustrating its critical function in performing arithmetic, logic, and shift operations, and integrating these processes to execute instructions within the CPU.

8.9 TECHNICAL TERMS:

RTL, Control signals, Bus, Arithmetic Logic Shift Unit (ALSU)

8.10 SELF ASSESSMENT QUESTIONS:

Essay questions:

1) Define Register Transfer Language and explain its importance in computer architecture?
2) Describe how data is transferred between the CPU, memory, and other components using buses?
3) Describe various arithmetic operations?
4) Demonstrate how logical operations are applied to binary data?

Short Questions:

1) Define and explain the purpose of Register Transfer Languages?
2) Explain the concept of register transfer and its significance in computer operations?
3) Explain the different types of buses and their functions?
4) Define shift micro-operations and their uses?

8.11 FURTHER READINGS:

1) M.Morris Mano, “Computer System Architecture”, 3rd Edition, Pearson Education (2008).

2) V. Rajaraman, T. Radha Krishnan, “Computer Organization and Architecture”, PHI.

3) BehroozParhami, “Computer Architecture”, Oxford (2007).

4) ISRD group, “Computer Organization”, ace series, TMH (2007).

5) William Stallings, “Computer Organization and Architecture-Designing for Performance”, Pearson Education (2005).

Dr. Neelima Guntupalli

LESSON-9

BASIC COMPUTER ORGANIZATION AND DESIGN

OBJECTIVES:

After going through this lesson, you will be able to

- Identify and explain different types of instruction formats and registers.
- Explain the execution of computer instructions.
- Understand the role of the control unit.
- Illustrate the sequence of events in the instruction cycle.
- Describe different types of memory reference instructions.
- Understand the concept of interrupts.

STRUCTURE OF THE LESSION:

9.1 INSTRUCTION CODES:

The organization of the computer is defined by its internal registers, the timing and control structure, and the set of instructions that it uses. Internal organization of a computer is defined by the sequence of micro-operations it performs on data stored in its registers.

Computer can be instructed about the specific sequence of operations it must perform. User controls this process by means of a Program.

Program: set of instructions that specify the operations, operands, and the sequence by which processing has to occur.

Instruction: a binary code that specifies a sequence of micro-operations for the computer.

The computer reads each instruction from memory and places it in a control register. The control then interprets the binary code of the instruction and proceeds to execute it by issuing a sequence of micro-operations.–Instruction Cycle

Instruction Code: group of bits that instruct the computer to perform specific operation.

Instruction code is usually divided into two parts: Opcode and address (operand)

Operation Code (opcode): group of bits that define the operation

Eg: add, subtract, multiply, shift, complement.

No. of bits required for opcode depends on no. of operations available in computer.

n bit opcode>= 2^n (or less) operations

Address (operand): specifies the location of operands (registers or memory words). Memory words are specified by their address.

Registers are specified by their k-bit binary code

k-bit address >= 2^k registers

Stored Program Organization

The ability to store and execute instructions is the most important property of a general-purpose computer. That type of stored program concept is called 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.

The below figure 9.1 shows the stored program organization

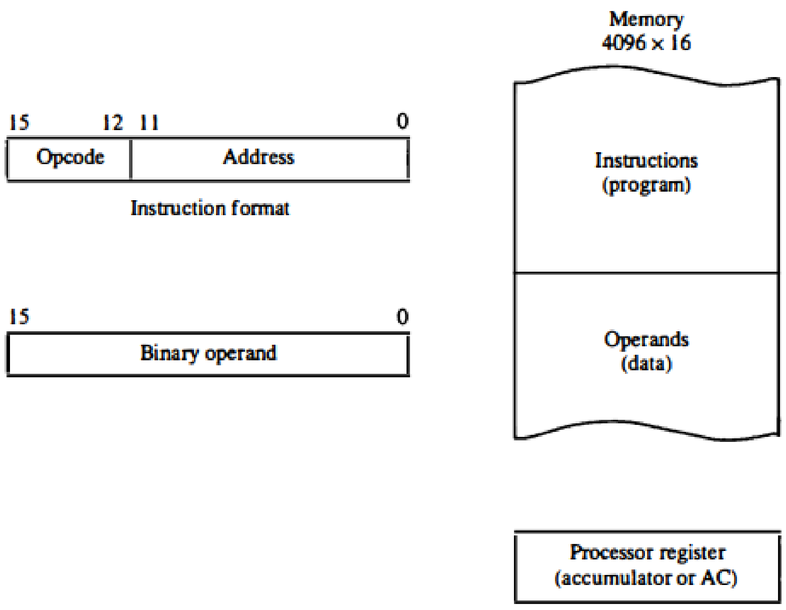


Figure 9.1 stored program organization

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 the operation code (abbreviated opcode) to specify one out of 16 possible operations, and 12 bits to specify the address of an operand.

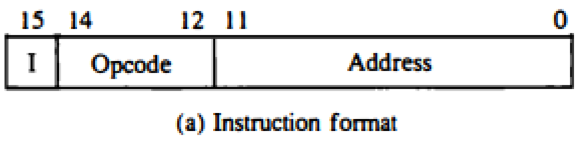
Accumulator (AC): Computers that have a single-processor register usually assign to it the name accumulator and label it AC. The operation is performed with the memory operand and the content of AC.

Indirect address:

Addressing of Operand: Sometimes convenient to use the address bits of an instruction code not as an address but as the actual operand.

When the second part of an instruction code specifies an operand, the instruction is said to have an *immediate operand.* When the second part specifies the address of an operand, the instruction is said to have a *direct address.* When second part of the instruction designate an address of a memory word in which the address of the operand is found such instruction have *indirect address.*

One bit of the instruction code can be used to distinguish between a direct and an indirect address.



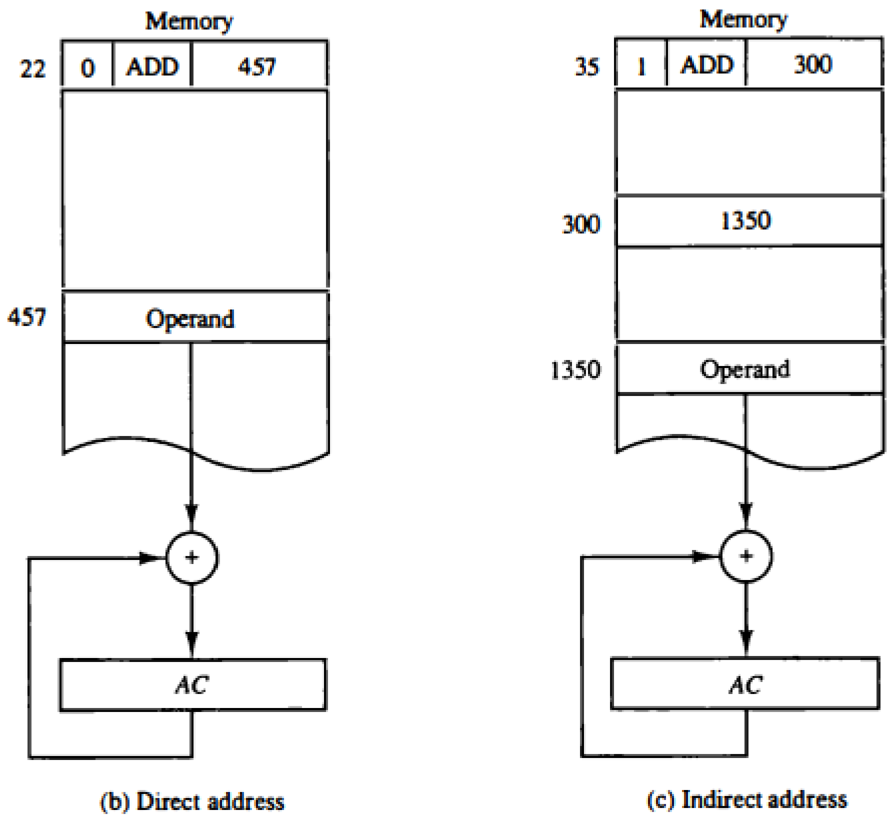


Figure 9.2 Demonstration of direct and indirect address

The instruction code format shown in Fig. 9.2(a). It consists of a 3-bit operation code, a 12-bit address, and an indirect address mode bit designated by I. The mode bit is 0 for a direct address and 1 for an indirect address.

A direct address instruction is shown in Fig. 9.2(b).

It is placed in address 22 in memory. The I bit is 0, so the instruction is recognized as a direct address instruction. The opcode specifies an ADD instruction, and the address part is the binary equivalent of 459. 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 Fig. 9.2(c) has a mode bit I = 1. Therefore, it is recognized as an indirect address instruction. The address part is the binary equivalent of 300. 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 *effective address* to be the address of the operand in a computation-type instruction or the target address in a branch-type instruction.

Thus the effective address in the instruction of Fig. 9.2(b) is 457 and in the instruction of Fig 9.2(c) is 1350.

9.2 COMPUTER REGISTERS:

The need of the registers in computer for

- Instruction sequencing needs a counter to calculate the address of the next instruction after execution of the current instruction is completed (PC).
- Necessary to provide a register in the control unit for storing the instruction code after it is read from memory (IR).
- Needs processor registers for manipulating data (AC and TR) and a register for holding a memory address (AR).

Table 9.1:

Register symbol	Number of 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

The registers are also listed in Table 9.1 together with a brief description of their function and the number of bits that they contain.

The data register (DR) holds the operand read from memory.

The accumulator (AC) register is a general purpose processing 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 12 bits since this is the width of a memory address.

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.

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.

The above requirements dictate the register configuration shown in Fig. 9.3.

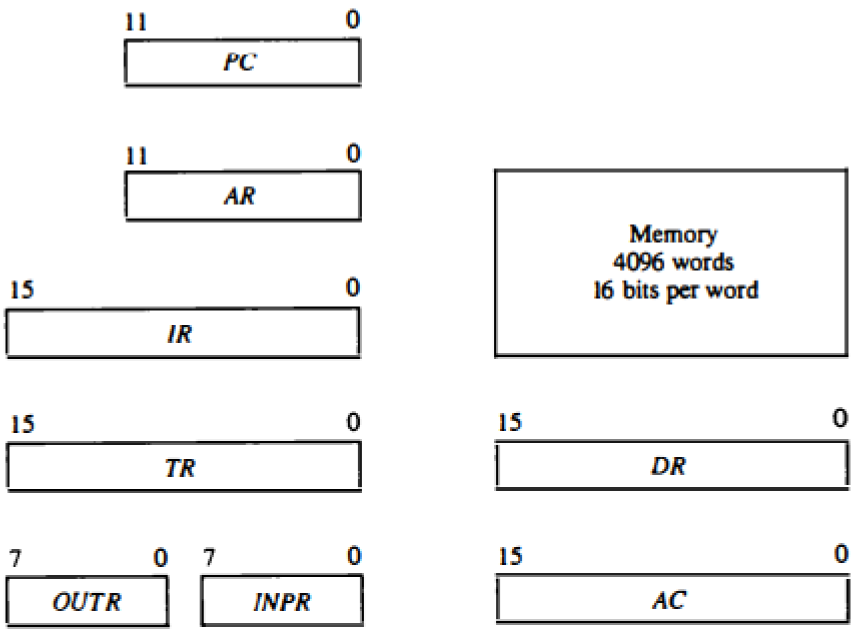


Figure 9.3 Basic computer register and memory

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 registers. A more efficient scheme for transferring information in a system with many registers 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 Fig. 9.4.

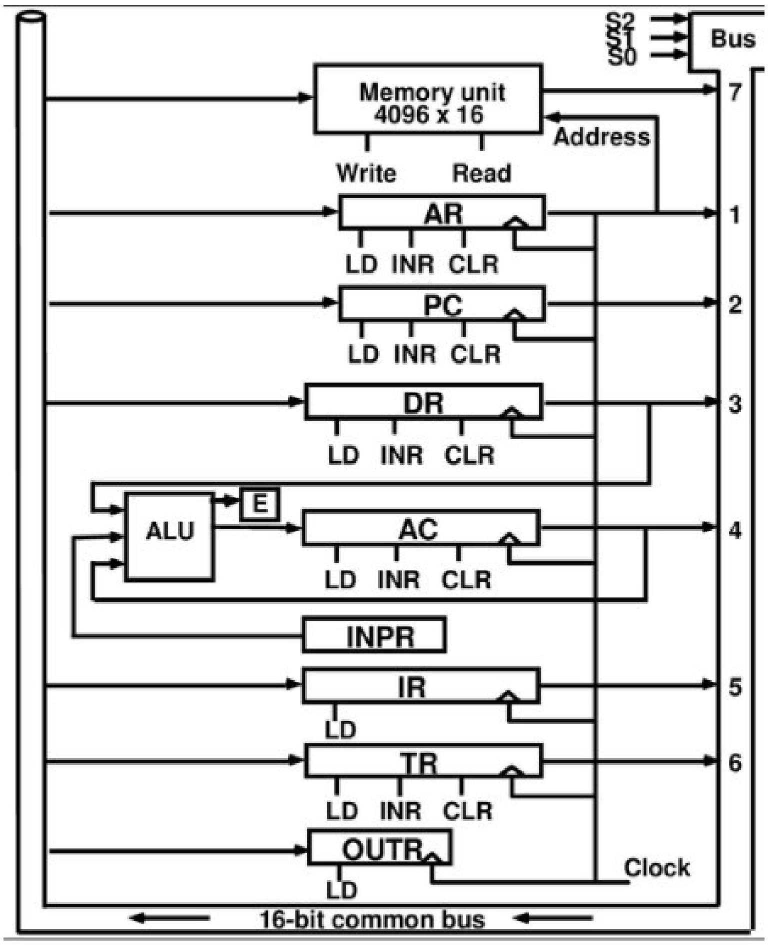


Figure 9.4 basic computer registers connected to a common bus

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 S_2, S_1, and S_0. The number along each output shows the decimal equivalent of the required binary selection.

For example, the number along the output of DR is 3. The 16-bit outputs of DR are placed on the bus lines when $S_2S_1S_0 = 011$. The lines from the common bus are connected to the inputs of each register and the data inputs of the memory.

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 $S_2S_1S_0 = 111$.

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 or PC receives 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. They 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. This is because INPR receives a character from an input device which is then transferred to AC. OUTR receives a character from AC and delivers it to an output device.

Five registers have three control inputs: LD (load), INR (increment), and CLR (clear). This type of register is equivalent to a binary counter with parallel load and synchronous clear. Two registers have only a LD input. The input data and output data of the memory are connected to the common bus, but the memory address is connected to AR. Therefore, AR must always be used to specify a memory address.

The 16 inputs of AC come from an adder and logic circuit. This circuit has three sets of inputs.

- One set of 16-bit inputs come from the outputs of AC.
- Another set of 16-bit inputs come from the data register DR.
- 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).
- A third set of 8-bit inputs come from the input register INPR.

The content of any register can be applied onto the bus and an operation can be performed in the adder and logic circuit during the same clock cycle.

For example, the two microoperations DR $\leftarrow$ AC and AC $\leftarrow$ DR can be executed at the same time.

This can be done by placing the content of AC on the bus (with $S_2S_1S_0 = 100$), enabling the LD (load) input of DR, transferring the content of DR through the adder and logic circuit into AC, and enabling the LD (load) input of AC, all during the same clock cycle.

9.3 COMPUTER INSTRUCTIONS:

The basic computer has three instruction code formats, as shown in Fig. 9.9. Each format has 16 bits.

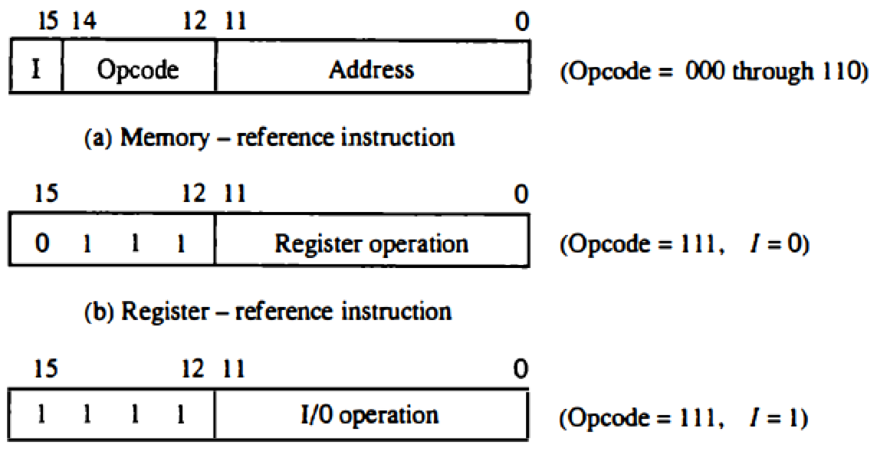


Figure 9.5 Basic computer instruction formats

The operation code (opcode) part of the instruction contains three bits and the meaning of the remaining 13 bits depends on the operation code encountered.

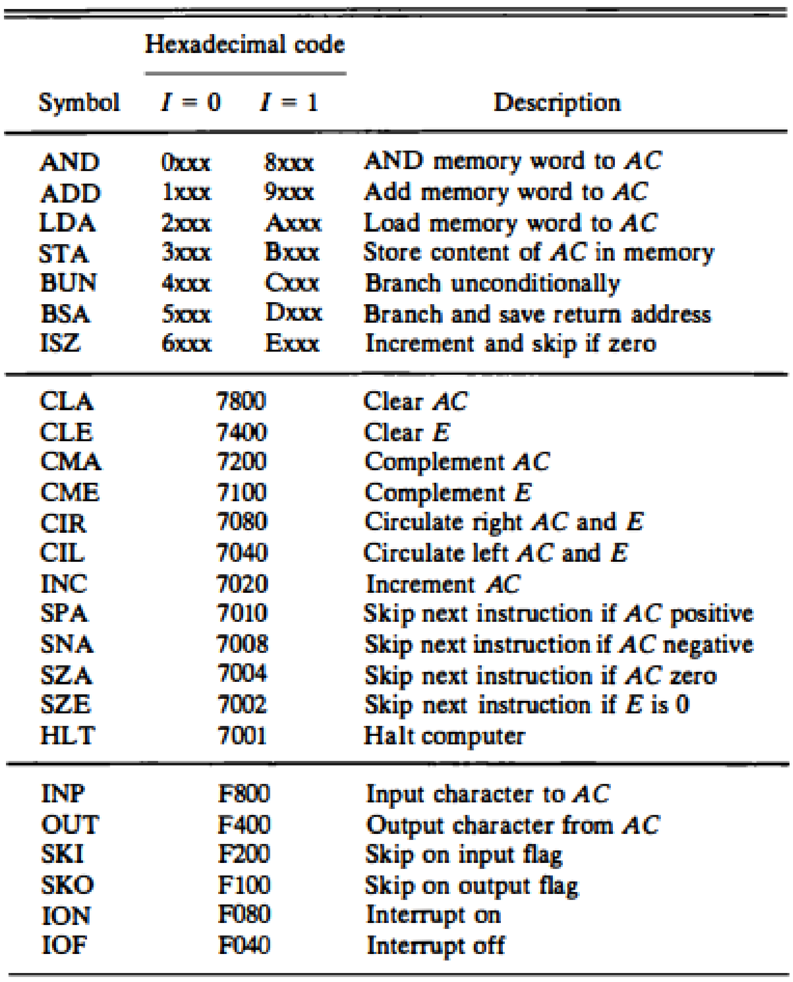
A memory-reference instruction uses 12 bits to specify an address and one bit to specify the addressing mode I. I is equal to 0 for direct address and to 1 for indirect address.

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 the AC register. So an operand from memory is not needed. Therefore, the other 12 bits are used to specify the operation 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.

The instructions for the computer are listed in Table 9.2.

Table 9.2:



	Hexadecimal code		
Symbol	$I = 0$	$I = 1$	Description
AND	0xxx	8xxx	AND memory word to *AC*
ADD	1xxx	9xxx	Add memory word to *AC*
LDA	2xxx	Axxx	Load memory word to *AC*
STA	3xxx	Bxxx	Store content of *AC* in memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear *AC*
CLE	7400		Clear *E*
CMA	7200		Complement *AC*
CME	7100		Complement *E*
CIR	7080		Circulate right *AC* and *E*
CIL	7040		Circulate left *AC* and *E*
INC	7020		Increment *AC*
SPA	7010		Skip next instruction if *AC* positive
SNA	7008		Skip next instruction if *AC* negative
SZA	7004		Skip next instruction if *AC* zero
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

The symbol designation is a three-letter word and represents an abbreviation intended for programmers and users. The hexadecimal code is equal to the equivalent hexadecimal number of the binary code used for the instruction.

Instruction Set Completeness:

A computer should have a set of instructions so that the user can construct machine language programs to evaluate any function. 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
- Data Instructions (for moving information to and from memory and processor registers)
- Program control or Brach
- Input and output instructions

There is one arithmetic instruction, ADD, and two related instructions, complement AC(CMA) and increment AC(INC). With these three instructions we can add and subtract binary numbers when negative numbers are in signed-2's complement representation.

The circulate instructions, CIR and CIL; can be used for arithmetic shifts as well as any other type of shifts desired.

There are three logic operations: AND, complement AC (CMA), and clear AC(CLA). The AND and complement provide a NAND operation.

Moving information from memory to AC is accomplished with the load AC (LDA) instruction. Storing information from AC into memory is done with the store AC (STA) instruction.

The branch instructions BUN, BSA, and ISZ, together with the four skip instructions, provide capabilities for program control and checking of status conditions.

The input (INP} and output (OUT) instructions cause information to be transferred between the computer and external devices.

9.4 TIMING AND CONTROL:

The timing for all registers in the basic computer is controlled by a master clock generator. The clock pulses are applied to all flip-flops and registers in the system, including the flip-flops and registers in the control unit. The clock pulses do not change the state of a register unless the register is enabled by a control signal.

The control signals are generated in the control unit and provide control inputs for the multiplexers in the common bus, control inputs in processor registers, and microoperations for the accumulator.

There are two major types of control organization:

- Hardwired control
- Microprogrammed control

The differences between hardwired and microprogrammed control are

Hardwired control Microprogrammed control

The control logic is implemented with gates, flip-flops, decoders, and other digital circuits. The control information is stored in a control memory. The control memory is programmed to initiate the required sequence of micro-operations.

The advantage that it can be optimized to produce a fast mode of operation. Compared with the hardwired control operation is slow. Requires changes in the wiring among the various components if the design has to be modified or changed. Required changes or modifications can be done by updating the microprogram in control memory.

The block diagram of the hardwired control unit is shown in Fig. 9.6.

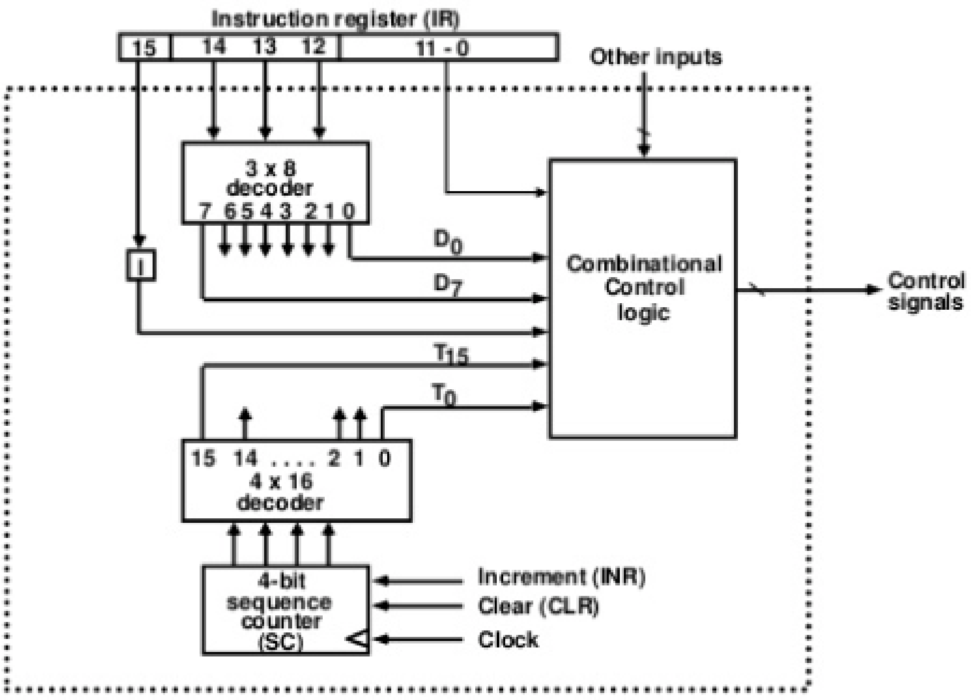


Figure 9.6: Control unit of basic computer

It consists of two decoders, a sequence counter, and a number of control logic gates. An instruction read from memory is placed in the instruction register (IR). It is divided into three parts: The I bit, the operation code, and bits 0 through 11. The operation code in bits 12 through 14 are decoded with a 3 x 8 decoder. The eight outputs of the decoder are designated by the symbols D_0 through D_9.Bit 15 of the instruction is transferred to a flip-flop designated by the symbol I. 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 the counter are decoded into 16 timing signals T_0 through T_{15}. The sequence counter SC can be incremented or cleared synchronously. The counter is incremented to provide the sequence of timing signals out of the 4 x 16 decoder.

As an example, consider the case where SC is incremented to provide timing signals T_0, T_1, T_2, T_3 and T_4 in sequence. At time T_4, SC is cleared to 0 if decoder output D_3 is active.

This is expressed symbolically by the statement

$$D_3T_4: SC \leftarrow 0$$

The timing diagram of Fig. 9.7 shows the time relationship of the control signals.

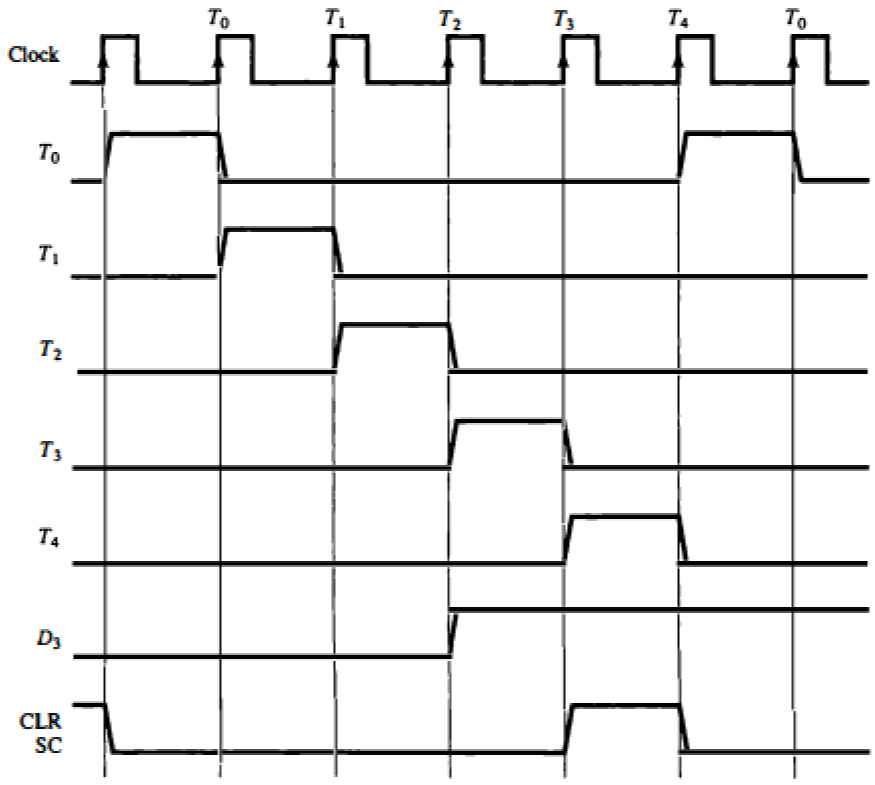


Figure 9.7: Example of control signal timings

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 signal T0 out of the decoder. T0 is active during one clock cycle. SC is incremented with every positive clock transition, unless its CLR input is active. This produces the sequence of timing signals T_0, T_1, T_2, T_3, $T_4$and so on, as shown in the diagram.

The last three waveforms in Fig.9.7 show how SC is cleared when $D_3T_4 = 1$.

Output D_3 from the operation decoder becomes active at the end of timing signal T2. 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 one marked T_4 in the diagram) 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

9.5 INSTRUCTION CYCLE:

A program residing in the memory unit of the computer consists of a sequence of instructions. The program is executed in the computer by going through a cycle for each instruction.

Each instruction cycle in turn is subdivided into a sequence of sub cycles or phases. In the basic computer each instruction cycle consists of the following phases:

1. Fetch an instruction from memory.
2. Decode the instruction.
3. Read the effective address from memory if the instruction has an indirect address.
4. Execute the instruction.

Upon the completion of step 4, the control goes back to step 1 to fetch, decode, and execute the next instruction.

Fetch and Decode:

Initially, the program counter PC is loaded with the address of the first instruction in the program. The sequence counter SC is cleared to 0, providing a decoded timing signal T0.

The micro-operations for the fetch and decode phases can be specified by the following register transfer statements.

Figure 9.8 shows how the first two register transfer statements are implemented in the bus system.

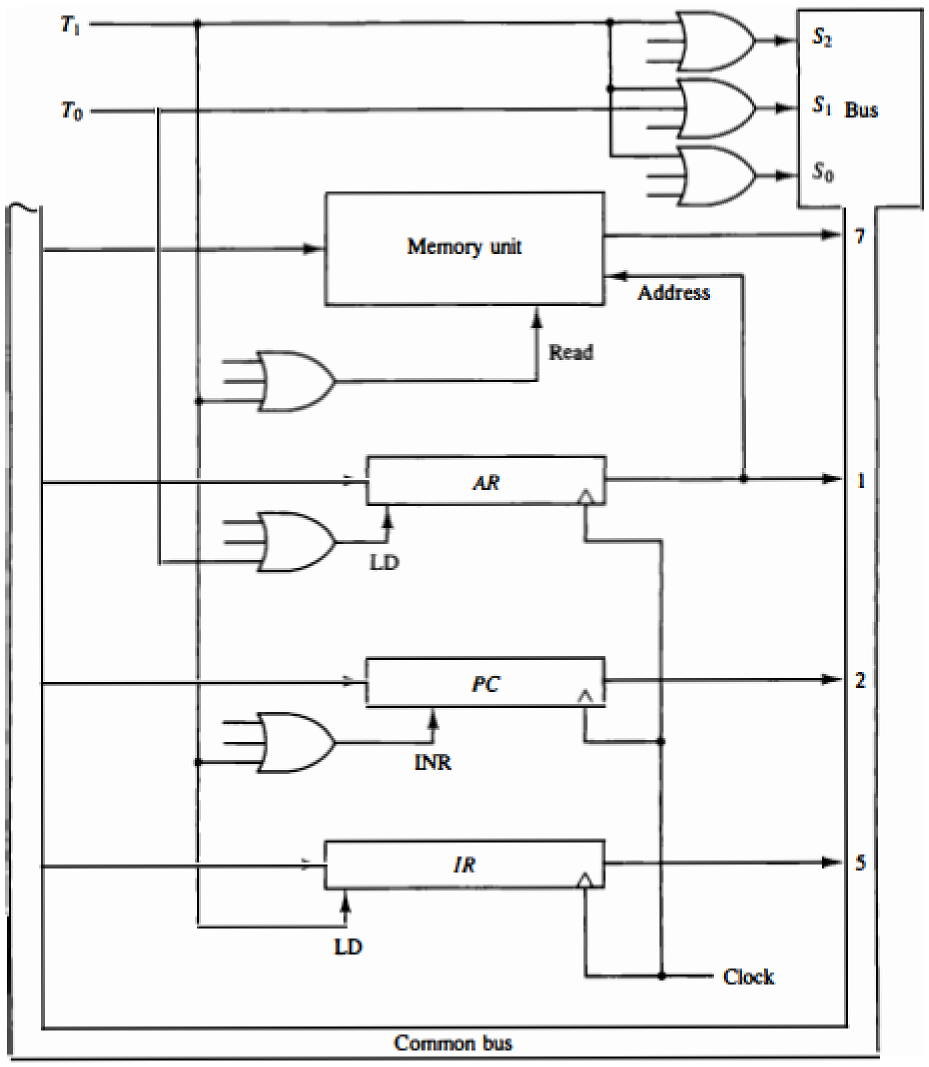


Figure 9.8 Registers Transfers for the fetch phase

To provide the data path for the transfer of PC to AR we must apply timing signal T0 to achieve the following connection:

- Place the content of PC onto the bus by making the bus selection inputs S_2, S_1, S_0 equal to 010.
- Transfer the content of the bus to AR by enabling the LD input of AR.

In order to implement the second statement it is necessary to use timing signal T_1 to provide the following connections in the bus system.

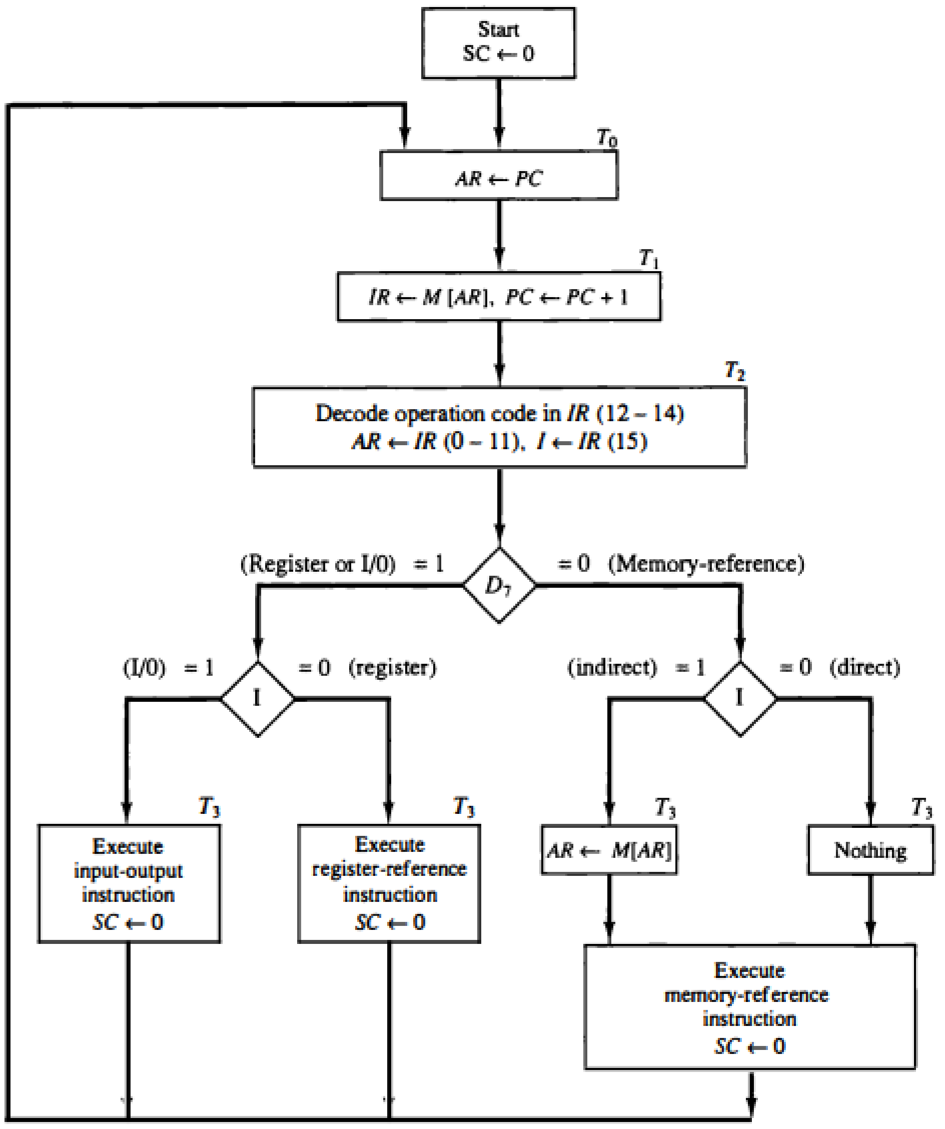
- Enable the read input of memory.
- Place the content of memory onto the bus by making $S_2S_1S_0$=111.
- Transfer the content of the bus to IR by enabling the LD input of IR.
- Increment PC by enabling the INR input of PC.

Multiple input OR gates are included in the diagram because there are other control functions that will initiate similar operations.

Determine the Type of Instruction:

The timing signal that is active after the decoding is T_3. During time T_3, the control unit determine the type of instruction that was read from the memory.

The flowchart of fig.9.9 shows the initial configurations for the instruction cycle and also how the control determines the instruction cycle type after the decoding.



Start
SC ← 0
T_0
AR ← PC
T_1
IR ← M [AR], PC ← PC + 1
T_2
Decode operation code in IR (12 – 14)
AR ← IR (0 – 11), I ← IR (15)
(Register or I/0) = 1
D_7
= 0 (Memory-reference)
(I/0) = 1
I
= 0 (register)
(indirect) = 1
I
= 0 (direct)
T_3
Execute input-output instruction SC ← 0
T_3
Execute register-reference instruction SC ← 0
T_3
AR ← M[AR]
T_3
Nothing
Execute memory-reference instruction SC ← 0

Figure 9.9: Flowchart for instruction cycle

Decoder output D_7 is equal to 1 if the operation code is equal to binary 111. If D_7=1, the instruction must be a register-reference or input-output type. If D_7 = 0, the operation code must be one of the other seven values 000 through 110, specifying a memory-reference instruction. Control then inspects the value of the first bit of the instruction, which is now available in flip-flop I.

If $D_7 = 0$ and I = 1, indicates a memory-reference instruction with an indirect address. So it is then necessary to read the effective address from memory.

If $D_7 = 0$ and I = 0, indicates a memory-reference instruction with a direct address.

If $D_7 = 1$ and I = 0, indicates a register-reference instruction.

If D_7 = 01and I = 1, indicates an input-output instruction.

The three instruction types are subdivided into four separate paths.

The selected operation is activated with the clock transition associated with timing signal T3.

This can be symbolized as follows:

Register-Reference Instructions:

Register-reference instructions are recognized by the control when D7=1 and I=0. These instructions use bits 0 through 11 of the instruction code to specify one of 12 instructions. These 12 bits are available in IR (0-11).

The control functions and micro-operations for the register-reference instructions are listed in Table 9.3.

Table 9.3:

$D_7I'T_3 = r$ (common to all register-reference instructions)
$IR(i) = B_i$ [bit in $IR(0\text{–}11)$ that specifies the operation]

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

These instructions are executed with the clock transition associated with timing variable T3. Control function needs the Boolean relation D_7I'T3, which we designate for convenience by the symbol r. By assigning the symbol Bi to bit i of IR, all control functions can be simply denoted by rBi.

For example, the instruction CLA has the hexadecimal code 7800, which gives the binary equivalent 0111 1000 0000 0000. The first bit is a zero and is equivalent to I′. The next three bits constitute the operation code and are recognized from decoder output D_9.

Bit 11 in IR is 1 and is recognized from B11. The control function that initiates the micro-operation for this instruction is $D_7I'T_3$ B_{11} = rB_{11}. The execution of a register-reference instruction is completed at time T_3. The sequence counter SC is cleared to 0 and the control goes back to fetch the next instruction with timing signal T_0. The first seven register-reference instructions perform clear, complement, circular shift, and increment micro-operations on the AC or E registers.

The next four instructions cause a skip of the next instruction in sequence when a stated condition is satisfied. The skipping of the instruction is achieved by incrementing PC once again. 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.

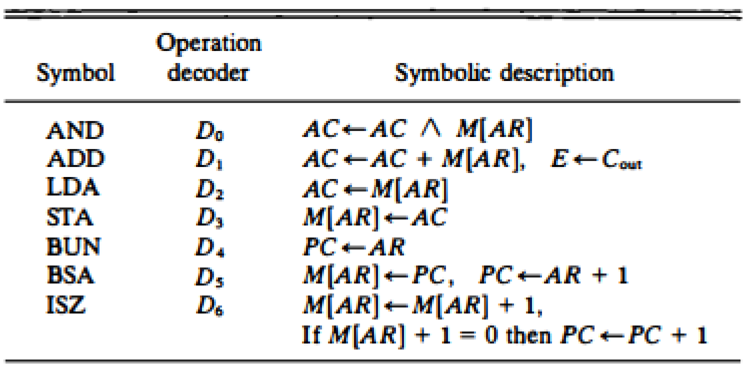
9.6 MEMORY-REFERENCE INSTRUCTIONS:

Table 9.4 lists the seven memory-reference instructions.

The decoded output D_i for i = 0, 1, 2, 3, 4, 5, and 6 from the operation decoder that belongs to each instruction is included in the table. 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 signal T_4.

The symbolic description of each instruction is specified in the table in terms of register transfer notation.

Table 9.4:



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

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.

The micro-operations that execute this instruction are:

D_0T_4: DR $\leftarrow$M[AR]

D_0T_5: AC ← AC ∧ DR, SC ← 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 Cout is transferred to the E (extended accumulator) flip-flop.

The micro-operations needed to execute this instruction are

D_1T_4: DR ←M[AR]

D_1T_5: AC ← AC + DR, E ←C_{out} ,SC← 0

LDA: Load to AC

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

The micro-operations needed to execute this instruction are

D_2T_4: DR ← M [AR]

D_2T_5: AC ←DR, SC ← 0

STA: Store AC

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

Since the output of AC is applied to the bus and the data input of memory is connected to the bus, we can execute this instruction with one micro-operation.

D_3T_4: M [AR] ← AC, SC ← 0

BUN: Branch Unconditionally

This instruction transfers the program to the instruction specified by the effective address. The BUN instruction allows the programmer to specify an instruction out of sequence and we say that the program branches (or jumps) unconditionally. The instruction is executed with one micro-operation:

D_4T_4: PC ← AR, SC ← 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.

The effective address plus one is then transferred to PC to serve as the address of the first instruction in the subroutine.

This operation was specified with the following register transfer:

M[AR] ← PC, PC ←AR + I

A numerical example that demonstrates how this instruction is used with a subroutine is shown in Fig. 9.10.

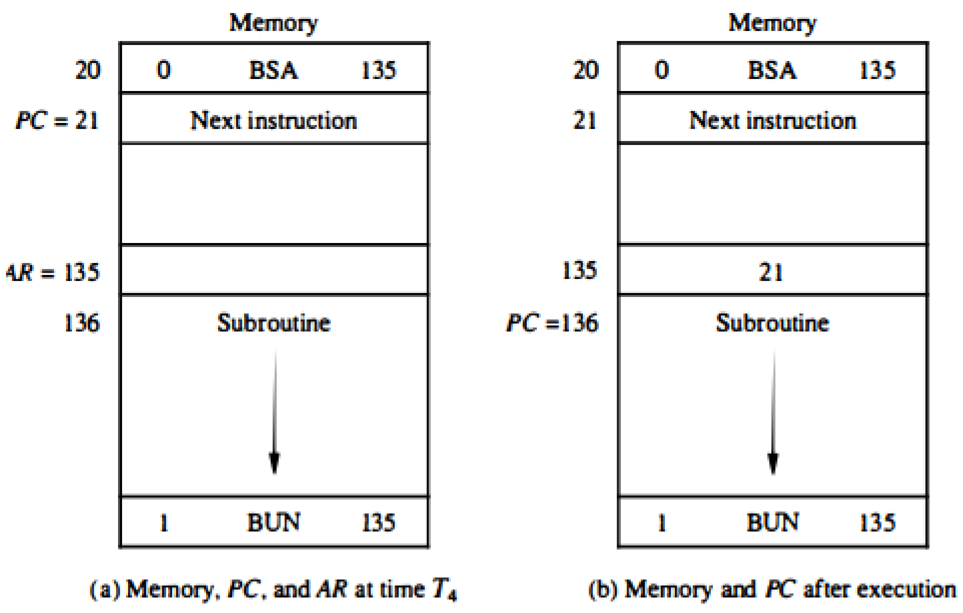


Figure 9.10: Example of BSA instruction execution

The BSA instruction is assumed to be in memory at address 20. The I bit is 0 and the address part of the instruction has the binary equivalent of 139.

After the fetch and decode phases, PC contains 21, which is the address of the next instruction in the program (referred to as the return address). AR holds the effective address 139. This is shown in part (a) of the figure. The BSA instruction performs the following numerical operation:

The result of this operation is shown in part (b) of the figure. The return address21 is stored in memory location 135 and control continues with the subroutine program starting from address 136. The return to the original program (at address 21) is accomplished by means of an indirect BUN instruction placed at the end of the subroutine.

When this instruction is executed, control goes to the indirect phase to read the effective address at location 135, where it finds the previously saved address 21. When the BUN instruction is executed, the effective address 21 is transferred to PC. The next instruction cycle finds PC with the value 21, so control continues to execute the instruction at the return address.

The BSA instruction must be executed with a sequence of two micro-operations:

ISZ: Increment and Skip if Zero

This instruction increment the word specified by the effective address, and if the incremented value is equal to 0, PC is incremented by 1 to skip the next instruction in the program. 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. This is done with the following sequence of micro-operations:

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

Control Flowchart:

A flowchart showing all micro-operations for the execution of the seven memory-reference instructions is shown in Fig. 9.11.

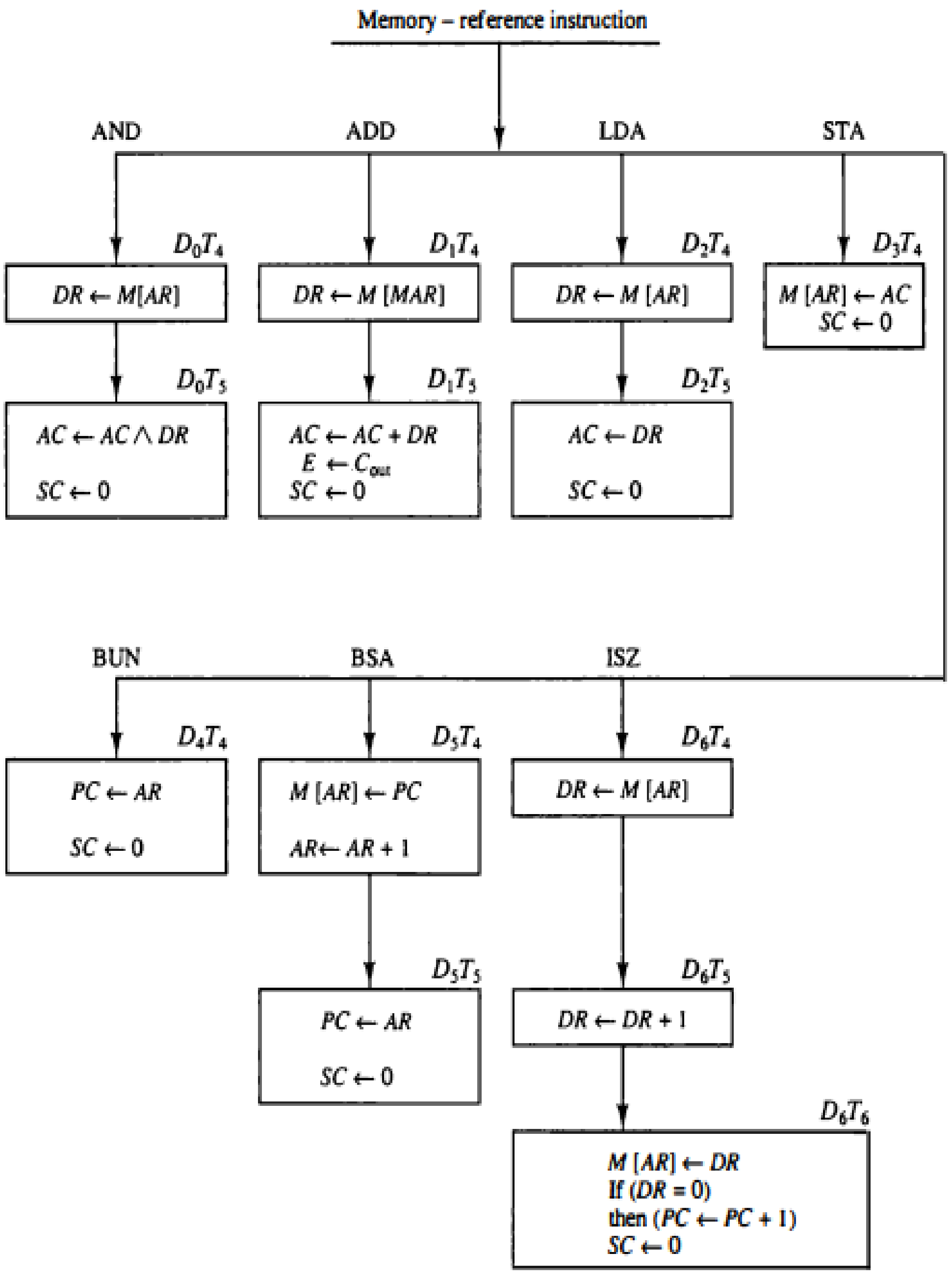


Figure 9.11: Flowchart for memory reference instructions

9.9 INPUT-OUTPUT AND INTERRUPT:

Instructions and data stored in memory must come from some input device. Computational results must be transmitted to the user through some output device. To demonstrate the most basic requirements for input and output communication, we will use as an illustration a terminal unit with a keyboard and printer.

Input-Output Configuration:

The terminal sends and receives serial information. 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 input—output configuration is shown in Fig. 9.12.

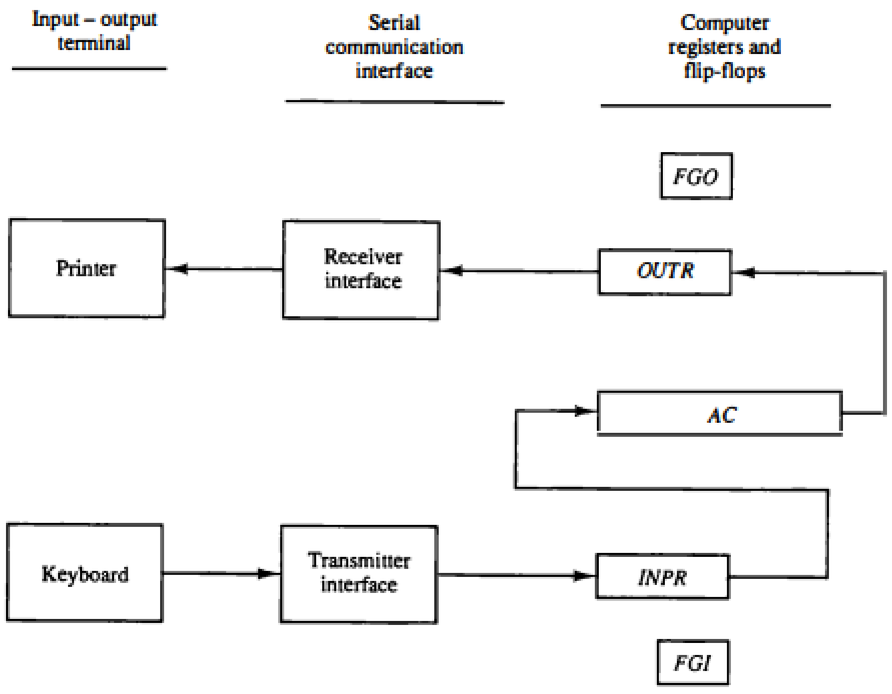


Figure 9.12 Input-output configuration

The input register INPR consists of eight bits and holds alphanumeric input information. The 1-bit input flag FGI is a control flip-flop. The flag bit 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 output register OUTR works similarly but the direction of information flow is 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.

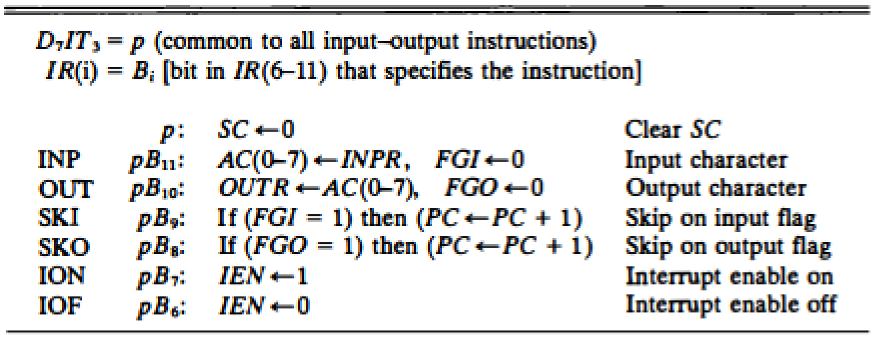
The output device accepts the coded information, prints the corresponding character, and when the operation is completed, it sets FGO to 1.

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-output instructions have an operation code 1111 and are recognized by the control when D_7 = 1 and I = 1. The remaining bits of the instruction specify the particular operation.

The control functions and micro-operations for the input-output instructions are listed in Table 9.5.

Table 9.5 Input-output instructions



$D_7IT_3 = p$ (common to all input–output instructions)
$IR(i) = B_i$ [bit in $IR(6–11)$ that specifies the instruction]

	p:	$SC \leftarrow 0$	Clear SC
INP	pB_{11}:	$AC(0–7) \leftarrow INPR$, $FGI \leftarrow 0$	Input character
OUT	pB_{10}:	$OUTR \leftarrow AC(0–7)$, $FGO \leftarrow 0$	Output character
SKI	pB_9:	If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$	Skip on input flag
SKO	pB_8:	If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$	Skip on output flag
ION	pB_7:	$IEN \leftarrow 1$	Interrupt enable on
IOF	pB_6:	$IEN \leftarrow 0$	Interrupt enable off

These instructions are executed with the clock transition associated with timing signal T_3. Each control function needs a Boolean relation D_7IT_3, which we designate for convenience by the symbol p.

The control function is distinguished by one of the bits in IR (6-11). By assigning the symbol Bi to bit i of IR, all control functions can be denoted by pBi for i = 6 though 11. The sequence counter SC is cleared to 0 when p = D_7IT_3 = 1. The last two instructions set and clear an interrupt enable flip-flop IEN.

Program Interrupt:

The computer keeps checking the flag bit, and when it finds it set, it initiates an information transfer. The difference of information flow rate between the computer and that of the input—output device makes this type of transfer inefficient.

An alternative to the programmed controlled procedure is to let the external device inform the computer when it is ready for the transfer. In the meantime the computer can be busy with other tasks. This type of transfer uses the interrupt facility.

While the computer is running a program, it does not check the flags. When a flag is set, the computer is momentarily interrupted from the current program. The computer deviates momentarily from what it is doing to perform of the input or output transfer. It then returns to the current program to continue what it was doing before the interrupt.

The interrupt enable flip-flop IEN can be set and cleared with two instructions.

- When IEN is cleared to 0 (with the IOF instruction), the flags cannot interrupt the computer.
- When IEN is set to (with the ION instruction), the computer can be interrupted.

The way that the interrupt is handled by the computer can be explained by means of the flowchart of Fig. 9.13.

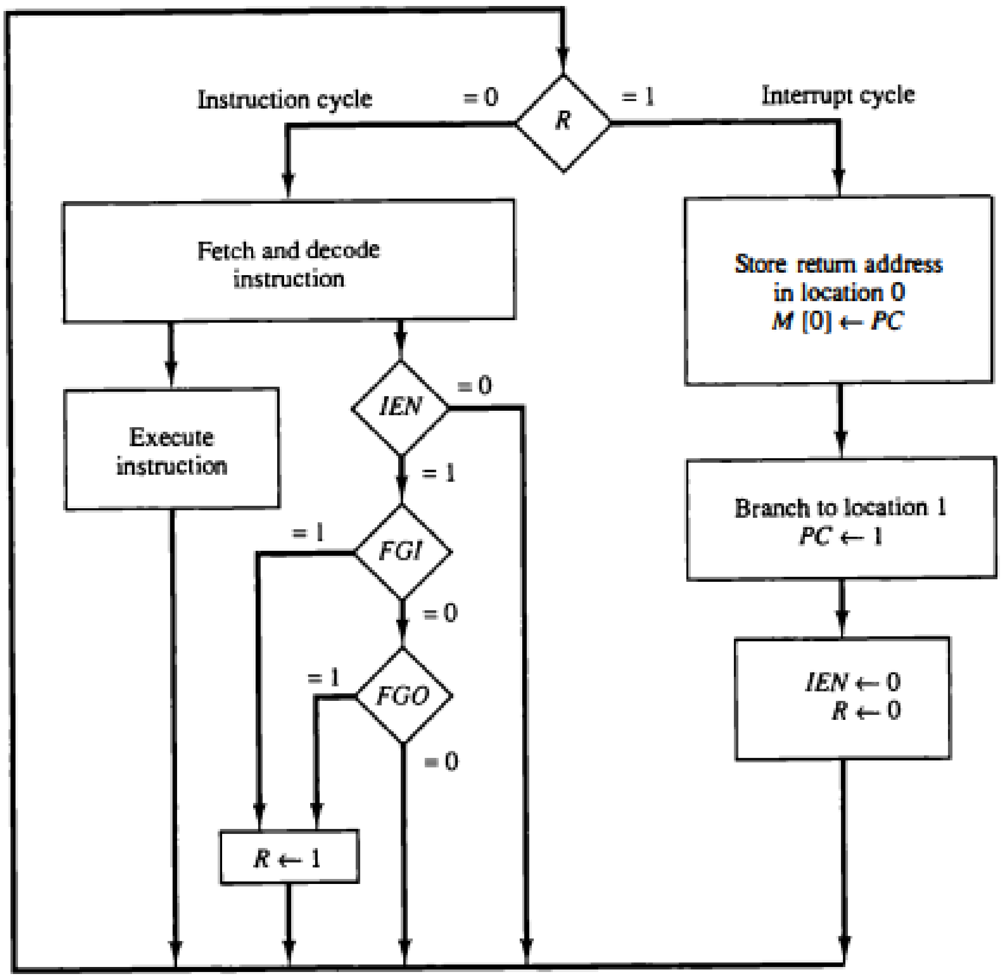


Figure 9.13: Flowchart for interrupt cycle

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 1EN = 1, flip-flop R is set to 1. 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.

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. This location may be a processor register, a memory stack, or a specific memory location.

$$T_0T_1T_2(IEN)\ (FGI + FGO) : R \leftarrow 1$$

An example that shows what happens during the interrupt cycle is shown in Fig. 9.14.

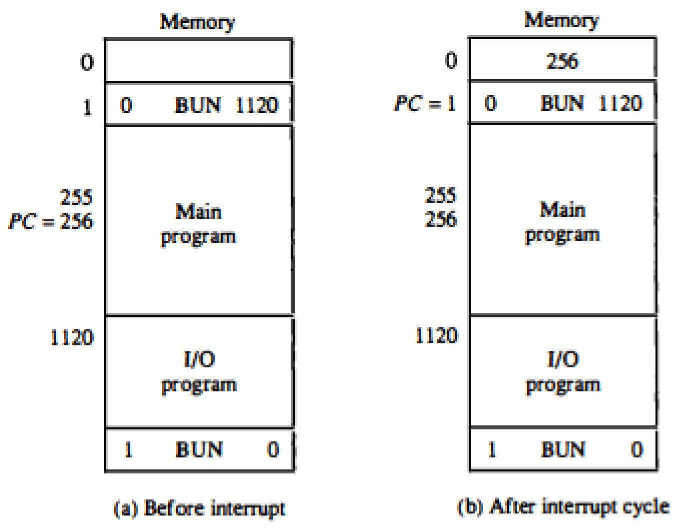


Figure 9.14: Demonstration of the interrupt cycle

When an interrupt occurs and R is set to 1 while the control is executing the instruction at address 259. At this time, the returns 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. This is shown in Fig. 9.14(a).

When control reaches timing signal T0and finds that R = 1, it proceeds with the interrupt cycle. The content of PC (256) is stored in memory location 0, PC is set to 1, and R is cleared to 0. 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. This is shown in Fig. 9.14(b).

9.8 SUMMARY:

The chapter on Basic Computer Organization and Design delves into the fundamental aspects of how computers execute operations and manage data. It begins by exploring instruction codes, which are essential for defining operations performed by the CPU. The chapter then examines various types of computer registers that temporarily hold data and instructions, followed by a discussion on the different categories of computer instructions such as data transfer, arithmetic, and control instructions. The importance of timing and control mechanisms is highlighted, detailing how control signals synchronize and manage the flow of data and instructions. The instruction cycle, which encompasses the phases of fetching, decoding, and executing instructions, is analyzed to illustrate the step-by-step execution within the CPU. The chapter also covers memory reference instructions, explaining how the CPU interacts with memory, and delves into input-output operations and the role of interrupts in managing external devices and efficient data transfer. This comprehensive overview provides a foundational understanding of the internal workings of a computer system.

9.9 TECHNICAL TERMS:

Operation code, Instruction code, Accumulator, Computer registers, Timing & control.

9.10 SELF ASSESSMENT QUESTIONS:

Essay questions:

1) Explain how a computer decodes and executes an arithmetic instruction?
2) Describe the role of timing signals in CPU operations and explain in detail?
3) Outline the phases of the instruction cycle and describe what happens in each phase?
4) What are memory reference instructions? Provide examples and explain their functions?
5) Describe the concept of interrupt priority. How does it affect the handling of multiple interrupts?

Short Questions:

1) Define an instruction code. What are the key components of an instruction code?
2) Differentiate between a direct and an indirect instruction code with examples
3) Explain the role of the program counter (PC) and instruction register (IR) during the instruction cycle.
4) Compare and contrast programmed I/O, interrupt-driven I/O, and direct memory access (DMA).

9.11 FURTHER READINGS:

1) M.Morris Mano, “Computer System Architecture”, 3rd Edition, Pearson Education (2008).
2) V. Rajaraman, T. Radha Krishnan, “Computer Organization and Architecture”, PHI.
3) BehroozParhami, “Computer Architecture”, Oxford (2007).
4) ISRD group, “Computer Organization”, ace series, TMH (2007).
5) William Stallings, “Computer Organization and Architecture–Designing for Performance”, Pearson Education (2005).

Dr. Kampa Lavanya

LESSON-10

MICROPROGRAMMED CONTROL

OBJECTIVES:

After going through this lesson, you will be able to

- Describe the structure and content of control memory.
- Explain the concept of address sequencing.
- Examine a specific microprogram example.
- Describe the steps involved in designing a microprogrammed control unit.
- Simulate the execution of a microprogram.

STRUCTURE OF THE LESSION:

10.1 CONTROL 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 μops) 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 10.1. The control memory is assumed to be a ROM, within which all control information is permanently stored.

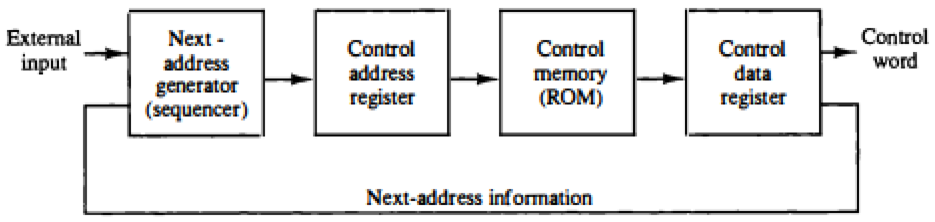


Figure 10.1: Micro-programmed control organization

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 microoperations 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 may be 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 *pipeline 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.

10.2 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 the operation 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.

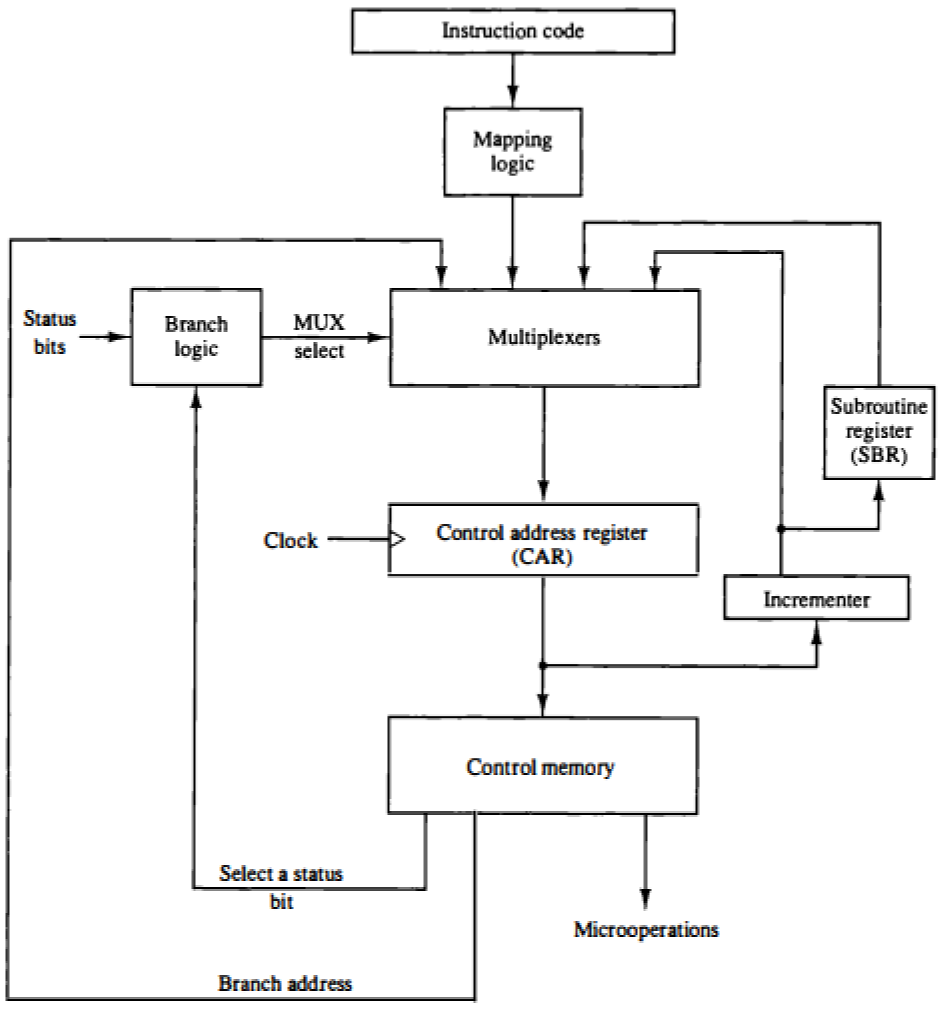


Figure 10.2: Selection of address for control memory

Conditional Branching

Above Figure 10.2 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 10.2 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 microprogram 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 10.3 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 10.bit address for control memory is shown in Figure 10.3.

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 addresses 1000000 through 1111111. If it uses fewer than four microinstructions, the unused memory locations would be available for other routines.

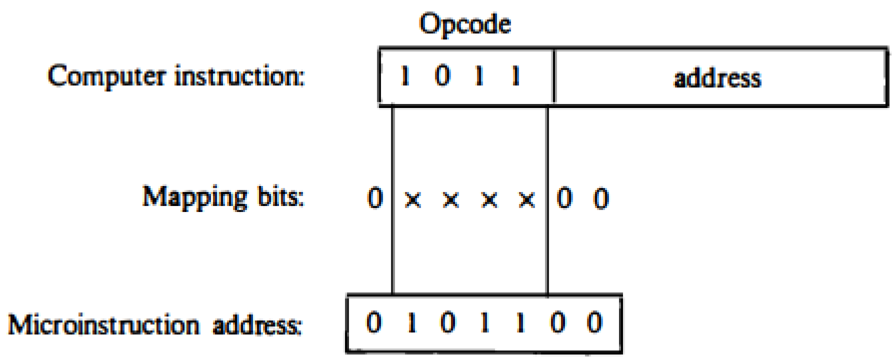


Figure 10.3: Mapping from instruction code to microinstruction address

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.

10.3 MICROPROGRAM EXAMPLE:

Once the configuration of a computer and its microprogrammed control unit is established, the designer's task is to generate the microcode for the control memory. This code generation is called microprogramming and is a process similar to conventional machine language programming.

Computer Configuration:

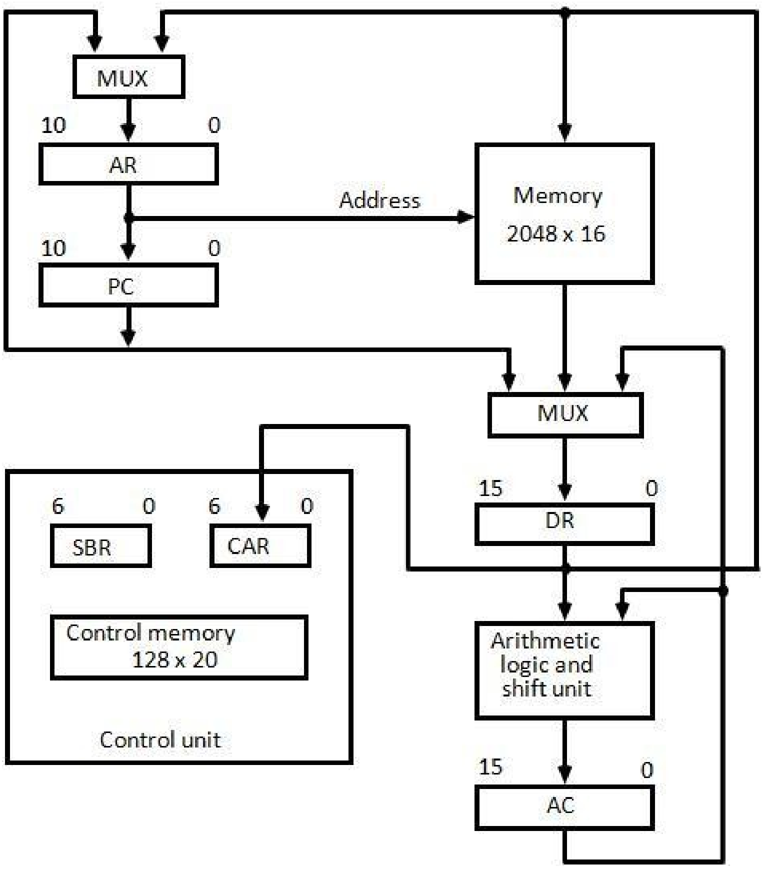


Figure 10.4: Computer hardware configuration

The block diagram of the computer is shown in Figure 10.4. 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.

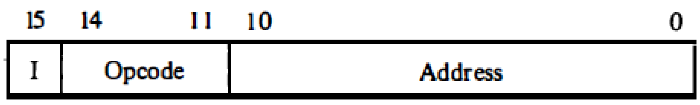
4) ALU: The arithmetic, logic, and shift unit performs microoperations with data from AC and DR and places the result in AC.

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:

The microinstruction format for the control memory is shown in Figure 10.5. The 20 bits of the microinstruction are divided into four functional parts as follows:

- 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.
- The CD field selects status bit conditions.
- The BR field specifies the type of branch to be used.
- The AD field contains a branch address. The address field is seven bits wide, since the control memory has 128 = 2^7 words.

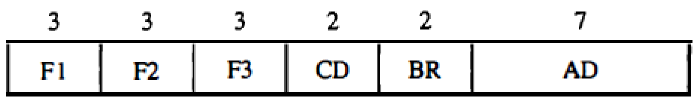


(a) Instruction format

Symbol	Opcode	Description
ADD	0000	$AC \leftarrow AC + M[EA]$
BRANCH	0001	If $(AC < 0)$ then $(PC \leftarrow EA)$
STORE	0010	$M[EA] \leftarrow AC$
EXCHANGE	0011	$AC \leftarrow M[EA], M[EA] \leftarrow AC$

EA is the effective address

(b) Four computer instructions



F1, F2, F3: Microoperation fields

CD: Condition for branching

BR: Branch field

AD: Address field

Figure 10.5: Microinstruction Format

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

DR←M[AR] with F2 = 100 , PC ←PC + 1 with F3 = 101

The nine bits of the microoperation fields will then be 000 100 101. The CD (condition) field consists of two bits which are encoded to specify four status bit conditions as listed in Table 10.1.

Table 10.1: Condition Field

F1	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC + DR$	ADD
010	$AC \leftarrow 0$	CLRAC
011	$AC \leftarrow AC + 1$	INCAC
100	$AC \leftarrow DR$	DRTAC
101	$AR \leftarrow DR(0\text{–}10)$	DRTAR
110	$AR \leftarrow PC$	PCTAR
111	$M[AR] \leftarrow DR$	WRITE

F2	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC - DR$	SUB
010	$AC \leftarrow AC \vee DR$	OR
011	$AC \leftarrow AC \wedge DR$	AND
100	$DR \leftarrow M[AR]$	READ
101	$DR \leftarrow AC$	ACTDR
110	$DR \leftarrow DR + 1$	INCDR
111	$DR(0\text{-}10) \leftarrow PC$	PCTDR

F3	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC \oplus DR$	XOR
010	$AC \leftarrow \overline{AC}$	COM
011	$AC \leftarrow$ shl AC	SHL
100	$AC \leftarrow$ shr AC	SHR
101	$PC \leftarrow PC + 1$	INCPC
110	$PC \leftarrow AR$	ARTPC
111	Reserved	

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

BR	Symbol	Function
00	JMP	$CAR \leftarrow AD$ if condition = 1
		$CAR \leftarrow CAR + 1$ if condition = 0
01	CALL	$CAR \leftarrow AD$, $SBR \leftarrow CAR + 1$ if condition = 1
		$CAR \leftarrow CAR + 1$ if condition = 0
10	RET	$CAR \leftarrow SBR$ (Return from subroutine)
11	MAP	$CAR(2\text{–}5) \leftarrow DR(11\text{–}14)$, $CAR(0,1,6) \leftarrow 0$

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 10.2.

Table 10.2: Branch Field

Binary Address	F1	F2	F3	CD	BR	AD
1000000	110	000	000	00	00	1000001
1000001	000	100	101	00	00	1000010
1000010	101	000	000	00	11	0000000

Symbolic Micro Instructions:

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 10.3.

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 inTable 10.2.
5) AD: The AD field specifies a value for the address field of the microinstruction in one of three possible ways:
   i. With a symbolic address, this must also appear as a label.
   ii. With the symbol NEXT to designate the next address in sequence.
   iii. 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 10.3: Symbolic Microinstruction

Label	Microoperations	CD	BR	AD
	ORG 0			
ADD:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ADD	U	JMP	FETCH
	ORG 4			
BRANCH:	NOP	S	JMP	OVER
	NOP	U	JMP	FETCH
OVER:	NOP	I	CALL	INDRCT
	ARTPC	U	JMP	FETCH
	ORG 8			
STORE:	NOP	I	CALL	INDRCT
	ACTDR	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 12			
EXCHANGE:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ACTDR, DRTAC	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 64			
FETCH:	PCTAR	U	JMP	NEXT
	READ, INCPC	U	JMP	NEXT
	DRTAR	U	MAP	
INDRCT:	READ	U	JMP	NEXT
	DRTAR	U	RET	

10.4 DESIGN OF CONTROL UNIT:

The bits of the microinstruction are usually divided into fields, with each field defining a distinct, separate function. The various fields encountered in 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. The number of control bits that initiate Microoperations can be reduced by grouping mutually exclusive variables into fields and encoding the k bits in each field to provide 2^kmicrooperations. Each field requires a decoder to produce the corresponding control signals. This method reduces the size of the microinstruction bits but requires additional hardware external to the control memory. It also increases the delay time of the control signals because they must propagate through the decoding circuits.

The encoding of control bits was demonstrated in the programming example of the preceding section. The nine bits of the microoperation field are divided into three subfields of three bits each. The control memory output of each subfield must be decoded to provide the distinct microoperations. The outputs of the decoders are connected to the appropriate inputs in the processor unit.

Figure 10.7 shows the three decoders and some of the connections that must be made from their outputs. Each of the three fields of the microinstruction presently available in the output of control memory are decoded with a 3 X 8 decoder to provide eight outputs. Each of these outputs must be connected to the proper circuit to initiate the corresponding microoperation as specified in Table 10.1. For example, when F1 = 101(binary 5), the next clock pulse transition transfers the content of DR (0-10) to AR (symbolized by DRTAR in Table 10.1). Similarly, when F1=110(binary 6) there is a transfer from PC to AR (symbolized by PCTAR). As shown in Fig. 10.7, outputs 5 and 6 of decoder F1are connected to the load input of AR so that when either one of these outputs is active, information from the multiplexers is transferred to AR. The multiplexers select the information from DR when output 5 is active and from PC when output 5 is inactive. The transfer into AR occurs with a clock pulse transition only when output 5 or output 6 of the decoder are active. The other outputs of to decoders that initiate transfers between registers must be connected in a similar fashion.

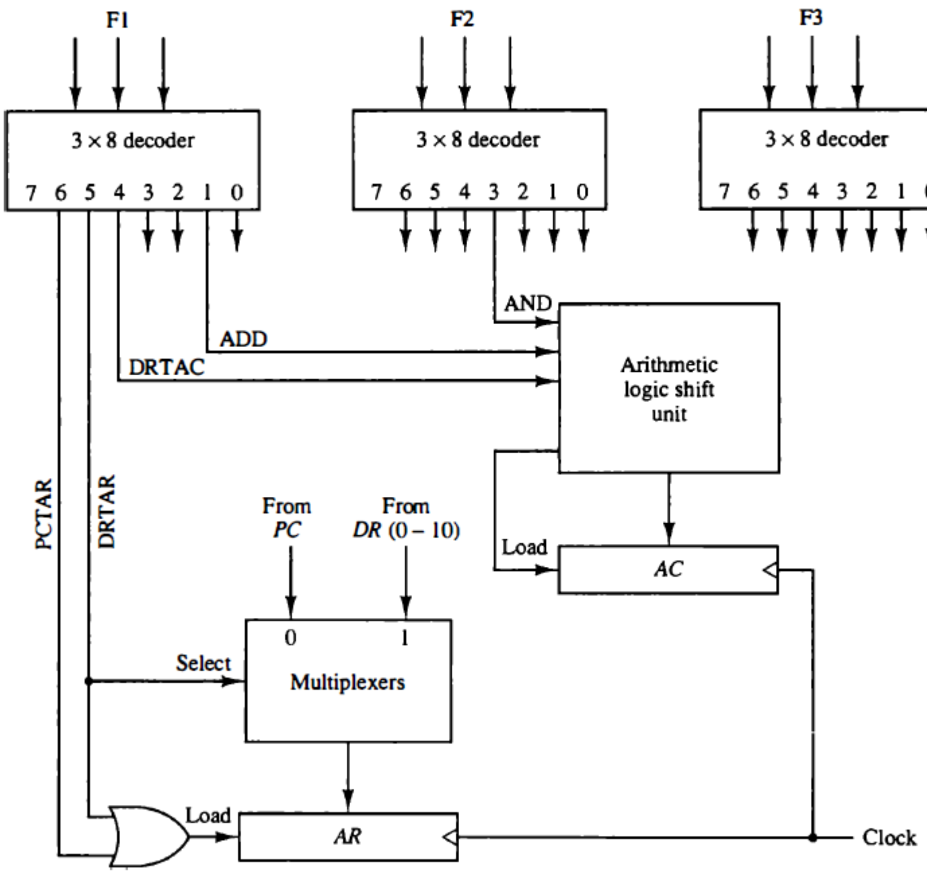


Figure 10.6 Design of microprocessor fields

Microprogram Sequencer:

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. However, just as there are large ROM units available in integrated circuit packages, so are general-purpose sequencers suited for the construction of microprogram control units. 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. The next-address logic of the sequencer determines the specific address source to be loaded into the control address register. The choice of the address source is guided by the next-address information bits that the sequencer receives from the present microinstruction. 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.

To illustrate the internal structure of a typical microprogram sequencer we will show a particular unit that is suitable for use in the microprogram computer example developed in the preceding section. The block diagram of the microprogram sequencer is shown in Fig. 10.10. The control memory is included in the diagram to show the interaction between the sequencer and the memory attached to it. 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 test is 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 register SBR. The other three inputs to multiplexer number 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 diagram 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. A push and pop operation, in conjunction with a stack pointer, stores and retrieves the return address during the call and return microinstructions.

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 go to an input logic circuit. The input logic in a particular sequencer will determine the type of operations that are available in the unit. 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.

The input logic circuit in Fig. 10.8 has three inputs, I_0, I_1, and T and three outputs, S_0, S_1, and L. Variables So and S, select one of the source addresses for CAR. Variable L enables the load input in SBR. The binary values of the two selection variables determine the path in the multiplexer. For example, with S_1S_0 = 10, multiplexer input number 2 is selected and establishes a transfer path from SBR to CAR. Note that each of the four inputs as well as the output of MUX 1 contains a 10.bit address.

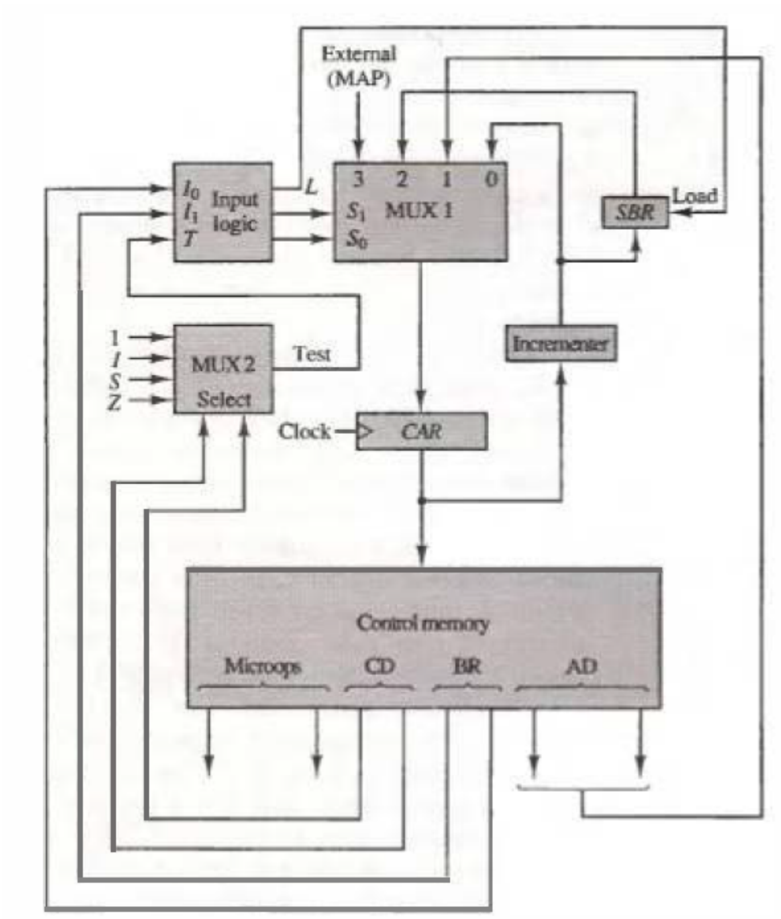


Figure 10.7 Microprogram sequencer for control memory

The truth table for the input logic circuit is shown in Table 10.4. Inputs I_1 and I_0 are Identical to the bit values in the BR field. The function listed in each entry was defined in Table 10.1. The bit values for S_1 and S_0 are determined from the stated function and the path in the multiplexer that establishes the required transfer. The subroutine register is loaded with the incremented value of CAR during a call microinstruction (BR = 01) provided that the status bit condition is satisfied (T = 1). The truth table can be used to obtain the simplified Boo lean functions for the input logic circuit:

$S_1 = I_1$

$S_0 = I_1I_0 + I'_1T$

$L = I'_1I_0T$

Table 10.4 Input logic Truth Table for Microprogram Sequencer

BR Fiel	Input I_1 I_0 T	MUX 1 S_1 S_0	Load SBR 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 ×	1 0	0
1 1	1 1 ×	1 1	0

Note that the incrementer circuit in the sequencer of Fig. 10.8 is not a counter constructed with flip-flops but rather a combinational circuit constructed with gates. A combinational circuit incrementer can be designed by cascading a series of half-adder circuits. The output carry from one stage must be applied to the input of the next stage. One input in the first least significant stage must be equal to 1 to provide the increment-by-one operation.

10.5 SUMMARY:

In this, the focus is on the intricacies of control unit design within digital systems. It covers fundamental topics such as the role and structure of control memory, which stores microinstructions that control the operation of the system's components. Address sequencing within this memory is explored in detail, illustrating how instructions are fetched and executed in a sequence that directs the flow of operations. A detailed micro program example is provided to demonstrate how these concepts are applied in practice, emphasizing the design principles behind efficient and effective control unit architectures.

10.6 TECHNICAL TERMS:

Control Memory, Address Sequencing, Micro Program, Control Unit.

10.7 SELF ASSESSMENT QUESTIONS:

Essay questions:

1) Demonstrate the general configuration of Micro programmed Control unit with a neat block diagram.
2) Present a Simple digital computer and show how it can be micro programmed with the help of necessary formats and notations.
3) Illustrate the phases involved in decoding of micro operation fields with necessary diagrams.
4) Draw and explain the schematic of Micro program sequencer for a control memo.

Short Questions:

1) Explain about address sequencing in control memory with neat diagrams?
2) What is the difference between a micro-processor and a micro program?
3) Differentiate Hardwired and Micro programmed control unit.
4) Define (i) Micro operation (ii) Microinstruction (iii) Micro program.

10.8 FURTHER READINGS:

1) M.Morris Mano, “Computer System Architecture”, 3rd Edition, Pearson Education (2008).
2) V. Rajaraman, T. Radha Krishnan, “Computer Organization and Architecture”, PHI.
3) Behrooz Parhami, “Computer Architecture”, Oxford (2007).
4) ISRD group, “Computer Organization”, ace series, TMH (2007).
5) William Stallings, “Computer Organization and Architecture-Designing for Performance”, Pearson Education (2005).

Dr. Kampa Lavanya

LESSON-11

CENTRAL PROCESSING UNIT

OBJECTIVES:

After going through this lesson, you will be able to

- Understanding the role of the CPU
- Exploring how registers are structured within the CPU
- Examining the stack-based memory structure
- Exploring how the CPU handles data movement between registers, memory, and I/O devices
- Investigating mechanisms for controlling program flow and execution

STRUCTURE OF THE LESSION:

11.1 INTRODUCTION:

The part of the computer that performs the bulk of data-processing operations is called the central proofing unit and is refeired to as the CPU. The CPU is made up of three major parts, as shown in Figure.

The register set stoics intermediate data used during the execution of the instructions. The arithmetic logic unit (ALU) perfoims the required mkraoperations for executing the instructions. The control unit supervises the transfer of information among the registers and instructs the ALU as to which operation to perform. The CPU performs a variety of functions dictated by the type of instructions that are incorporated in the computer.

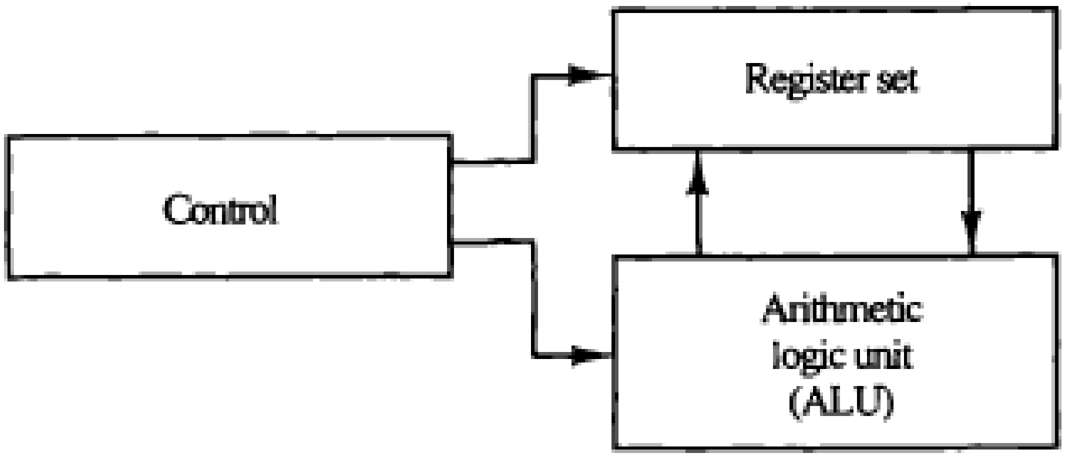


Figure 11.1 major components of the CPU

Computer architecture is sometimes defined as the computer structure and behavior as seen by the programmer that uses machine language instructions. This includes the instruction formats, addressing modes, the instruction set, and the general organization of the CPU registers. One boundary where the computer designer and the computer programmer see the same machine is the part of the CPU associated with the instruction set. From the designer's point of view the computer induction set provides the specifications for the design of the CPU.

The design of a CPU is a task that in large part involves choosing the hardware for implementing the machine instructions. The user who programs the computer in machine or assembly language must be aware of the register set, the memory structure, the type of data supported by the instructions, and the function that each instruction performs.

11.2 GENERAL REGISTER ORGANIZATION:

In a CPU, some registers are very specific to their task hence they are used very frequently by the control unit. Some anonymous registers are also used and operations of theses registers are subject to the need and type of information stored in them.

Following is an organization of these registers in a CPU:

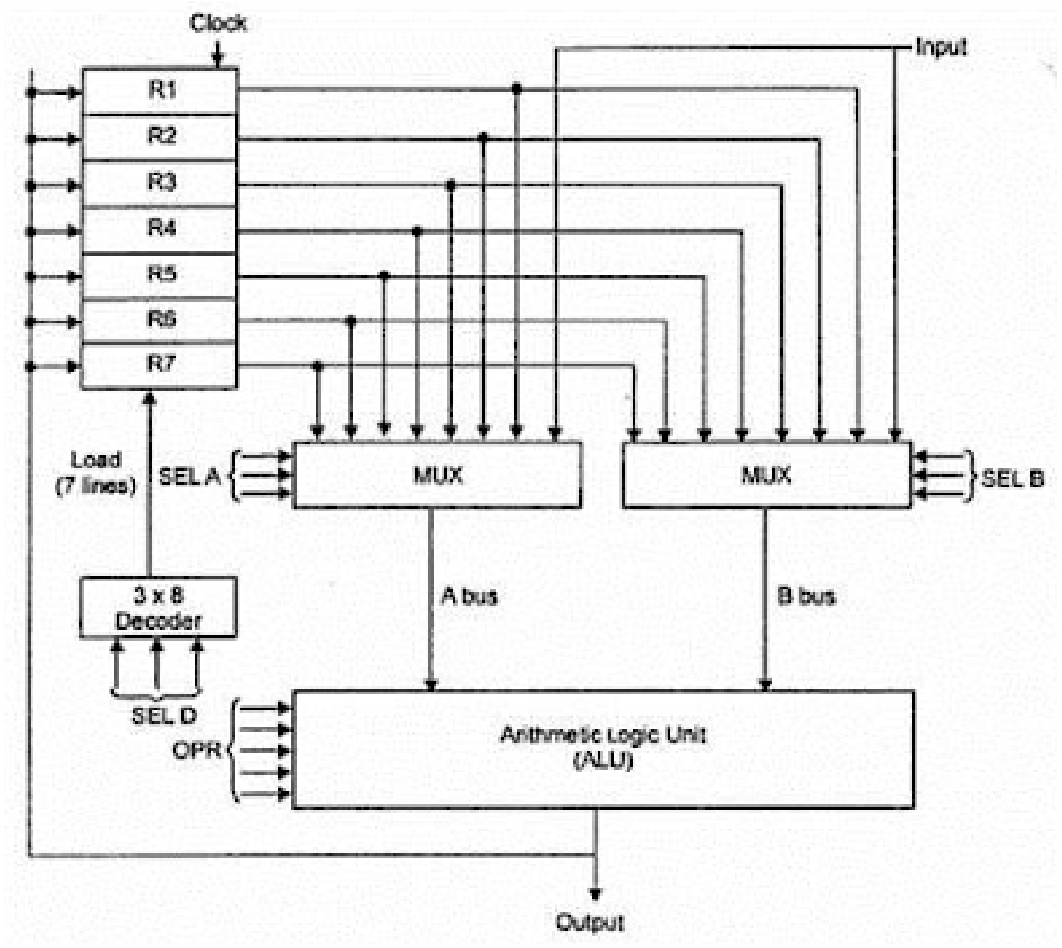


Figure 11.2 Register set with common ALU

As the diagram depicts, a block of seven registers is used for storing anonymous data. Two multiplexers of 8 to 1 size are used to extract the contents of two registers at a time. Here MUXes are of 8 to 1 size because we have another line for I/O devices along with seven registers. The two extracted values are processed in AC with the help of ALU. Here to specify the appropriate operation, five lines of OPR input are used. The computed output is then ANDed with the outputs of a 3 to 8 decoder. Since a decoder activates only one output line at a time, the computed output goes in only one register. The appropriate destination register is specified by the 3 input lines of decoder.

For example, to perform the operation

$$R_1 \leftarrow R_2 + R_3$$

The control must provide binary selection variables to the following selector inputs:

1. MUX A selector (SELA): to place the content of R2 into bus A.

2. MUX B selector (SELB): to place the content o f R 3 into bus B.

3. ALU operation selector (OPR): to provide the arithmetic addition A + B.

4. Decoder destination selector (SELD): to transfer the content of the output bus into R_1.

Control Word:

The unit has 14 binary selection inputs, each of which defines a control word when their values are added together. Figure 11.3 shows the 14-bit control word (b). There are four fields in all. Each of the three fields has three bits, while one field has five bits. The three bits of SELA are used to pick a source register for the ALU's A input. The three bits of SELB are used to choose a register for the ALU's B input. Using the decoder's seven load outputs, the three bits of SELD choose a destination register. The five bits of OPR are used to pick one of the ALU's operations. When applied to the selection inputs, the 14-bit control word specifies a specific micro operation.

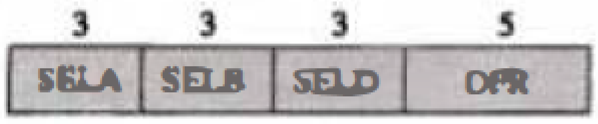


Figure 11.3 Control Word

Table 11.1 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

The binary code for each of the three fields is specified by the 3-bit binary code stated in the table's first column. The register chosen by fields SELA, SELB, and SELD is the one whose decimal number equals the code's binary number. When SELA or SELB is set to 000, the external input data is selected by the appropriate multiplexer. No destination register is selected when SELD = 000, however the contents of the output bus are available in the external output. The ALU is a computer that performs arithmetic and logic operations. Furthermore, the CPU must do shift operations. The shifter can be connected to the ALU's input to give preshifting capabilities or to the ALU's output to provide postshifting capacity. The shift operations are sometimes incorporated with the ALU. Table 11.2 shows the function table for this ALU. Table 11.2 specifies the encoding of ALU operations for the CPU. Each operation is given a symbolic name in the OPR field, which contains five bits.

Table 11.2: 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

Examples of Microoperations:

A control word of 14 bits is needed to specify a microoperation in the CPU. The control word for a given microoperation can be derived from the selection variables. For example, the subtract rnicrooperation given by the statement R1 ← R2 - R3 specifies R2 for the A input of the ALU, R3 for the B input of the ALU, R1 for the destination register, and an ALU operation to subtract A - B.

Thus the control word is specified by the four fields and the corresponding binary value for each field is obtained from the encoding listed in Tables 11.1 and 11.2.

The binary control word for the subtract rnicrooperation is 010 011 001 00101 and is obtained as follows:

Field:	SELA	SELB	SELD	OPR
Symbol:	R2	R3	R1	SUB
Control word:	010	011	001	00101

The control word for this rnicrooperation and a few others are listed in Table 11.3.

The increment and transfer microoperations do not use the B input of the ALU.

For these cases, the B field is marked with a dash. We assign 000 to any unused field when formulating the binary control word, although any other binary number may be used.

To place the content of a register into the output terminals we place the content of the register into the A input of the ALU, but none of the registers are selected to accept the data.

The ALU operation TSFA places the data from the register, through the ALU, into the output terminals.

The direct transfer from input to output is accomplished with a control word of all 0's (making the B field 000). A register can be cleared to 0 with an exclusive-OR operation. This is because $x \oplus x = 0$.

Table 11.3 Examples for the Microoperations for the CPU

	Symbolic Designation				
Microoperation	SELA	SELB	SELD	OPR	Control Word
$R1 \leftarrow R2 - R3$	R2	R3	R1	SUB	010 011 001 00101
$R4 \leftarrow R4 \vee R5$	R4	R5	R4	OR	100 101 100 01010
$R6 \leftarrow R6 + 1$	R6	—	R6	INCA	110 000 110 00001
$R7 \leftarrow R1$	R1	—	R7	TSFA	001 000 111 00000
Output $\leftarrow R2$	R2	—	None	TSFA	010 000 000 00000
Output $\leftarrow$ Input	Input	—	None	TSFA	000 000 000 00000
$R4 \leftarrow$ shl $R4$	R4	—	R4	SHLA	100 000 100 11000
$R5 \leftarrow 0$	R5	R5	R5	XOR	101 101 101 01100

It is apparent from these examples that many other microoperations can be generated in the CPU. The most efficient way to generate control words with a large number of bits is to store them in a memory unit. A memory unit that stores control words is referred to as a control memory. By reading consecutive control words from memory, it is possible to initiate the desired sequence of microoperations for the CPU. This type of control is referred to as microprogrammed control.

11.3 STACK ORGANIZATION:

A useful feature that is included in the CPU of most computers is a stack or last-in, first-out (UFO) list. 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 (after an initial value is loaded into it). 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 two operations of a stack are the insertion and deletion of items. However, nothing is pushed or popped in a computer stack. These operations are simulated by incrementing or decrementing the stack pointer register.

Register 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 11.4 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.

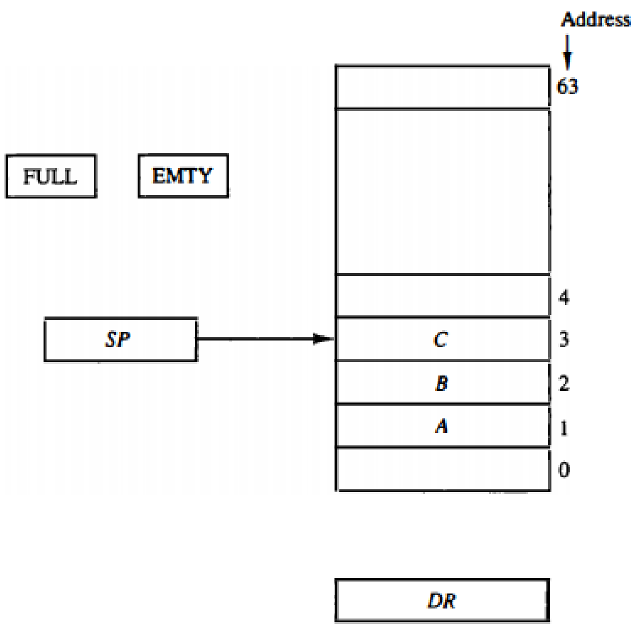


Figure11.4: Computer memory with program, data, and stack segments

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. Note that item C has been read out but not physically removed. This does not matter because when the stack is pushed, a new item is written in its place. 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). When63 is 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. Initially, SP is cleared to 0, EMTY is set to 1, and FULL is cleared to 0, so that SP points to the word at address 0 and the stack is marked empty and not full. If the stack is not full (if FULL = 0), a new item is inserted with a push operation.

The push operation is implemented with the following sequence of microoperations;

SP ← SP + 1 Increment stack pointer

M[SP] ← DR   Write item on top of the stack

If (SP = 0) then (FULL ←1) Check if stack is full

EMTY ← 0   Mark the stack not empty

The stack pointer is incremented so that it points to the address of the next-higher word. A memory write operation inserts the word from DR into the top of the stack. Note that 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 L The last item is stored at address 0.

If SP reaches 0, the stack is full of items, so FULL is set to L. 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. A new item is deleted from the stack if the stack is not empty (if EMTY = 0). The pop operation consists of the following sequence of microoperations:

DR ← M[SP] Read item from the top of stack

SP ← SP - 1 Decrement stack pointer

If (SP = 0) then (EMTY ← 1) Check if stack is empty

FULL ← 0 Mark the stack not full

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. Note that if a pop operation reads the item from location 0 and then SP is decremented, SP 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:

A stack can exist as a stand-alone unit as in Fig. 3 or can be implemented in a random-access memory attached to a CPU. 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 11.5 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. The address register AR points at an array of data.

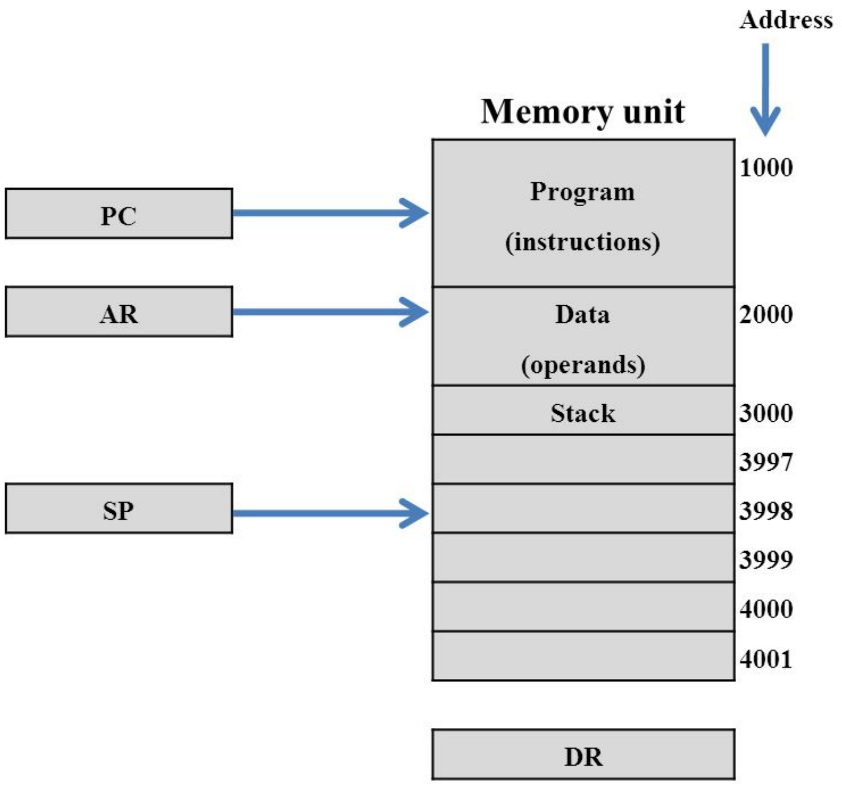


Figure 11.5 Computer Memory with program, data, and stack segments

The stack pointer SP points at the top of the stack. The three registers are connected to a common address bus, and either one can provide an address for memory. PC is used during the fetch phase to read an instruction. AR is used during the execute phase to read an operand. SP is used to push or pop items into or the stack. As shown in Fig. 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. No provisions are available for stack limjt checks.

We assume that the items in the stack communicate with a data register DR. A new item is inserted with the push operation as follows:

SP ← SP - 1

M[SP] ← 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. A new item is deleted with a pop operation as follows:

DR ← M[SP]

SP ← 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.

Most computers do not provide hardware to check for stack overflow (full stack) or underflow (empty stack). The stack limits can be checked by using two processor registers: one to hold the upper limit (3000 in this case), and the other to hold the lower limit (4001 in this case). After a push operation, SP is compared with the upper-limit register and after a pop operation, SP is compared with the lower-limit register. 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 Fig. 11.5 the stack grows by decreasing the memory address. The stack may be constructed to grow by increasing the memory address as in Fig. 11.4. In such a case, SP is incremented for the push operation and decremented for the pop operation. A stack may be constructed so that SP points at the next empty location above the top of the stack. In this case the sequence of microoperations must be interchanged. A stack pointer is loaded with an initial value. This initial value must be the bottom address of an assigned stack in memory. Henceforth, SP is automatically decremented or incremented with every push or pop operation.

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.

11.4 INSTRUCTION FORMAT:

A computer will usually have a variety of instruction code formats. It is the function of the control unit within the CPU to interpret each instruction code and provide the necessary control functions needed to process the instruction. The format of an instruction is usually depicted in a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a control register. The bits of the instruction are divided into groups called fields. The most common fields found in instruction formats are:

1. An operation code field that specifies the operation to be performed.
2. An address field that designates a memory address or a processor register.
3. A mode field that specifies the way the operand or the effective address is determined.

Other special fields are sometimes employed under certain circumstances, as for example a field that gives the number of shifts in a shift-type instruction. The operation code field of an instruction is a group of bits that define various processor operations, such as add, subtract, complement, and shift.

The bits that define the mode field of an instruction code specify a variety of alternatives for choosing the operands from the given address. Operations specified by computer instructions are executed on some data stored in memory or processor registers. Operands residing in memory are specified by their memory address. Operands residing in processor registers are specified with a register address.

A register address is a binary number of k bits that defines one of '2k' registers in the CPU. Thus a CPU with 16 processor registers R0 through R15 will have a register address field of four bits. The binary number 0101, for example, will designate register RS. Computers may have instructions of several different lengths containing varying number of addresses. The number of address fields in the instruction format of a computer depends on the internal organization of its registers.

Most computers fall into one of three types of CPU organizations:

1. Single accumulator organization.

2. General register organization.

3. Stack organization.

In an accumulator-type organization all operations are performed with an implied accumulator register. The instruction format in this type of computer uses one address field.

For example, the instruction that specifies an arithmetic addition is defined by an assembly language instruction as ADD X where X is the address of the operand. The ADD instruction in this case results in the operation AC ← AC + M [X].

AC is the accumulator register and M [X] symbolizes the memory word located at address X.

In a general register type of organization the instruction format in this type of computer needs three register address fields.

Thus the instruction for an arithmetic addition may be written in an assembly language as ADD R1 , R2 , R3 to denote the operation R1 ← R2 + R3.

The number of address fields in the instruction can be reduced from three to two if the destination register is the same as one of the source registers.

Thus the instruction ADD R1, R2 would denote the operation R1 ← R1 + R2. Only register addresses for R1 and R2 need be specified in this instruction.

Computers with multiple processor registers use the move instruction with a mnemonic MOV to symbolize a transfer instruction. Thus the instruction MOV R1, R2 denotes the transfer R1 ← R2 (or R2 ← R1, depending on the particular computer).

Thus transfer-type instructions need two address fields to specify the source and the destination.

General register-type computers employ two or three address fields in the instruction.

Three-Address Instructions:

Computers with three-address instruction formats can use each address field to specify either a processor register or a memory operand. The program in assembly language that evaluates X = (A + B) * (C + D) is shown below, together with comments that explain the register transfer operation of each instruction.

ADD R1, A, B R1 ← M[A] + M[B]

ADD R2, C, D R2 ← M[C] + M[D]

MOL X, R1, R2 M[X] ← R1 * R2

It is assumed that the computer has two processor registers, R1 and R2. The symbol M[A] denotes the operand at memory address symbolized by A.

The advantage of the three-address format is that it results in short programs when evaluating arithmetic expressions. The disadvantage is that the binary-coded instructions require too many bits to specify three addresses.

Two-Address Instructions:

Two-address instructions are the most common in commercial computers. Here again each address field can specify either a processor register or a memory word. The program to evaluate X = (A + B) * (C + D) is as follows:

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]

MOL R1, R2 R1 ← R1 * R2

MOV X, R1   M[X] ← R1

The MOV instruction moves or transfers the operands to and from memory and processor registers. The first symbol listed in an instruction is assumed to be both a source and the destination where the result of the operation is transferred.

One-Address Instructions:

One-address instructions use an implied accumulator (AC) register for all data manipulation. For multiplication and division there is a need for a second register. However, here we will neglect the second register and assume that the AC contains the result of all operations.

The program to evaluate X = (A + B) * (C + D) is

LOAD A AC ← M[AJ

ADD B   AC ← AC + M[B]

STORE T M[T] ← AC

LOAD c AC ← M[C]

ADD D   AC ← AC + M[D]

MOL T   AC ← AC*M[T]

STORE X M[X] ← AC

All operations are done between the AC register and a memory operand. T is the address of a temporary memory location required for storing the intermediate result.

Zero-Address Instructions:

A stack-organized computer does not use an address field for the instructions ADD and MUL. The PUSH and POP instructions, however, need an address field to specify the operand that communicates with the stack. The following program shows how X = (A + B) * (C + D) will be written for a stack organized computer. (TOS stands for top of stack.)

PUSH A TOS ← A

PUSH B TOS ← B

ADD TOS ← (A + B)

PUSH C TOS ← C

PUSH D TOS ← D

ADD TO S ← (C + D)

MOL TOS ← (C + D)*(A + B)

POP X M[X] ← TOS

To evaluate arithmetic expressions in a stack computer, it is necessary to convert the expression into reverse Polish notation. The name "zero-address" is given to this type of computer because of the absence of an address field in the computational instructions.

11.5 ADDRESSING MODES:

The operation field of an instruction specifies the operation to be performed. This operation must be executed on some data stored in computer registers or memory words. The way the operands are chosen during program execution is dependent on the addressing mode of the instruction. The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced.

Computers use addressing mode techniques for the purpose of accommodating one or both of the following provisions:

- To give programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data, and program relocation.
- To reduce the number of bits in the addressing field of the instruction.

The availability of the addressing modes gives the experienced assembly language programmer flexibility for writing programs that are more efficient with respect to the number of instructions and execution time.

To understand the various addressing modes to be presented in this section, it is imperative that we understand the basic operation cycle of the computer.

The control unit of a computer is designed to go through an instruction cycle that is divided into three major phases:

1. Fetch the instruction from memory.
2. Decode the instruction.
3. Execute the instruction.

There is one register in the computer called the program counter or PC that keeps track of the instructions in the program stored in memory. PC holds the address of the instruction to be executed next and is incremented each time an instruction is fetched from memory.

The decoding done in step 2 determines the operation to be performed, the addressing mode of the instruction, and the location of the operands. The computer then executes the instruction and returns to step 1 to fetch the next instruction in sequence.

In some computers the addressing mode of the instruction is specified with a distinct binary code, just like the operation code is specified. Other computers use a single binary code that designates both the operation and the mode of the instruction.

Instructions may be defined with a variety of addressing modes, and sometimes, two or more addressing modes are combined in one instruction. An example of an instruction format with a distinct addressing mode field is shown in Fig. 6. The operation code specifies the operation to be permode field formed. The mode field is used to locate the operands needed for the operation.

There may or may not be an address field in the instruction. If there is an address field, it may designate a memory address or a processor register. Moreover, as discussed in the preceding section, the instruction may have more than one address field, and each address field may be associated with its own particular addressing mode.

Although most addressing modes modify the address field of the instruction, there are two modes that need no address field at all. These are the implied and immediate modes.

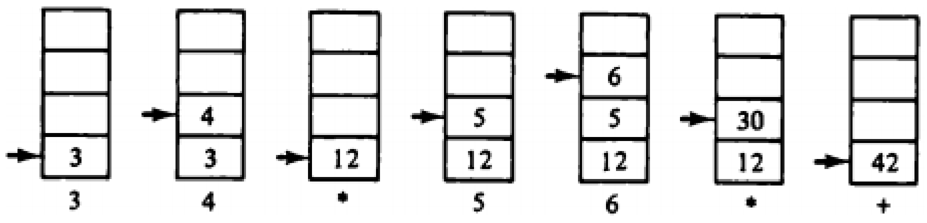


*Figure 11.6 Stack operations to evaluate 3 * 4 + 5 * 6*

Implied Mode:

In this mode the operands are specified implicitly in the definition of the instruction. For example, the instruction “complement accumulator” is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction. In fact, all register reference instructions that use an accumulator are implied-mode instructions.

Zero-address instructions in a stack-organized computer are implied-mode instructions since the operands are implied to be on top of the stack.

Immediate Mode:

In this mode the operand is specified in the instruction itself. In other words, an immediate-mode instruction has an operand field rather than an address field.

The operand field contains the actual operand to be used in conjunction with the operation specified in the instruction. Immediate-mode instructions are useful for initializing registers to a constant value.

It was mentioned previously that the address field of an instruction may specify either a memory word or a processor register. When the address field specifies a processor register, the instruction is said to be in the register mode.

Register Mode:

In this mode the operands are in registers that reside within the CPU. The particular register is selected from a register field in the instruction.

A k-bit field can specify any one of 2^k registers.

Register Indirect Mode:

In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory. In other words, the selected register contains the address of the operand rather than the operand itself.

Before using a register indirect mode instruction, the programmer must ensure that the memory address of the operand is placed in the processor register with a previous instruction. A reference to the register is then equivalent to specifying a memory address.

The advantage of a register indirect mode instruction is that the address field of the instruction uses fewer bits to select a register than would have been required to specify a memory address directly.

Autoincrement or Autodecrement Mode:

This is similar to the register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory. When the address stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table. This can be achieved by using the increment or decrement instruction.

However, because it is such a common requirement, some computers incorporate a special mode that automatically increments or decrements the content of the register after data access. The address field of an instruction is used by the control unit in the CPU to obtain the operand from memory.

Sometimes the value given in the address field is the address of the operand, but sometimes it is just an address from which the address of the operand is calculated. To differentiate among the various addressing modes it is necessary to distinguish between the address part of the instruction and the effective address used by the control when executing the instruction.

The effective address is defined to be the memory address obtained from the computation dictated by the given addressing mode. The effective address is the address of the operand in a computational- type instruction. It is the address where control branches in response to a branch-type instruction.

Direct Address Mode:

In this mode the effective address is equal to the address part of the instruction.The operand resides in memory and its address is given directly by the address field of the instruction.In a branch-type instruction the address field specifies the actual branch address.

Indirect Address Mode:

In this mode the address field of the instruction gives the address where the effective address is stored in memory. Control fetches the instruction from memory and uses its address part to access memory again to read the effective address. A few addressing modes require that the address field of the instruction be added to the content of a specific register in the CPU.

The effective address in these modes is obtained from the following computation:

Effective address = address part of instruction + content of CPU register

The CPU register used in the computation may be the program counter, an index register, or a base register. In either case we have a different addressing mode which is used for a different application.

Relative Address Mode:

In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address.

The address part of the instruction is usually a signed number (in 2's complement representation) which can be either positive or negative. When this number is added to the content of the program counter, the result produces an effective address whose position in memory is relative to the address of the next instruction.

To clarify with an example, assume that the program counter contains the number 825 and the address part of the instruction contains the number 24.

The instruction at location 825 is read from memory during the fetch phase and the program counter is then incremented by one to 826. The effective address computation for the relative address mode is 826 + 24 = 850.

This is 24 memory locations forward from the address of the next instruction. Relative addressing is often used with branch-type instructions when the branch address is in the area surrounding the instruction word itself.

It results in a shorter address field in the instruction format since the relative address can be specified with a smaller number of bits compared to the number of bits required to designate the entire memory address.

Indexed Addressing Mode:

In this mode the content of an index register is added to the address part of the instruction to obtain the effective address. The index register is a special CPU register that contains an index value. The address field of the instruction defines the beginning address of a data array in memory. Each operand in the array is stored in memory relative to the beginning address. The distance between the beginning address and the address of the operand is the index value stored in the index register.

Any operand in the array can be accessed with the same instruction provided that the index register contains the correct index value. The index register can be incremented to facilitate access to consecutive operands. Note that if an index type instruction does not include an address field in its format, the instruction converts to the register indirect mode of operation.

Some computers dedicate one CPU register to function solely as an index register. This register is involved implicitly when the index-mode instruction is used. In computers with many processor registers, any one of the CPU registers can contain the index number. In such a case the register must be specified explicitly in a register field within the instruction format.

Base Register Addressing Mode:

In this mode the content of a base register is added to the address part of the instruction to obtain the effective address. This is similar to the indexed addressing mode except that the register is now called a base register instead of an index register.

The difference between the two modes is in the way they are used rather than in the way that they are computed. An index register is assumed to hold an index number that is relative to the address part of the instruction. A base register is assumed to hold a base address and the address field of the instruction gives a displacement relative to this base address. The base register addressing mode is used in computers to facilitate the relocation of programs in memory.

When programs and data are moved from one segment of memory to another, as required in multiprogramming systems, the address values of instructions must reflect this change of position. With a base register, the displacement values of instructions do not have to change. Only the value of the base register requires updating to reflect the beginning of a new memory segment.

Numerical Example:

To show the differences between the various modes, we will show the effect of the addressing modes on the instruction defined in Fig. 11.

The two-word instruction at address 200 and 201 is a "load to AC" instruction with an address field equal to 500.The first word of the instruction specifies the operation code and mode, and the second word specifies the address part. PC has the value 200 for fetching this instruction. The content of processor register R1 is 400, and the content of an index register XR is 100.AC receives the operand after the instruction is executed. The figure lists a few pertinent addresses and shows the memory content at each of these addresses.

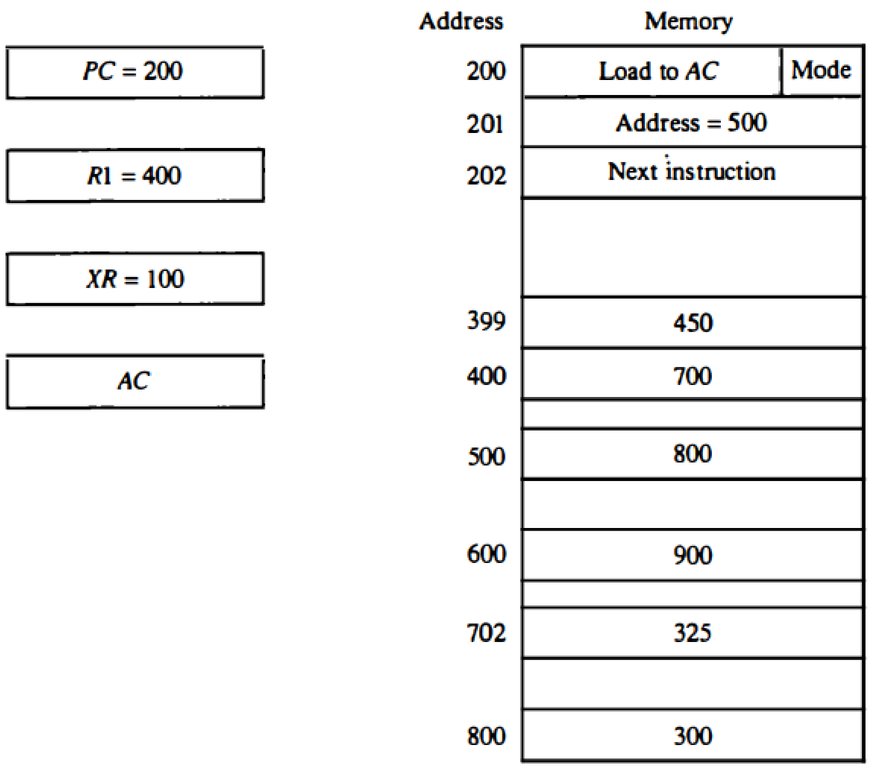


Figure 11.7 Numerical Example for addressing modes

The mode field of the instruction can specify any one of a number of modes. For each possible mode we calculate the effective address and the operand that must be loaded into AC.In the direct address mode the effective address is the address part of the instruction 500 and the operand to be loaded into AC is 800. In the immediate mode the second word of the instruction is taken as the operand rather than an address, so 500 is loaded into AC. (The effective address in this case is 201.)

In the indirect mode the effective address is stored in memory at address 500. Therefore, the effective address is 800 and the operand is 300. In the relative mode the effective address is 500 + 202 = 702 and the operand is 325.

(Note that the value in PC after the fetch phase and during the execute phase is 202.) In the index mode the effective address is XR + 500 = 100 + 500 = 600 and the operand is 900. In the register mode the operand is in R1 and 400 is loaded into AC.

(There is no effective address in this case.) In the register indirect mode the effective address is 400, equal to the content of R1 and the operand loaded into AC is 700.

The auto increment mode is the same as the register indirect mode except that R1 is incremented to 401 after the execution of the instruction. The auto decrement mode decrements R1 to 399 prior to the execution of the instruction.

The operand loaded into AC is now 450. Table 11.4 lists the values of the effective address and the operand loaded into AC for the nine addressing modes.

Table 11.4 Tabular list of numerical example

Addressing Mode	Effective Address	Content of *AC*
Direct address	500	800
Immediate operand	201	500
Indirect address	800	300
Relative address	702	325
Indexed address	600	900
Register	—	400
Register indirect	400	700
Autoincrement	400	700
Autodecrement	399	450

11.6 DATA TRANSFER AND MANIPULATION:

Most computer instructions can be classified into three categories:

1. Data transfer instructions

2. Data manipulation instructions

3. Program control instructions

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.

Table 7-5 gives a list of eight data transfer instructions used in many computers.

The load instruction has been used mostly to designate a transfer from memory to a processor register, usually an accumulator. The store instruction designates a transfer from a processor register into memory.

Table 11.5: Typical data transfer instructions

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

The move instruction has been used in computers with multiple CPU registers to designate a transfer from one register to another and also 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. The push and pop instructions transfer data between processor registers and a memory stack.

Different computers use different mnemonics symbols for differentiate the addressing modes. As an example, consider the load to accumulator instruction when used with eight different addressing modes.

Table 11.6 shows the recommended assembly language convention and actual transfer accomplished in each case

Table 11.6 Eight addressing modes for the Load instruction

Mode	Assembly Convention	Register Transfer
Direct address	LD ADR	$AC \leftarrow M[ADR]$
Indirect address	LD @ADR	$AC \leftarrow M[M[ADR]]$
Relative address	LD $ADR	$AC \leftarrow M[PC + ADR]$
Immediate operand	LD #NBR	$AC \leftarrow NBR$
Index addressing	LD ADR(X)	$AC \leftarrow M[ADR + XR]$
Register	LD R1	$AC \leftarrow R1$
Register indirect	LD (R1)	$AC \leftarrow M[R1]$
Autoincrement	LD (R1)+	$AC \leftarrow M[R1],\ R1 \leftarrow R1 + 1$

ADR stands for an address.

NBA a number or operand.

X is an index register.

R1 is a processor register.

AC is the accumulator register.

The @ character symbolizes an indirect addressing.

The $ character before an address makes the address relative to the program counter PC.

The # character precedes the operand in an immediate-mode instruction.

An indexed mode instruction is recognized by a register that placed in parentheses after the symbolic address. The register mode is symbolized by giving the name of a processor register. In the register indirect mode, the name of the register that holds the memory address is enclosed in parentheses. The auto-increment mode is distinguished from the register indirect mode by placing a plus after the parenthesized register. The auto-decrement mode would use a minus instead.

Data Manipulation Instructions:

Data manipulation instructions perform operations on data and provide the computational capabilities for the computer.

The data manipulation instructions in a typical computer are usually divided into three basic types:

1. Arithmetic instructions
2. Logical and bit manipulation instructions
3. Shift instructions

1. Arithmetic instructions:

The four basic arithmetic operations are addition, subtraction, multiplication and division. Most computers provide instructions for all four operations. Some small computers have only addition and possibly subtraction instructions. The multiplication and division must then be generated by mean software subroutines.

A list of typical arithmetic instructions is given in Table 11.11.

Table 11.7: Typical 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

The increment instruction adds 1 to the value stored in a register or memory word. A number with all 1's, when incremented, produces a number with all 0's. The decrement instruction subtracts 1 from a value stored in a register or memory word. A number with all 0's, when decremented, produces number with all 1's.Theadd, subtract, multiply, and divide instructions may be use different types of data.

The data type assumed to be in processor register during the execution of these arithmetic operations is defined by an operation code. An arithmetic instruction may specify fixed-point or floating-point data, binary or decimal data, single-precision or double-precision data.

The mnemonics for three add instructions that specify different data types are shown below.

ADDI	*Add two binary integer numbers*
ADDF	*Add two floating-point numbers*
ADDD	*Add two decimal numbers in BCD*

A special carry flip-flop is used to store the carry from an operation. The instruction "add carry" performs the addition on two operands plus the value of the carry the previous computation. Similarly, the "subtract with borrow" instruction subtracts two words and borrow which may have resulted from a previous subtract operation. The negate instruction forms the 2's complement number, effectively reversing the sign of an integer when represented it signed-2's complement form.

2. Logical and bit manipulation instructions:

Logical instructions perform binary operations on strings of bits store, registers. They are useful for manipulating individual bits or a group of that represent binary-coded information. The logical instructions consider each bit of the operand separately and treat it as a Boolean variable.

By proper application of the logical instructions it is possible to change bit values, to clear a group of bits, or to insert new bit values into operands stored in register memory words. Some typical logical and bit manipulation instructions are listed in Table 11.8.

Table 11.8 Typical 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
Disable interrupt	DI

The clear instruction causes the specified operand to be replaced by 0’s.The complement instruction produces the 1's complement by inverting all bits of the operand. The AND, OR, and XOR instructions produce the corresponding logical operations on individual bits of the operands.

The logical instructions can also be used to performing bit manipulation operations. There are three bit manipulation operations possible: a selected bit can cleared to 0, or can be set to 1, or can be complemented.

- o The AND instruction is used to clear a bit or a selected group of bits of an operand.
- o The OR instruction is used to set a bit or a selected group of bits of an operand.
- o Similarly, the XOR instruction is used to selectively complement bits of an operand.

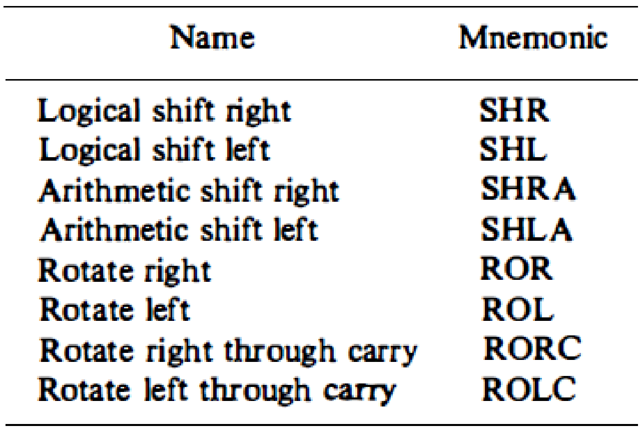
Other bit manipulations instructions are included in above table perform the operations on individual bits such as a carry can be cleared, set, or complemented. Another example is a flip-flop that controls the interrupt facility and is either enabled or disabled by means of bit manipulation instructions.

3. Shift Instructions:

Shifts are operations in which the bits of a word are moved to the left or right. The bit shifted in at the end of the word determines the type of shift used. Shift instructions may specify logical shifts, arithmetic shifts, or rotate-type operations. In either case the shift may be to the right or to the left.

Table 11.9 lists four types of shift instructions.

Table 11.9 Typical 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 through carry	RORC
Rotate left through carry	ROLC

The logical shift inset to the end bit position. The end position is the leftmost bit position for shift rights the rightmost bit position for the shift left. Arithmetic shifts usually conform to the rules for signed-2's complement numbers. The arithmetic shift-right instruction must preserve the sign bit in the leftmost position. The sign bit is shifted to the right together with the rest of the number, but the sign bit itself remains unchanged.

This is a shift-right operation with the end bit remaining the same.The arithmetic shift-left instruction inserts 0 to the end position and is identical to the logical shift-instruction. The rotate instructions produce a circular shift. Bits shifted out at one of the word are not lost as in a logical shift but are circulated back into the other end.

The rotate through carry instruction treats a carry bit as an extension of the register whose word is being rotated. Thus a rotate-left through carry instruction transfers the carry bit into the rightmost bit position of the register, transfers the leftmost bit position into the carry, and at the same time, shift the entire register to the left**.**

11.7 PROGRAM CONTROL:

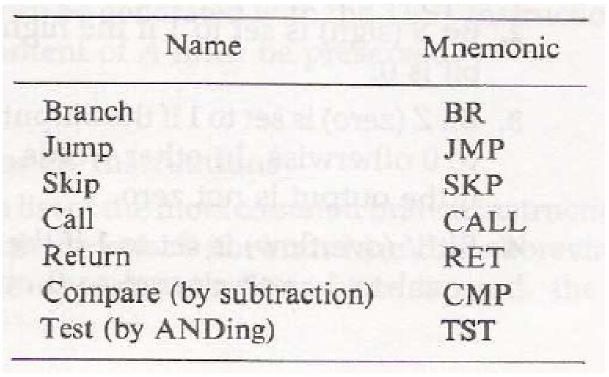
Instructions are always stored in successive memory locations. When processed in the CPU, the instructions are fetched from consecutive memory locations and executed. Each time an instruction is fetched from memory, the program counter is incremented so that it contains the address of the next instruction in sequence.

After the execution of a data transfer or data manipulation instruction, control returns to the fetch cycle with the program counter containing the address of the instruction next in sequence. On the other hand, a program control type of instruction, when executed, may change the address value in the program counter and cause the flow of control to be altered.

In other words, program control instructions specify conditions for altering the content of the program counter, while data transfer and manipulation instructions specify conditions for data-processing operations.

The change in value of the program counter as a result of the execution of a program control instruction causes a break in the sequence of instruction execution. This is an important feature in digital computers, as it provides control over the flow of program execution and a capability for branching to different program segments.

Some typical program control instructions are listed in Table 11.10.



Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TST

The branch is usually a one-address instruction. It is written in assembly language as BR ADR, where ADR is a symbolic name for an address.

When executed, the branch instruction causes a transfer of the value of ADR into the program counter. Since the program counter contains the address of the instruction to be executed, the next instruction will come from location ADR. Branch and jump instructions may be conditional or unconditional. An unconditional branch instruction causes a branch to the specified address without any conditions.

Status Bit Conditions:

The status register comprises the status bits. The bits of the status register are modified according to the operations performed in the ALU. Status bits are also called as condition-code or bits or flag bits. The figure displays a 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.

1. If the end carry C_8 is 1, then *carry (C)* is set to 1. If C_8 is 0, then C is cleared to 0.
2. If the highest order bit F_7 is 1, then *Sign (S)* is set to 1. If F_7 is 0, then S is set to 0.
3. If the output of ALU is 0, then *zero (Z)* is set to 1, otherwise, Z is set to 0.
4. If the XOR of the last two carries is equal to 1, then *overflow (V)* is set to 1, otherwise, V is cleared to 0.

The result of the 8-bit ALU operation is either 127 or -127.

Z is a status bit used to indicate the result obtained after comparing A and B. Here, the XOR operation is used to compare two numbers (Z = 0 if A = B).

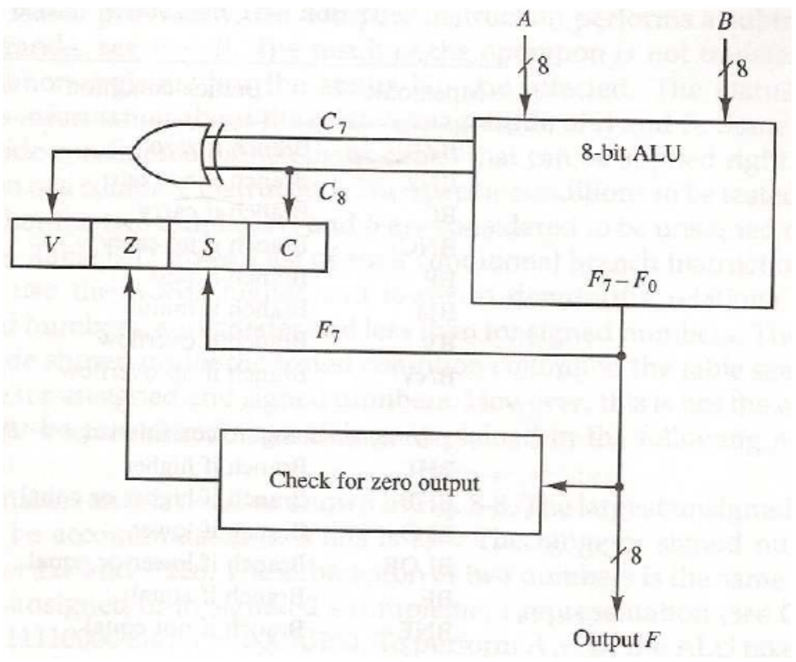


Figure 11.8 Status register bits

Conditional Branch Instructions:

A conditional branch instruction is a branch instruction that may or may not cause a transfer of control depending on the value of stored bits in the PSR (processor status register). Each conditional branch instruction tests a different combination of Status bits for a condition. If the condition is true, control is transferred to the effective address (PC←Add). If the condition is false, the program continues with the next instruction (PC←PC+1).

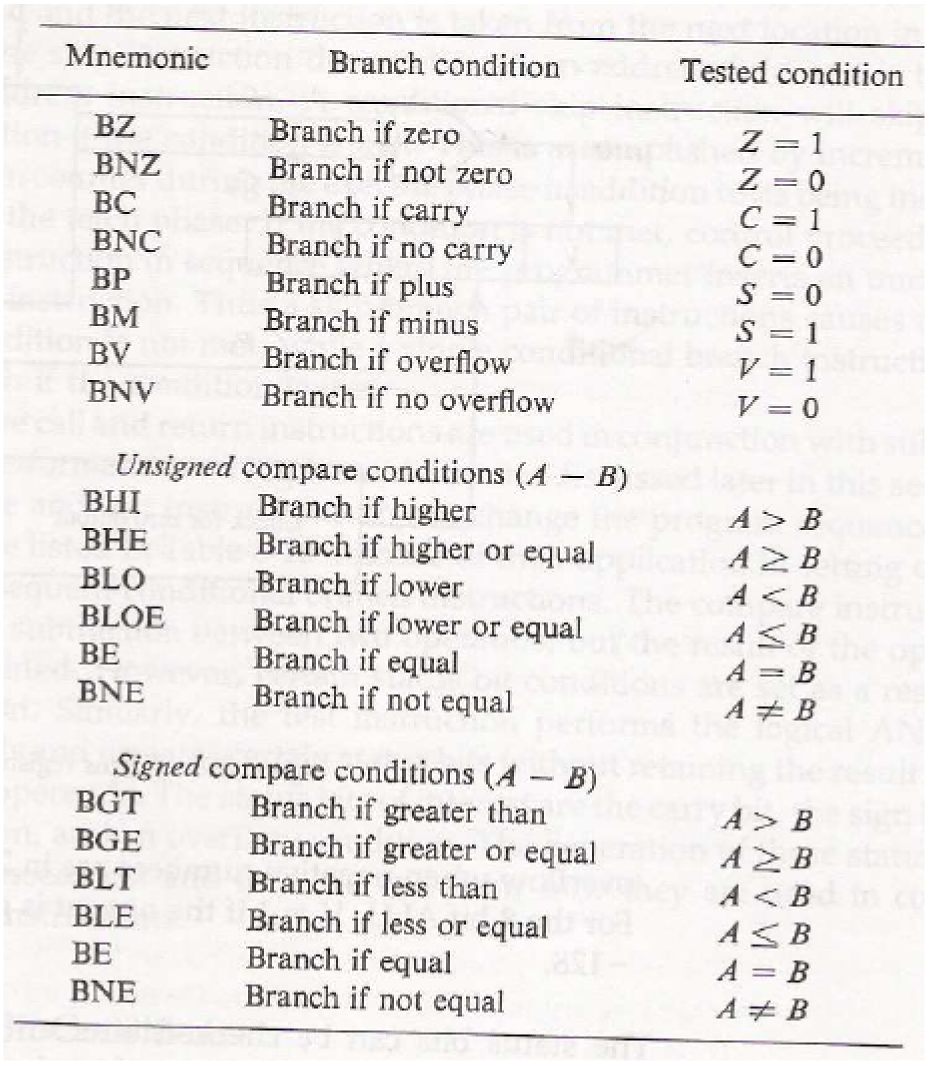
Below is a list of Conditional Branch instructions, letter 'N' stands for NOT. Some branch instructions are a combination of compare and conditional branch instructions. They are run after the compare instruction has performed the comparison and status bits are updated. Different status bits are checked for signed and unsigned numbers.

A≥B is complement of A<B.

A≤B is complement of A>B

That means if we know the condition of status bits for one, the condition for the other complementary relation is obtained by complement

Table 11.8: 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 overflow	$V = 1$
BNV	Branch if no overflow	$V = 0$
	Unsigned compare conditions $(A - B)$	
BHI	Branch if higher	$A > B$
BHE	Branch if higher or equal	$A \geq B$
BLO	Branch if lower	$A < B$
BLOE	Branch if lower or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$
	Signed compare conditions $(A - B)$	
BGT	Branch if greater than	$A > B$
BGE	Branch if greater or equal	$A \geq B$
BLT	Branch if less than	$A < B$
BLE	Branch if less or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$

11.8 SUMMARY:

This chapter covers essential aspects of computer architecture crucial to understanding its operation. It begins with an introduction to the CPU as the core component responsible for executing instructions and managing data within a computer system. General Register Organization details the role and structure of registers for storing operands and intermediate results during computation. Stack Organization explores the stack-based memory structure used for managing subroutine calls and local variables. Instruction Format delves into the structure of machine instructions, including op codes and addressing modes that specify data access methods. Addressing Modes further elaborate on how operands are accessed in instructions, influencing program flexibility. Data Transfer and Manipulation elucidate how the CPU processes data through operations like arithmetic, logic, and movement between registers and memory. Program Control examines mechanisms governing program flow, including branching, subroutine handling, and interrupt management, crucial for efficient execution of instructions. Together, these topics provide a comprehensive foundation on the CPU's internal mechanisms and its pivotal role in computing systems.

11.9 TECHNICAL TERMS:

Central Processing Unit, Register, Stack, Addressing Mode.

11.10 SELF ASSESSMENT QUESTIONS:

Essay questions:

1) Explain the General Register Organization in a CPU
2) Describe Stack Organization in detail.
3) Discuss different Instruction Formats in CPU architecture.
4) What are Addressing Modes?
5) Explain Program Control in CPU.

Short Questions:

1) What is the role of a General Register in a CPU?
2) Differentiate between a stack and a register.
3) Define Instruction Format. What are its components?
4) List any four types of Addressing Modes with examples.
5) What are the different types of Program Control Instructions?

11.11 FURTHER READINGS:

1) M.Morris Mano, “Computer System Architecture”, 3rd Edition, Pearson Education (2008).
2) V. Rajaraman, T. Radha Krishnan, “Computer Organization and Architecture”, PHI.
3) Behrooz Parhami, “Computer Architecture”, Oxford (2007).
4) ISRD group, “Computer Organization”, ace series, TMH (2007).
5) William Stallings, “Computer Organization and Architecture-Designing for Performance”, Pearson Education (2005).

Dr. Kampa Lavanya

LESSON-12

COMPUTER ARITHMETIC

OBJECTIVES:

After going through this lesson, you will be able to

- Develop the ability to perform and implement basic arithmetic operations in both hardware and software.
- Learn the rules for binary addition and subtraction.
- Understand the process of binary multiplication.
- Understand the complexity of different division algorithms.
- Perform arithmetic operations with floating-point numbers.

STRUCTURE OF THE LESSION:

12.1 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.

12.2 ADDITION AND SUBTRACTION:

12.2.1 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. 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).

Table 12.1 Addition and Subtraction of Signed Magnitude Numbers

Operation	Add Magnitudes	Subtract Magnitudes		
		When $A > B$	When $A < B$	When $A = B$
$(+A) + (+B)$	$+(A + B)$			
$(+A) + (-B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(-A) + (+B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$
$(-A) + (-B)$	$-(A + B)$			
$(+A) - (+B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(+A) - (-B)$	$+(A + B)$			
$(-A) - (+B)$	$-(A + B)$			
$(-A) - (-B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$

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).

Hardware Implementation:

When the signs of A and B are same, add the two magnitudes and attach the sign of result is that of A. When the signs of A and B are not same, compare the magnitudes and subtract the smaller number from the larger. Choose the sign of the result to be the same as A, if A > B or the complement of the sign of A if A < B. If the two magnitudes are equal, subtract B from A and make the sign of the result will be positive.

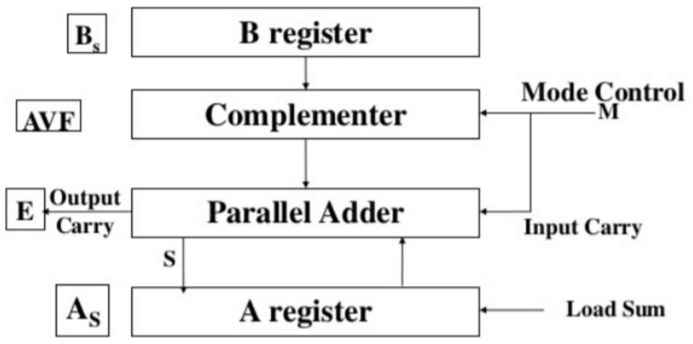


Figure 12.1: Hardware Architecture for Addition and Subtraction of Signed-Magnitude Numbers

The flowchart for the hardware algorithm is presented in Fig. 12.2.

The two signs A, and B are compared by an exclusive-OR gate. If the output of the gate is 0, the signs are identical; if it is 1, the signs are different. For an 0dd operation, identical signs dictate that the magnitudes be added. For a subtraction operation, different signs dictate that the magnitudes be added. The magnitudes are added with a microoperation EA ←A + B. where EA is a register that combines E: and A. The carry in E: after the addition constitutes an overflow if it is equal to 1. The value of E: is transferred into the add-over now flip-flop AVF.

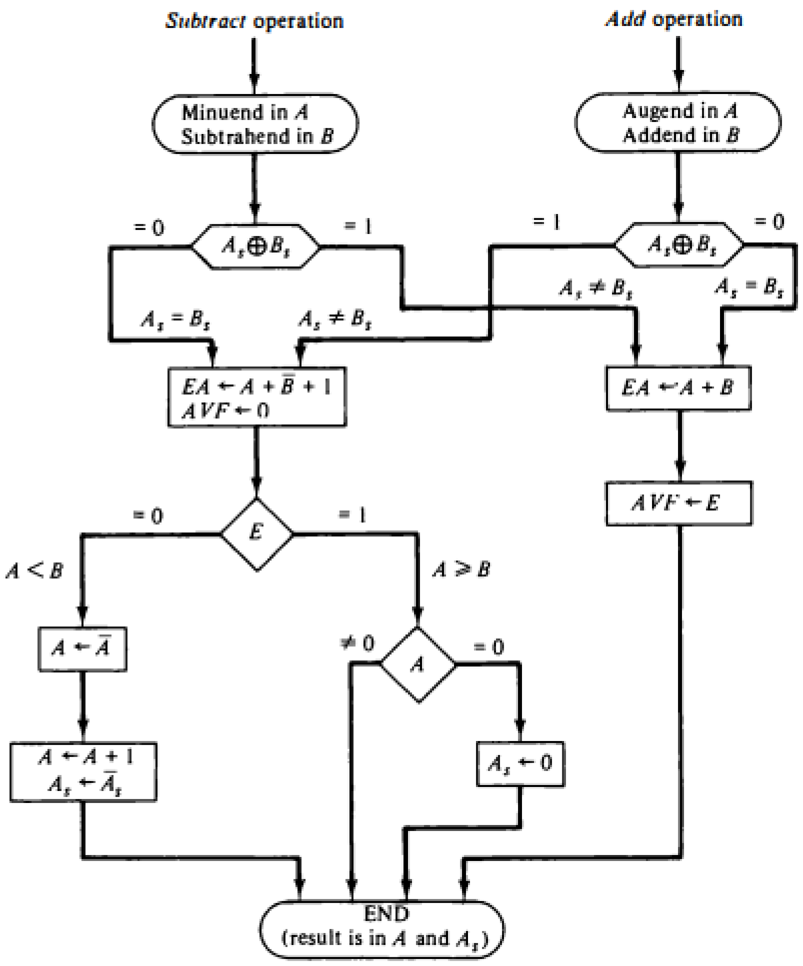


Figure 12.2 Flowchan for add and subtract operation

The two magnitudes are subtracted if the signs are different for an odd operation or identical for a subtract operation. The magnitudes are subtracted by adding A to the 2's complement of B. No overflow can occur if the numbers are subtracted so AVF is cleared to 0. A 1 in E indicates that A ≥ B a.nd the number in A is the correct result. If this nu.mberi.s zero, the sign A, must be made positive to avoid a negative zero. A 0 in E: indicates that A < B. For this case it is necessary to take the 2's complement of the value in A. This operation can be done with one microoperation$A \leftarrow \bar{A}$ + 1. However, we assume that the A register has circuits for mlcrooperations complm.mt and inlTtment, so the 2's complement is obtained from these two mlcrooperations. In other paths of the flowchart, the sign of the result is the same as the sign of A, so no change in A, is required. However, when A < B, the sign of the result is the complement of the original sign of A. It is then necessary to complement A, to obtain the correct sign. The final result is found in register A and its sign in As. The value in AVF provides an overflow indication. The final value of E is immaterial.

Addition and Subtraction with Signed-2's Complement Data

The signed-2's complement representation of numbers together with arithmetic algorithms for addition and subtraction are summarized here for easy reference. The leftmost bit of a binary number represents the sign bit: 0 for positive and 1 for negative. If the sign bit is 1, the entire number is represented in 2's complement form. Thus +33 is represented as 00100001 and -33 as 11011111. Note that 11011111 is the 2's complement of 00100001, and vice versa. The addition of two numbers in signed-2's complement form consists of adding the numbers with the sign bits treated the same as the other bits of the number. A carry-out of the sign-bit position is discarded. The subtraction consists of first taking the 2's complement of the subtrahend and then adding it to the minuend. When two numbers of n digits each are added and the sum occupies n + 1 digits, we say that an overflow occurred. An overflow can be detected by inspecting the last two carries out of the addition. When the two carries are applied to an exclusive-OR gate, the overflow is detected when the output of the gate is equal to 1. The register configuration for the hardware implementation is shown in Figure. The sign bits are not separated from the rest of the registers. We name the A register AC and the B register BR. The leftmost bit in AC and BR represent the sign bits of the numbers. The two sign bits are added or subtracted together with the other bits in the complementer and parallel adder. The overflow flip-flop V is set to 1 if there is an overflow. The output carry in this case is discarded.

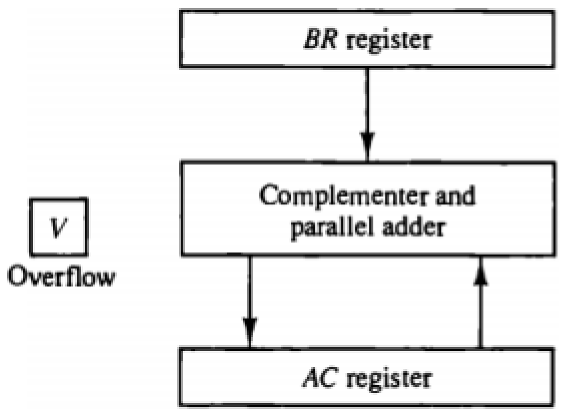


Figure 12.3 Hardware for signed-2's complement addition and subtraction

The algorithm for adding and subtracting two binary numbers in signed-2's complement representation is shown in the flowchart of Figure. 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 in the AC register.

Comparing this algorithm with its signed-magnitude counterpart, we note that it is much simpler to add and subtract numbers if negative numbers are maintained in signed-2's complement representation. For this reason most computers adopt this representation over the more familiar signed-magnitude.

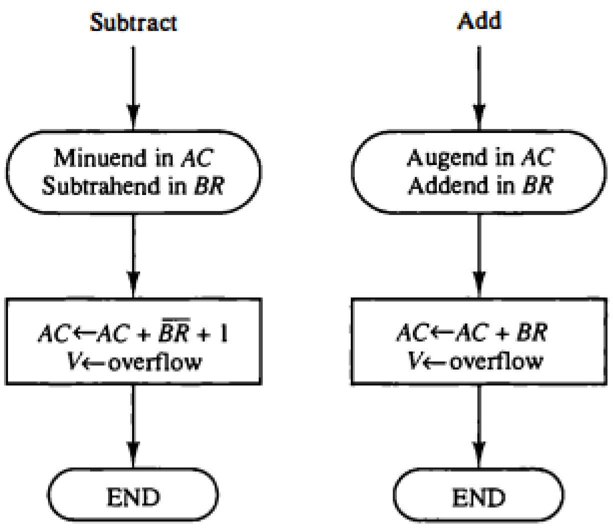


Figure 12.4 Algorithm for adding and subtracting numbers in signed - 2's complement representation

12.3 MULTIPLICATION ALGORITHM:

Multiplication of two fixed-point binary numbers in signed-magnitude representation is done with paper and pencil by a process of successive shift and add operations. This process is best illustrated with a numerical example.

The process consists of looking at successive bits of the multiplier, least significant bit first. If the muliplier bit is a 1, the multiplicand is copied down; otherwise, zeros are copied down. The numbers copied down in successive lines are shifted one position to the left from the previous number. Finally, the numbers are added and their sum forms the product. The sign of the product is determined from the signs of the multiplicand and multiplier. If they are alike, the sign of the product is positive. If they are unlike, the sign of the product is negative.

Hardware Implementation for Signed-Magnitude Data:

When multiplication is implemented in a digital computer, it is convenient to change the process slightly. First, instead of providing registers to store and add simultaneously as many binary numbers as there are bits in the multiplier, it is convenient to provide an adder for the summation of only two binary numbers and successively accumulate the partial products in a register. Second, instead of shifting the multiplicand to the left, the partial product is shifted to the right, which results in leaving the partial product and the multiplicand in the required relative positions. Third, when the corresponding bit of the multiplier is 0, there is no need to add all zeros to the partial product since it will not alter its value.

The hardware for multiplication consists of the equipment shown in Figure.

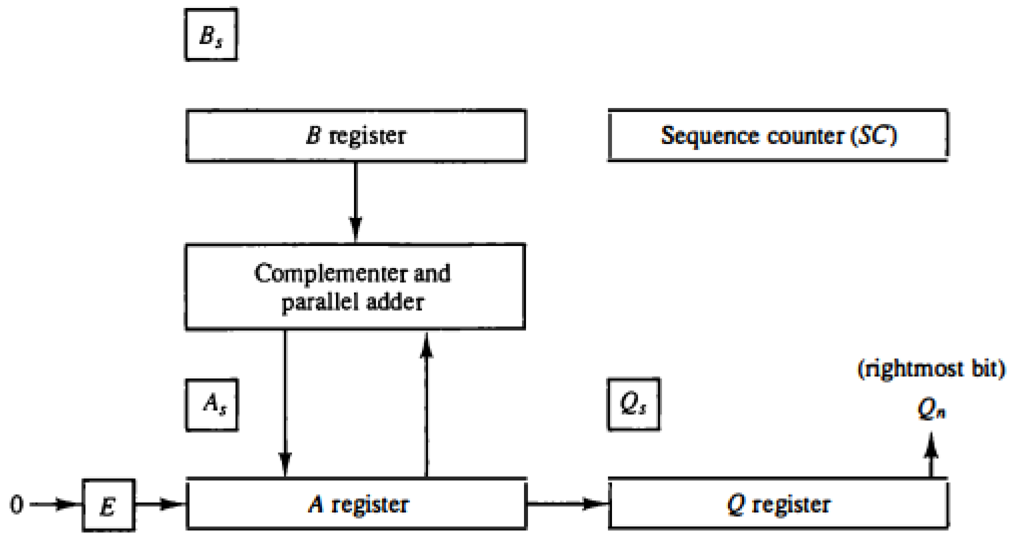


Figure 12.5 Hardware for multiply operation

The multiplier is stored in the Q register and its sign in Qs. The sequence counter SC is initially set to a number equal to the number of bits in the multiplier. The counter is decremented by 1 after forming each partial product. When the content of the counter reaches zero, the product is formed and the process stops.

Initially, the multiplicand is in register B and the multiplier in Q. The sum of A and B forms a partial product which is transferred to the EA register. Both partial product and multiplier are shifted to the right. This shift will be denoted by the statement shr EAQ to designate the right shift depicted in Figure. The least significant bit of A is shifted into the most significant position of Q, the bit from E is shifted into the most significant position of A, and 0 is shifted into E. After the shift, one bit of the partial product is shifted into Q, pushing the multiplier bits one position to the right. In this manner, the rightmost flip-flop in register Q, designated by Q„, will hold the bit of the multiplier, which must be inspected next.

Hardware Algorithm

Figure shows a flowchart of the hardware multiply algorithm.

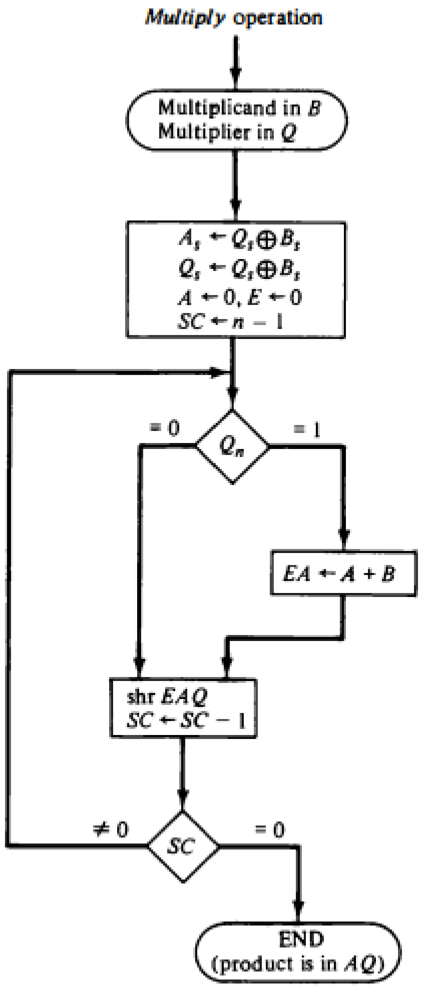


Figure 12.6 Flowchart for multiply operation

Initially, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in Bs and Qs, respectively. The signs are compared, and both A and Q are set to correspond to the 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 a number equal to the number of bits of the multiplier. We are assuming here that operands are transferred to registers from a memory unit that has words of n bits. 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. After the initialization, the low-order bit of the multiplier in Qn is tested. If it is a 1, the multiplicand in B is added to the present partial product in A. If it is a 0, nothing is done. 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. The process stops when SC = 0. Note that the partial product formed in A is shifted into Q one bit at a time and eventually replaces the multiplier. The final product is available in both A and Q, with A holding the most significant bits and Q holding the least significant bits. The previous numerical example is repeated in Table is shown to clarify the hardware multiplication process. The procedure follows the steps outlined in the flowchart.

Table 12.2 Numerical Example for Binary Multiplier

Multiplicand 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		10111		
Fifth partial product	0	11011		
Shift right EAQ	0	01101	10101	000
Final product in AQ = 0110110101				

Booth Multiplication 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 Ts in the multiplier from bit weight 2* to weight 2 m can be treated as $2^{k+1} - 2^{m}$. For example, thebinary number 001110 (+14) has a string of 1's from 2^k to 2^m (k = 3, m = 1). The number can berepresented as $2^{k+1} - 2^{m}$= 16 - 2 = 14. Therefore, the multiplication M x 14, where M is the multiplicandand 14 the multiplier, can be done as M x 2^4- M x 2^1

Thus the product can be obtained by shifting the binary multiplicand M four times to the left andsubtracting M shifted left once. As in all multiplication schemes. Booth algorithm requires examination ofthe multiplier bits and shifting of the partial product. Prior to the shifting, the multiplicand may be added tothe partial product, subtracted from the partial product, or left unchanged according to the following rules:

1. The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1's in the multiplier.
2. The multiplicand is added to the partial product upon encountering the first 0 (provided that there was a previous 1) in a string of 0's in the multiplier.
3. The partial product does not change when the multiplier bit is identical to the previous multiplier bit.

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. For example, a multiplier equal to -14 is represented in 2's complement as 110010 and is treated as $-2^4 + 2^2 – 2^1 = –14$.

The hardware implementation of Booth algorithm requires the register configuration shown in Figure.

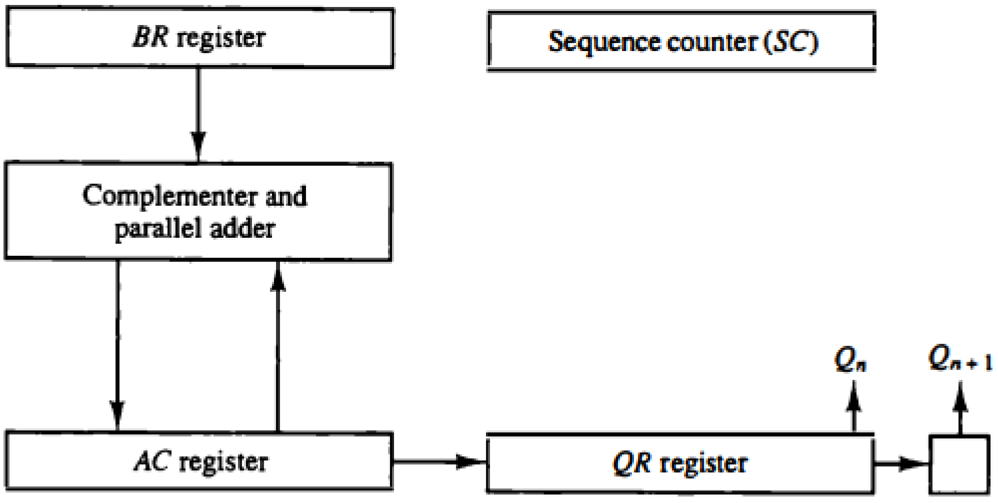


Figure 12.7 Hardware for Booth algorithm

We rename registers A, B, and Q, as AC, BR, and QR, respectively. Qn designates the least significant bit of the multiplier in register QR. An extra flip-flop Qn+1 is appended to QR to facilitate a double bit inspection of the multiplier. The flowchart for Booth algorithm is shown in Figure.

AC and the appended bit Qn+1 are initially cleared to 0 and the sequence counter SC is set to a number n equal to the number of bits in the multiplier. 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 l'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 twobits are equal, the partial product does not change. An overflow cannot occur because the addition andsubtraction of the multiplicand follow each other. As a consequence, the two numbers that are added alwayshave opposite signs, a condition that excludes an overflow. The next step is to shift right the partial product and the multiplier (including bit Qn+1). This is an arithmetic shift right (ashr) operation which shifts AC and QR to the right and leaves the sign bit in AC unchanged. The sequence counter is decremented and the computational loop is repeated n times.

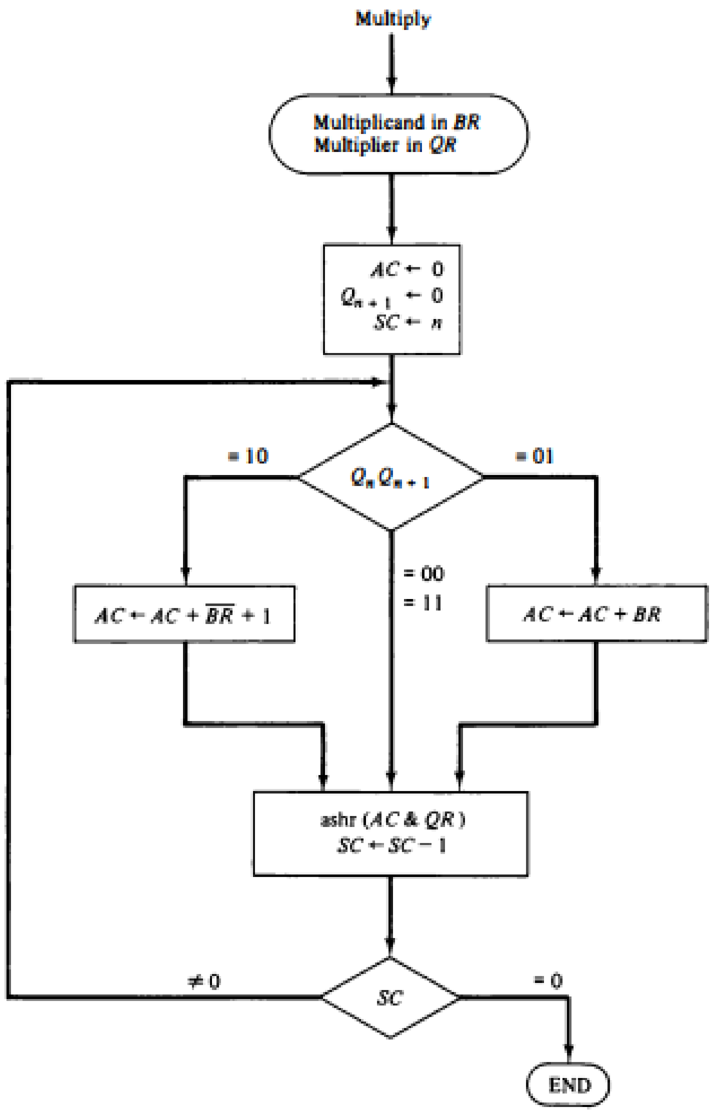


Figure 12.8 Booth Algorithm for multiplication of signed-2's complement numbers

A numerical example of Booth algorithm is shown in Table for n = 5. It shows the step-by-stepmultiplication of (-9) x (-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 Qn+1 is the original sign bit of the multiplier and should not be taken as part of the product.

TABLE 12.3 Example of Multiplication with Booth Algorithm

$Q_n Q_{n+1}$	$BR = 10111$ $\overline{BR} + 1 = 01001$	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

12.4 DIVISION ALGORITHM:

Binary division – simpler because 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

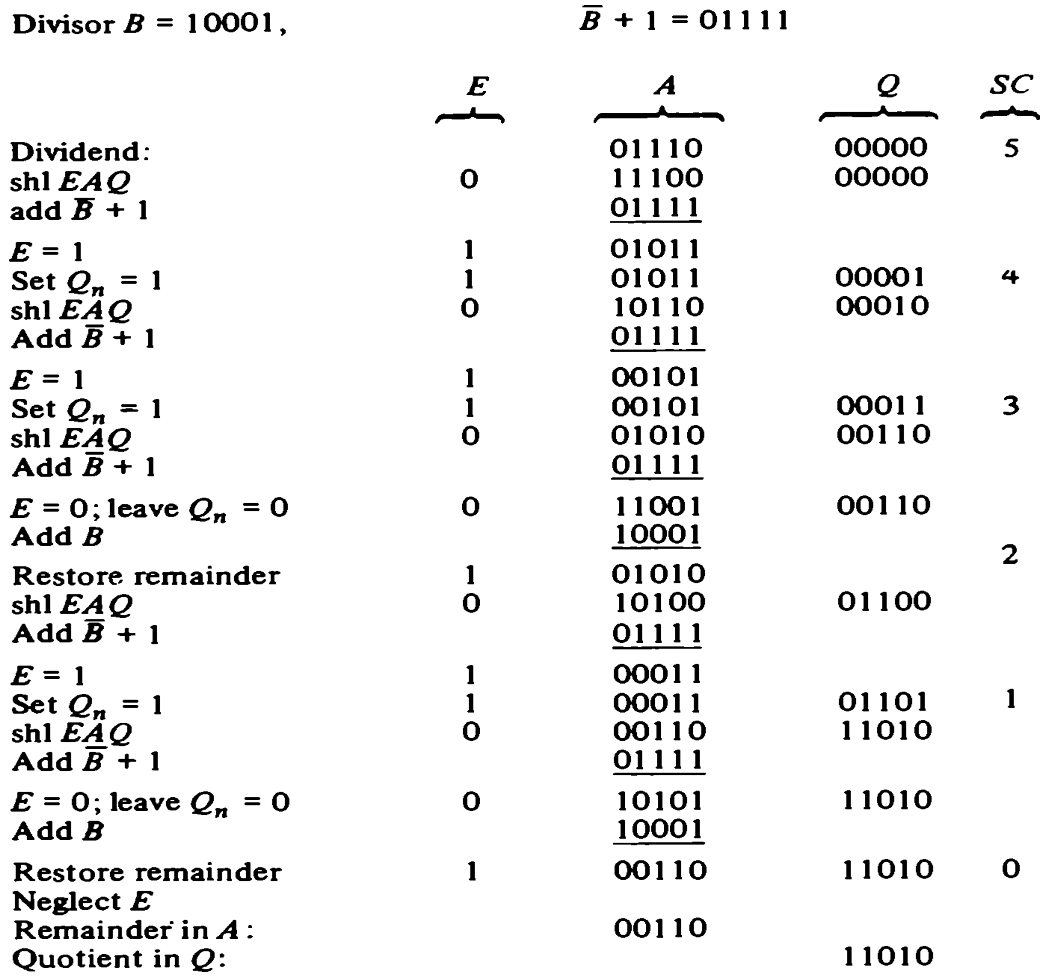
Division operation may result in a quotient with an overflow. Divide overflow flip flop (DVF) is used to detect overflow Divisor-B register, Dividend-A and Q register

If the signs of divisor and dividend are alike, the sign of the quotient is plus. Otherwise it is minus.

Best way to avoid divide over flow is to use floating point data.

Example for binary division:

Divisor:	11010	Quotient = Q
B = 10001	0111000000	Dividend = A
	01110	5 bits of $A < B$, quotient has 5 bits
	011100	6 bits of $A \geqslant B$
	-10001	Shift right B and subtract; enter 1 in Q
	-010110	7 bits of remainder $\geqslant B$
	--10001	Shift right B and subtract; enter 1 in Q
	--001010	Remainder $< B$; enter 0 in Q; shift right B
	---010100	Remainder $\geqslant B$
	----10001	Shift right B and subtract; enter 1 in Q
	----000110	Remainder $< B$; enter 0 in Q
	-----00110	Final remainder



Divisor B = 10001, $\overline{B}$ + 1 = 01111

	E	A	Q	SC
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add $\overline{B}$ + 1		01111		
E = 1	1	01011		
Set Q_n = 1	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\overline{B}$ + 1		01111		
E = 1	1	00101		
Set Q_n = 1	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\overline{B}$ + 1		01111		
E = 0; leave Q_n = 0	0	11001	00110	
Add B		10001		
Restore remainder	1	01010		2
shl EAQ	0	10100	01100	
Add $\overline{B}$ + 1		01111		
E = 1	1	00011		
Set Q_n = 1	1	00011	01101	1
shl EAQ	0	00110	11010	
Add $\overline{B}$ + 1		01111		
E = 0; leave Q_n = 0	0	10101	11010	
Add B		10001		
Restore remainder	1	00110	11010	0
Neglect E				
Remainder in A:		00110		
Quotient in Q:			11010	

Figure 12.9 Example of binary division with digital hardware

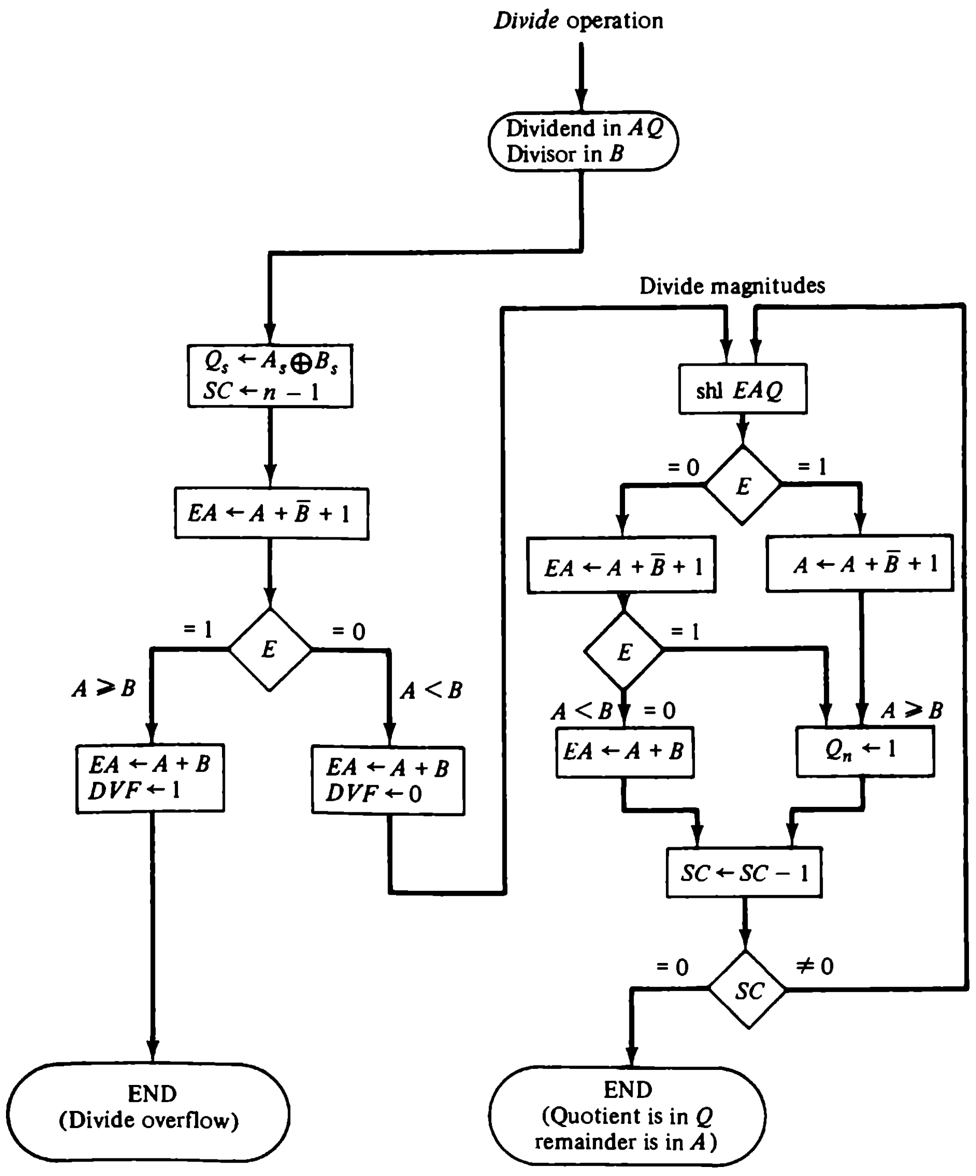


Fig.12.10 Flow chart for divide operation

12.5 FLOATING - POINT ARITHMETIC OPERATIONS:

- Numbers too large for standard integer representations or that have fractional components are usually represented in scientific notation, a form used commonly by scientists and engineers.
- Examples: 4.25×10^1
- Addition and subtraction are more complex than multiplication and division.
- Need to align mantissas
- Algorithm: Check for zeros-Align significant (adjusting exponents)–Add or subtract significant - Normalize result

$$F = m \times r^e$$

Where m: Mantissa, r: Radix, e: Exponent

It is necessary to make two exponents be equal before the mantissas can be added. We can either shift the first number three positions to the left, or shift thesecond number three positions to the right. When we store the mantissas in registers, shifting to the left causes a loss of most significant digits. Shifting to the right causes a loss of least significant digits. The second method is preferable because it only reduces the accuracy, while the first method may cause an error. The usual alignment procedure is to shift the mantissa that has the smaller exponent to the right by a number of places equal to the difference between the exponents. Now, the mantissas can be added.

$$\begin{array}{r} .5372400\text{x}10^{2} \\ +.0001580\text{x}\ 10^{2} \\ \hline .5373980\text{x}10^{2} \end{array}$$

When two normalized mantissas are added, the sum may contain an overflow digit. An overflow can be corrected easily by shifting the sum once to the right and incrementing the exponent. When two numbers are subtracted, the result may contain most significant zeros as shown in the following example:

$$\begin{array}{r} .56780 \times 10^{5} \\ -.56430 \times 10^{5} \\ \hline .00350 \times 10^{5} \end{array}$$

An underflowoccurs if a floating-point number that has a 0 in the most significant position of the mantissa. To normalize a number that contains an underflow, we shift the mantissa to the left and decrement the exponent until a nonzero digit appears inthe first position. Here, it is necessary to shift left twice to obtain .35000 x 10^{3}. In most computers a normalization procedure is performed after each operation to ensure that all results are in a normalized form.

Floating-point multiplication and division need not do an alignment of the mantissas. Multiplying the two mantissas and adding the exponents can form the product. Dividing the mantissas and subtracting the exponents perform division.

Register Configuration:

Three registers are there, BR, AC, and QR. Each register is subdivided into two parts. The mantissa part has the same uppercase letter symbols as in fixed-point representation. The exponent part may use corresponding lower-case letter symbol.

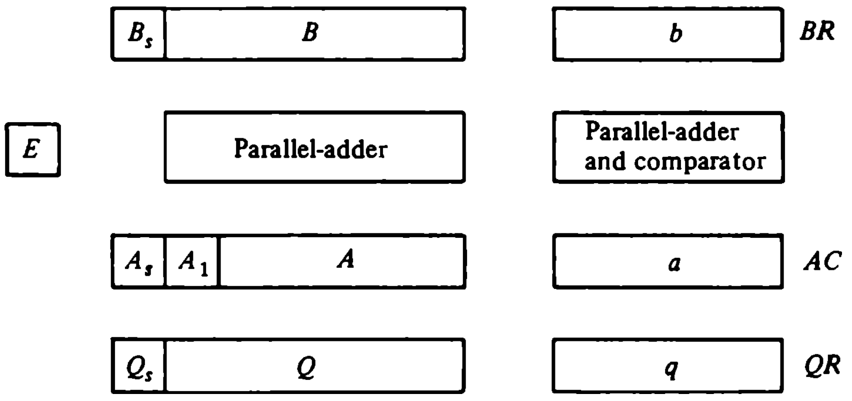


Fig.12.11 Registers for floating point arithmetic operations

Assuming that each floating-point number has a mantissa in signed-magnitude representation and a biased exponent. Thus the AC has a mantissa whose signis in A_s, and a magnitude that is in A. The diagram shows the most significant bit of A, labeled by A1.Thebit in his position must be a1to normalize the number. Note that the symbol AC represents the entire register, that is, the concatenation of A_s, A and a.In the similar way, register BR is subdivided into B_s, B, and b and QR into Q_s, Q and q.A parallel-adder adds the two mantissas and loads the sum into A and the carry into E.

Addition and Subtraction of Floating Point Numbers

During addition or subtraction, the two floating-point operand sare kept in AC and BR. The sum or difference is formed in the AC.

The algorithm can be divided into four consecutive parts:

1. Check for zeros.
2. Align the mantissas.
3. Add or subtract the mantissas
4. Normalize the result

A floating-point number cannot be normalized, if it is 0. If this number is used for computation, the result may also be zero. Instead of checking for zeros during the normalization process we check for zeros at the beginning and terminate the process if necessary. The alignment of the mantissas must be carried out prior to their operation. After the mantissas are added or subtracted, the result may be un-normalized. The normalization procedure ensures that the result is normalized before it is transferred to memory.

For adding or subtracting two floating-point binary numbers, if BR is equal tozero, the operation is stopped, with the value in the AC being the result. If AC = 0, we transfer the content of BR into AC and also complement its sign we have to subtract the numbers. If neither number is equal it to zero, we proceed to align the mantissas. The magnitude comparator attached to exponents a and b gives three outputs, which show their relative magnitudes.

If the two exponents are equal, we go to perform the arithmetic operation. If the exponents are not equal, the mantissa having the smaller exponent is shifted to the right and its exponent incremented. This process is repeated until two exponents are equal.

The addition and subtraction of the two mantissas is similar to the fixed-point addition and subtraction algorithm presented in the Figure 2.15

Multiplication:

The algorithm can be divided into four consecutive parts:

1. Check for zeros.
2. Add the exponents.
3. Multiply the mantissas
4. Normalize the result

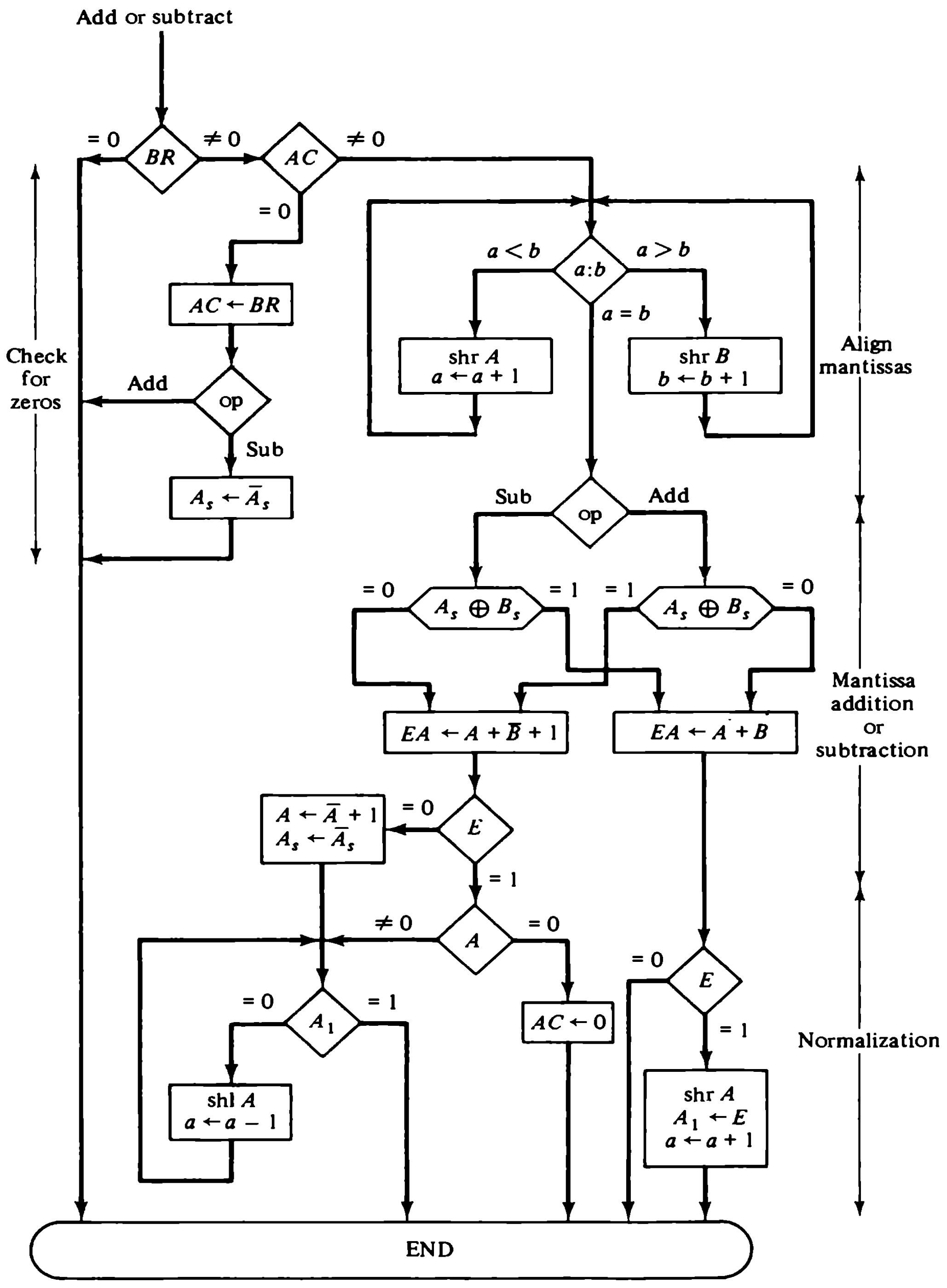


Fig.12.12 Addition and subtraction of floating point numbers

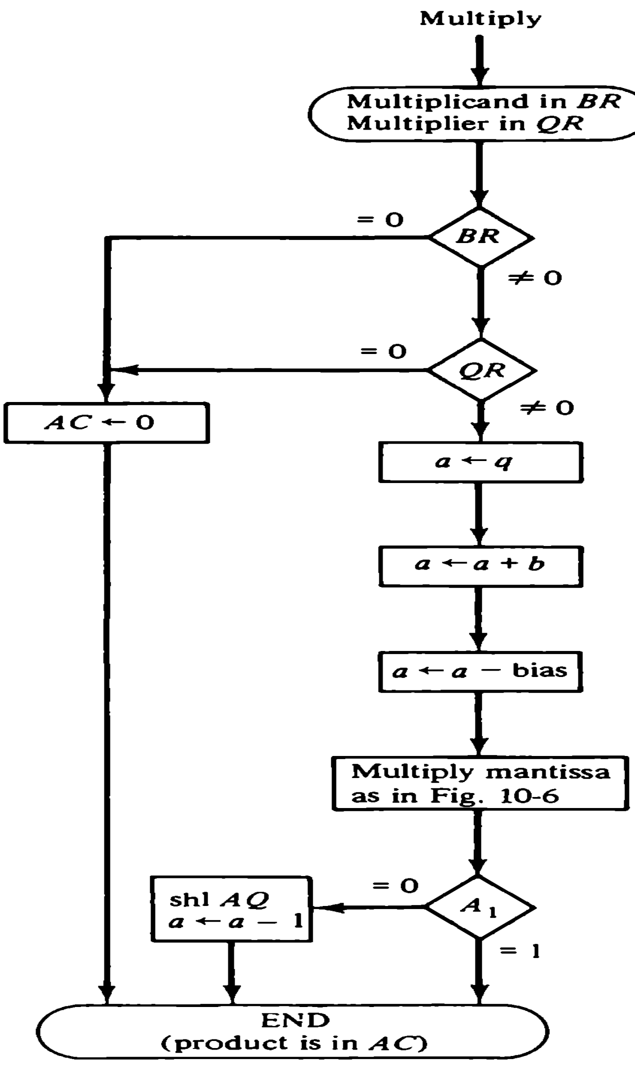


Fig.12.13 Multiplication of floating point numbers

Division:

The algorithm can be subdivided into five consecutive parts:

1. Check for zeros.
2. Initialize the register and evaluate the sign
3. Align the dividend
4. Subtract the exponents.
5. Divide the mantissa

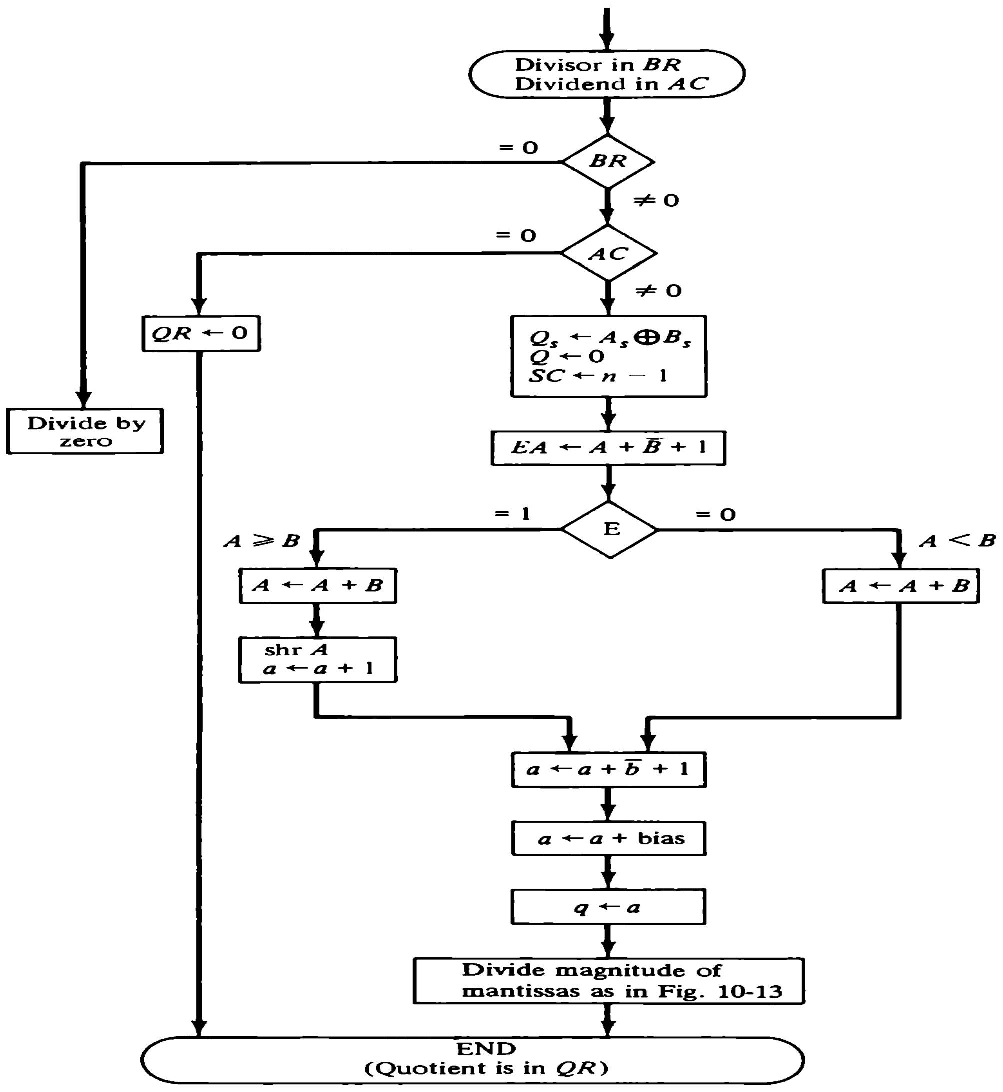


Fig.12.14 Division of floating point numbers

12.6 SUMMARY:

The chapter on Computer Arithmetic explores fundamental operations crucial to digital computation. It covers binary addition and subtraction, highlighting their simplicity and efficiency in a base-2 system. Multiplication algorithms, such as binary multiplication and Booth's algorithm, are discussed for their methods of efficiently calculating products. Division algorithms, including binary division and non-restoring division, are detailed for their approaches to computing quotients and remainders. Additionally, the chapter delves into floating-point arithmetic, explaining how real numbers are represented and manipulated using scientific notation, with emphasis on maintaining accuracy and precision through normalization and denormalization techniques. Overall, the chapter provides a comprehensive overview of the foundational arithmetic operations that underpin digital computing systems, ensuring reliable and efficient numerical computations across various applications.

12.7 TECHNICAL TERMS:

Computer arithmetic, Addition, Subtraction, Multiplication, Division Algorithms, Floating Point Arithmetic Operation.

12.8 SELF ASSESSMENT QUESTIONS:

Essay questions:

1) Explain in detail about booth multiplication algorithm with an example?
2) Design a Decimal Arithmetic Unit to perform BCD addition and BCD Subtraction?
3) Design hardware for signed magnitude addition and subtraction?
4) Explain the process for signed magnitude addition and subtraction with flow chart?

Short Questions:

1) Write short notes on 2's complement addition and subtraction operations?
2) Implement hardware for multiplying Two fixed- point binary numbers in signed-magnitude representation along with its flowchart?
3) Describe the importance of BCD in digital system design?
4) Design hardware for signed magnitude addition and subtraction?

12.9 FURTHER READINGS:

1) M.Morris Mano, “Computer System Architecture”, 3rd Edition, Pearson Education (2008).
2) V. Rajaraman, T. Radha Krishnan, “Computer Organization and Architecture”, PHI.
3) Behrooz Parhami, “Computer Architecture”, Oxford (2007).
4) ISRD group, “Computer Organization”, ace series, TMH (2007).
5) William Stallings, “Computer Organization and Architecture-Designing for Performance”, Pearson Education (2005).

Dr. Kampa Lavanya

LESSON-13

INPUT-OUTPUT ORGANIZATION

OBJECTIVES:

After going through this lesson, you will be able to

- Recognize various types of peripheral devices
- Learn about different types of input-output interfaces and their design
- Understand the concept of asynchronous data transfer
- Explore various modes of data transfer
- Comprehend the concept of priority interrupt

STRUCTURE OF THE LESSION:

13.1 PERIPHERAL DEVICES:

The input and output devices that are attached to the computer are called peripheral devices.

Eg: Keyboard, printer, display unit.

Peripherals that provide auxiliary storage to the system are magnetic disks and tapes. Printers provide a permanent record on paper of computer output data or text. The 3 types of printers are Daisy wheel, Dot matrix and Laser

Magnetic tapes are used for storing files of data.

- Access-Sequential
- Slowest and cheapest method.
- Tapes can be removed when not in use.

Magnetic disks are used for bulk storage of programs and data.

- Access-by moving the read/write mechanism to a track in the magnetized surface.

Input and output devices that communicate with people and the computer are usually involved in the transfer of alphanumeric information to and from the device and the computer. The standard binary code for the alphanumeric characters is ASCII (American Standard Code for Information Interchange). It uses seven bits to code 128 characters as shown in Table 13.1. The seven bits of the code are designated by b1 through b7, with b, being the most significant bit. The letter A, for example, is represented in ASCII as 1000001 (column 100, row 0001). The ASCII code contains 94 characters that can be printed and 34 nonprinting characters used for various control functions. The printing characters consist of the 26 uppercase letters A through Z, the 26 lowercase letters, the 10 numerals 0 through 9, and 32 special printable characters such as %, *, and $.

There are three types of control characters: format effectors, information separators, and communication control characters

ASCII is a 7-bit code, but most computers manipulate an 8-bit quantity as a single unit called a byte.

13.2 INPUT-OUTPUT INTERFACE:

Input-output interface provides a method for transferring information between internal storage and external I/O devices. Peripherals connected to a computer need special communication links for interfacing them with the central processing unit.

The major differences are:

1) Peripherals are electromechanical and electromagnetic devices and their manner of operation is different from the operation of the CPU and memory, which are electronic devices. Therefore, a conversion of signal values may be required.
2) The data transfer rate of peripherals is usually slower than the transfer rate of the CPU, and consequently, a synchronization mechanism may be needed.
3) Data codes and formats in peripherals differ from the word format in the CPU and memory.
4) The operating modes of peripherals are different from each other and each must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

A typical communication link between the processor and several peripherals is shown in figure. The I/O bus consists of data lines, address lines, and control lines. The magnetic disk, printer, and terminal are employed in practically any general-purpose computer. The interface selected responds to the function code and proceeds to execute it. The function code is referred to as an I/O command and is in essence an instruction that is executed in the interface and its attached peripheral unit.

There are three ways that computer buses can be used to communicate with memory and I/O:

1) Use two separate buses, one for memory and the other for I/O.

2) Use one common bus for both memory and I/O but have separate control lines for each.
3) Use one common bus for memory and I/O with common control lines.

Isolated I/O versus Memory-Mapped I/O

Many computers use one common bus to transfer information between memory or I/O and the CPU. In the isolated I/O configuration, the CPU has distinct input and output instructions, and each of these instructions is associated with the address of an interface register. The isolated I/O method isolates memory and I/O addresses so that memory address values are not affected by interface address assignment since each has its own address space. The other alternative is to use the same address space for both memory and I/O. This is the case in computers that employ only one set of read and write signals and do not distinguish between memory and I/O addresses. This configuration is referred to as memory- mapped I/O. In a memory-mapped I/O organization there is no specific input or output instructions. Computers with memory-mapped I/O can use memory-type instructions to access I/O data.

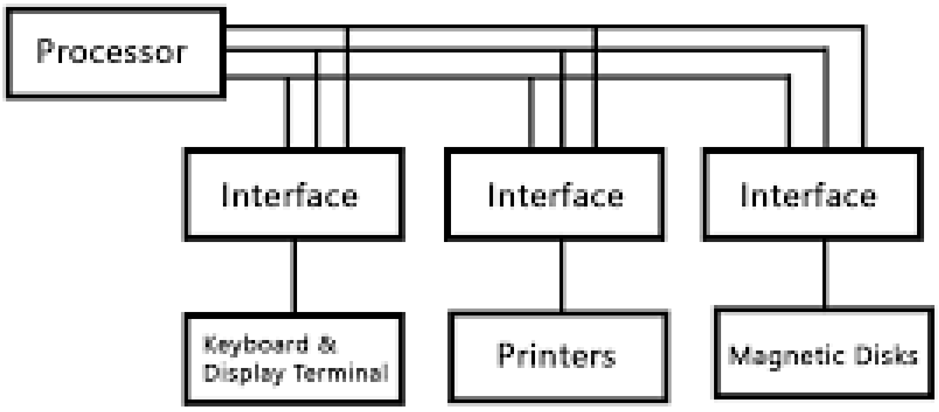
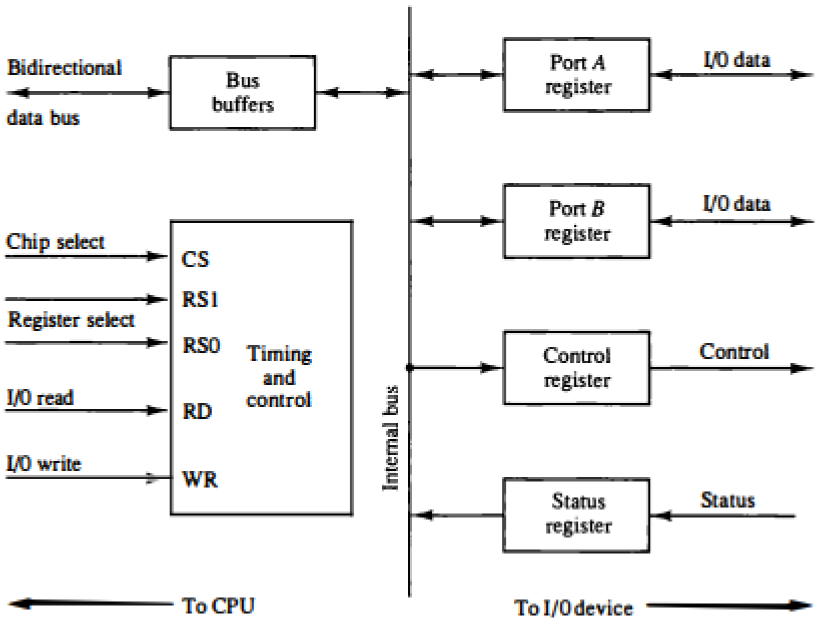


Figure 13.1 Connection of input/output devices

An example of an I/O interface unit is shown in block diagram form in Fig.13.1. It consists of two data registers called ports, a control register, a status register, bus buffers, and timing and control circuits. The interface communicates with the CPU through the data bus. The chip select and register select inputs determine the address assigned to the interface. The I/O read and write are two control lines that specify an input or output, respectively. The four registers communicate directly with the I/O device attached to the interface.



CS	RS1	RS0	Register selected
0	×	×	None: data bus in high-impedance
1	0	0	Port *A* register
1	0	1	Port *B* register
1	1	0	Control register
1	1	1	Status register

Figure 13.2 Example of I/O interface unit

13.3 ASYNCHRONOUS DATA TRANSFER:

The internal operations in a digital system are synchronized by means of clock pulses supplied by a common pulse generator. If the registers in the interface share a common clock with the CPU registers, the transfer between the two units is said to be synchronous. In most cases, the internal timing in each unit is independent from the other in that each uses its own private clock for internal registers. In that case, the two units are said to be asynchronous to each other. This approach is widely used in most computer systems. Asynchronous data transfer between two independent units requires that control signals be transmitted between the communicating units to indicate the time at which data is being transmitted.

Two way of achieving this:

- The strobe: pulse supplied by one of the units to indicate to the other unit when the transfer has to occur.
- The handshaking: The unit receiving the data item responds with another control signal to acknowledge receipt of the data.

The strobe pulse method and the handshaking method of asynchronous data transfer are not restricted to I/O transfers.

The strobe may be activated by either the source or the destination unit. Figure 38 shows a source-initiated transfer and the timing diagram.

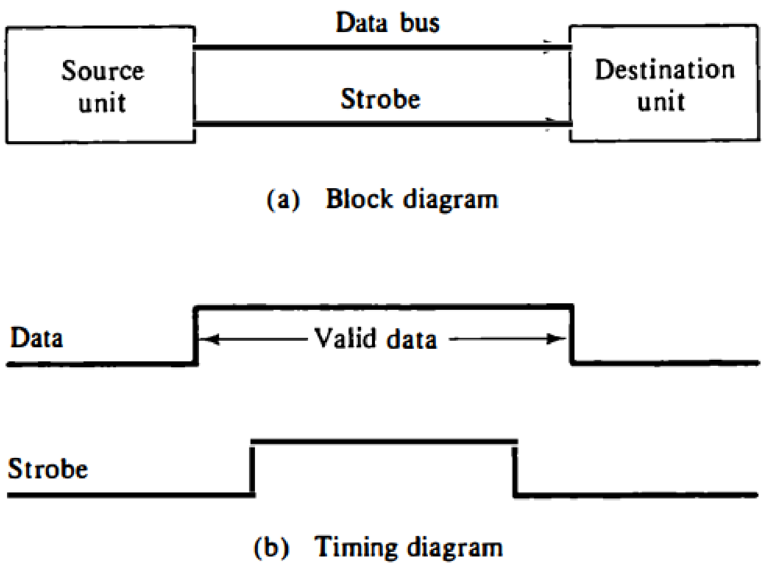


Figure: 13.3 Source initiated strobe for data transfer

Fig.13.4 shows the strobe of a memory-read control signal from the CPU to a memory

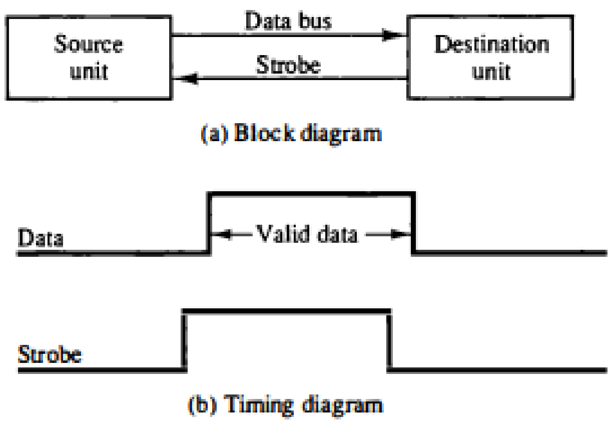


Figure 13.4 Destination initiated strobe for data transfer

Handshaking:

The disadvantage of the strobe method is that the source unit that initiates the transfer has no way of knowing whether the destination unit has actually received the data item that was placed in the bus. The handshake method solves this problem by introducing a second control signal that provides a reply to the unit that two-wire control initiates the transfer.

Figure 13.5 shows the data transfer procedure when initiated by the source. The two handshaking lines are data valid, which is generated by the source unit, and data accepted, generated by the destination unit. The timing diagram shows the exchange of signals between the two units.

Figure 13.6 the destination-initiated transfer using handshaking lines. Note that the name of the signal generated by the destination unit has been changed to ready for data to reflect its new meaning.

Asynchronous Serial Transfer:

The transfer of data between two units may be done in parallel or serial. In parallel data transmission, each bit of the message has its own path and the total message is transmitted at the same time. This means that a w-bit message must be transmitted through n separate conductor paths. In serial data transmission, each bit in the message is sent in sequence one at a time. This method requires the use of one pair of conductors or one conductor and a common ground. Parallel transmission is faster but requires many wires. It is used for short distances and where speed is important. Serial transmission is slower but is less expensive since it requires only one pair of conductors. Serial transmission can be synchronous or asynchronous.

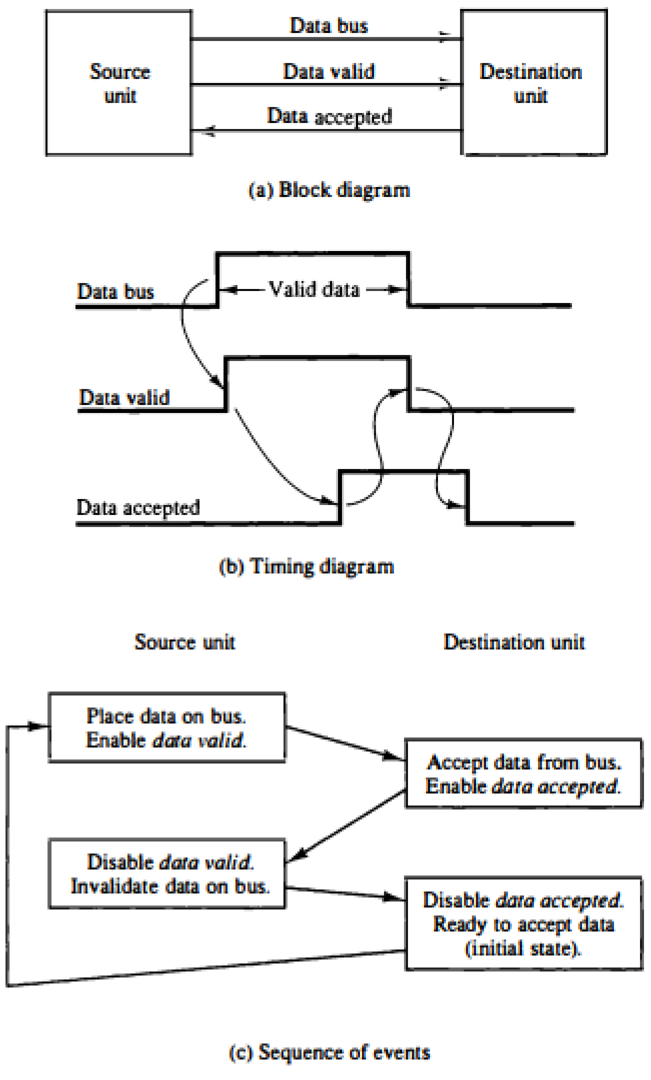
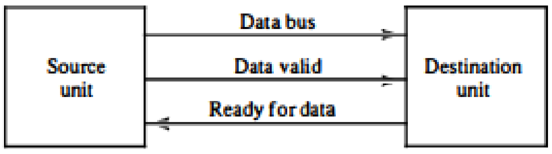

Source unit
Data bus
Data valid
Data accepted
Destination unit
(a) Block diagram
Data bus
Valid data
Data valid
Data accepted
(b) Timing diagram
Source unit
Destination unit
Place data on bus. Enable *data valid*.
Accept data from bus. Enable *data accepted*.
Disable *data valid*. Invalidate data on bus.
Disable *data accepted*. Ready to accept data (initial state).
(c) Sequence of events

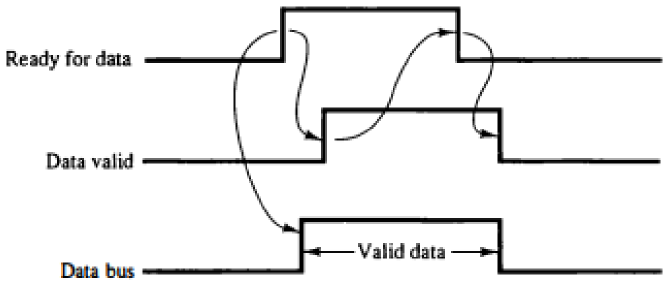
Figure 13.5 Source initiated transfer using handshaking

A transmitted character can be detected by the receiver from knowledge of the transmission rules:

1. When a character is not being sent, the line is kept in the 1-state.
2. The initiation of a character transmission is detected from the start bit, which is always (0).
3. The character bits always follow the start bit.
4. After the last bit of the character is transmitted, a stop bit is detected when the line returns to the 1-state for at least one bit time.



(a) Block diagram



(b) Timing diagram

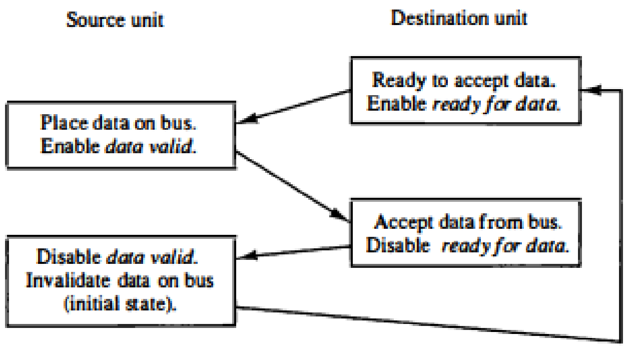

Source unit
Destination unit
Ready to accept data. Enable *ready for data.*
Place data on bus. Enable *data valid.*
Accept data from bus. Disable *ready for data.*
Disable *data valid.* Invalidate data on bus (initial state).

Figure 13.6 Destination initiated transfer using handshaking

An example of serial transmission format is shown in Fig. 13.7

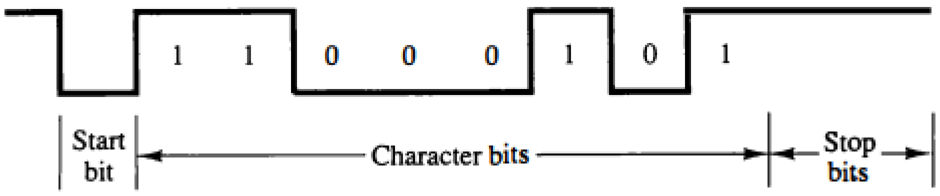


Figure 13.7 Asynchronous serial transmission

As an illustration, consider the serial transmission of a terminal whose transfer rate is 10 characters per second. Each transmitted character consists of a start bit, eight information bits, and two stop bits, for a total of 11 bits. Ten characters per second means that each character takes 0.1s for transfer. Since there are 11 bits to be transmitted, it follows that the bit time is 13.09ms. The baud rate is defined as the rate at which serial information is transmitted and is equivalent to the data transfer in bits per second. Ten characters per second with an 11-bit format have a transfer rate of 110 baud. Integrated circuits are available which are specifically designed to provide the interface between computer and similar interactive terminals. Such a circuit is called an asynchronous communication interface or a Universal Asynchronous Receiver-Transmitter (UART).

13.4 MODES OF TRANSFER:

Binary information received from an external device is usually stored in memory for later processing. Information transferred from the central computer into an external device originates in the memory unit. The CPU merely executes the I/O instructions and may accept the data temporarily, but the ultimate source or destination is the memory unit. Data transfer between the central computer and I/O devices may be handled in a variety of modes. Some modes use the CPU as an intermediate path; others transfer the data directly to and from the memory unit.

Data transfer to and from peripherals may be handled in one of three possible modes:

1. Programmed I/O
2. Interrupt-initiated I/O
3. Direct memory access (DMA)

Programmed I/O:

Each data item is initiated by an instruction in the program. The transfer is to and from a CPU register and peripheral. Transferring data under program control requires constant monitoring of the peripheral by the CPU. The CPU stays in a program loop until the I/O unit indicates that it is ready for data transfer. Time consuming process since it keeps the processor busy needlessly.

Interrupt initiated I/O:

The interface monitors the device. When the device is ready for data transfer, the interface generates an interrupt request to the computer. Upon detecting the external interrupt signal, the CPU momentarily stops the task it is processing, branches to a service program to process the I/O transfer and then returns to the task it was originally performing.

DMA:

The interface transfers data into and out of the memory unit through the memory bus. The CPU initiates the transfer by supplying the interface with the starting address and the number of words needed to be transferred and then proceeds to execute other tasks. When the transfer is made, the DMA requests memory cycles through the memory bus. When the request is granted by the memory controller, the DMA transfers the data directly into memory. The CPU delays its memory access operation to allow the direct memory I/O transfer.\

13.5 PRIORITY INTERRUPT:

In a typical application, a number of I/O devices are attached to the computer, with each device being able to origin a tean interrupt request. The first task of the interrupt system is to identify the source of the interrupt. There is also the possibility that several sources will request service simultaneously. In this case the system must also decide which device to service first.

A priority interrupt is a system that establishes a priority over the various sources to determine which condition is to be services first when two or more requests arise simultaneously. Higher priority interrupt levels are assigned to requests which, if delayed or interrupted could have serious consequences. Devices with high speed transfers such as magnetic disks are given high priority and slow devices such as keyboards receive low priority.

When two devices interrupt the computer at the same time, the computer services the devices with the higher priority first.

Establishing the priority of simultaneous interrupts can be done by software or hardware:

Software: Polling

Hardware: Daisy chain, Parallel Priority

Polling: A polling procedure is used to identify the highest-priority source by software means.

In this method, there is one common branch address for all interrupts. The program that takes care of interrupts begins at the branch address and polls the interrupt sources in sequence. The order in which they are tested determines the priority of each interrupt. The disadvantage of the software method is that if there are many interrupts, the time required to poll them can exceed the time available to service the I/O device. A hardware priority interrupt unit can be used to speed up the operation.

A hardware priority-interrupt unit functions as an overall manager in an interrupt system environment. It accepts interrupt requests from many sources, determines which of the incoming requests has the highest priority, and issues an interrupt request to the computer based on the determination.

Daisy-Chaining Priority: The daisy-chaining method of establishing priority consists of a serial connection of all devices that request an interrupt. The device with the highest priority is placed in the first position, followed by the lower-priority devices up to the device with the lowest priority, which is placed last in the chain.

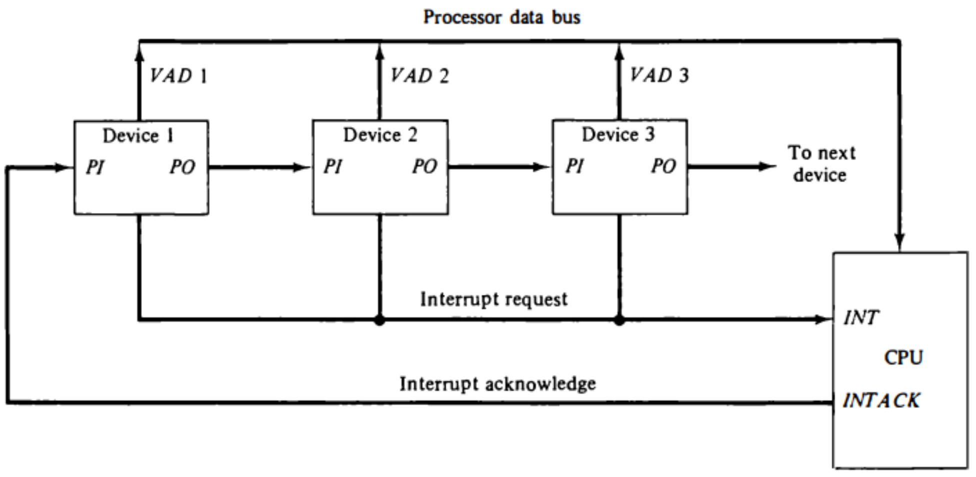


Figure 13.8 Daisy chain priority interrupt

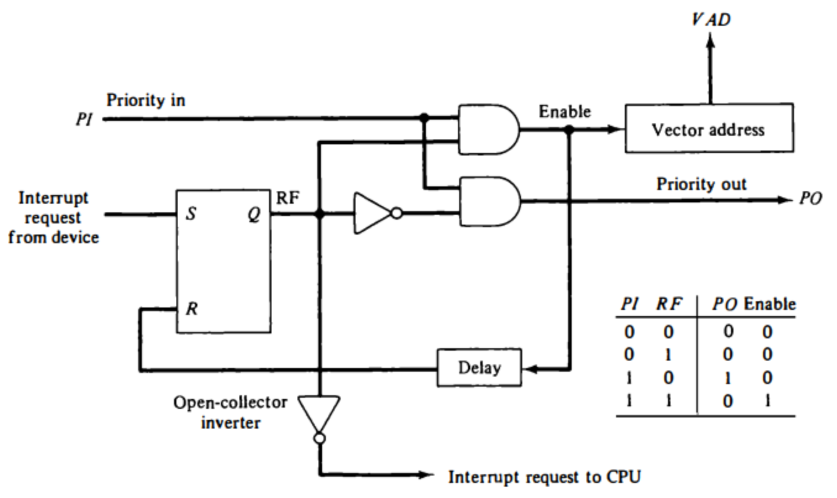
The interrupt request line is common to all devices and forms a wired logic connection. If any device has an interrupt signal in the low-level state, the interrupt line goes to the low-level state and enables the interrupt input in the CPU. When no interrupts are pending, the interrupt line stays in the high-level state and no interrupts are recognized by the CPU.

- This is equivalent to the negative logic OR operation.

The CPU responds to an interrupt request by enabling the interrupt acknowledge line. This signal is received by device 1 at its PI (Priority In) input. The acknowledge signal passes on to the next device through the PO (Priority Out) output only if device 1 is not requesting an interrupt. If device 1 has a pending interrupt, it blocks the acknowledge signal from the next device by placing a 0 in the PO output.

Parallel Priority Interrupt:

The parallel priority interrupt method uses a register whose bits are set separately by the interrupt signal from each device. Priority is established according to the position of the bits in the register. Mask register is used to provide facility for the higher priority devices to interrupt when lower priority device is being serviced or disable all lower priority devices when higher is being services.



PI	RF	PO	Enable
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

Figure 13.9: Priority Interrupt Hardware

It consists of:

- An interrupt register whose individual bits are set by external conditions and cleared by program instructions.

The magnetic disk, being a high-speed device is given the highest priority.

- The mask register has the same number of bits as the interrupt register. By means of program instructions, it is possible to set or reset any bit in the mask register.
- Each interrupt bit and it corresponding mask bit are applied to an AND gate to produce the four inputs to a priority encoder.

In this way an interrupt is recognized only if its corresponding mask bit is set to 1 by the program. The priority encoder generates two bits of the vector address, which is transferred to the CPU.

- Another output from the encoder sets an interrupt status flip-flop IST when an interrupt that is not masked occurs.
- The interrupt enable flip-flop IEN can be set or cleared by the program to provide an overall control over the interrupt system.
- The outputs of IST ANDed with IEN provide a common interrupt signal for the CPU.
- The interrupt acknowledge INTACK signal from the CPU enables the bus buffers in the output register and a vector address VAD is placed into the data bus.

The priority encoder is a circuit that implements the priority function. The logic of the priority encoder is such that if two or more inputs arrive at the same time, the input having the highest priority will take precedence.

13.6 SUMMARY:

This chapter examines the various components and processes involved in managing data exchange between a computer and its external devices. It covers peripheral devices, which serve as the primary means for input and output operations, and the input-output interface, which facilitates communication between these devices and the central processing unit. The chapter delves into asynchronous data transfer, highlighting methods to handle data transmission without synchronized clocks. It also explores different modes of transfer, such as programmed I/O, interrupt-driven I/O, and direct memory access (DMA), each offering unique trade-offs in terms of efficiency and complexity. Lastly, it addresses priority interrupts, mechanisms to prioritize and manage multiple simultaneous interrupt signals to ensure efficient processing and system responsiveness.

13.7 TECHNICAL TERMS:

Input-Output Interface, Asynchronous Data Transfer, Modes of Transfer, Priority Interrupt.

13.8 SELF ASSESSMENT QUESTIONS:

Essay questions:

1) With practical Examples, Explain the connection of I/O bus to input-output devices and its mapping Specifications.
2) What is the disadvantage of strobe method? Explain how handshake method solves the problem?
3) Classify and describe the possible modes of data transfer to and from peripherals with examples.
4) Explain Daisy-Chaining priority and Parallel priority Interrupt with its hardware diagram
5) With a neat schematic, Explain about DMA controller and its mode of data transfer

Short Questions:

1) List out the importance of interfacing.
2) Explain in detail about strobe control method of asynchronous data transfer?
3) List out the methodologies involved in handling Priority Interrupt
4) Elaborate how DMA bypasses CPU and speeds up the memory operation?

13.9 FURTHER READINGS:

1) M.Morris Mano, “Computer System Architecture”, 3rd Edition, Pearson Education (2008).
2) V. Rajaraman, T. Radha Krishnan, “Computer Organization and Architecture”, PHI.
3) Behrooz Parhami, “Computer Architecture”, Oxford (2007).
4) ISRD group, “Computer Organization”, ace series, TMH (2007).
5) William Stallings, “Computer Organization and Architecture-Designing for Performance”, Pearson Education (2005).

Dr. Vasantha Rudramalla

LESSON-14
MEMORY ORGANIZATION

OBJECTIVES:

After going through this lesson, you will be able to

- Grasp the concept and significance of the memory hierarchy.
- Differentiate between various types of main memory.
- Define auxiliary memory and its importance for long-term data storage.
- Identify scenarios and applications where associative memory is particularly beneficial.
- Explore different cache mapping techniques.

STRUCTURE OF THE LESSION:

14.1 MEMORY HIERARCHY:

Memory hierarchy in a computer system:

Memory hierarchy system consists of all storage devices employed in a computer system from the slow but high capacity auxiliary memory to a relatively faster main memory, to an even smaller and faster cache memory accessible to the high speed processing logic.

- **Main Memory**: Memory unit that communicates directly with the CPU (RAM)
- **Auxiliary Memory**: Device that provide backup storage (Disk Drives)
- **Cache Memory**: Dpecial very-high-speed memory to increase the processing speed (Cache RAM).

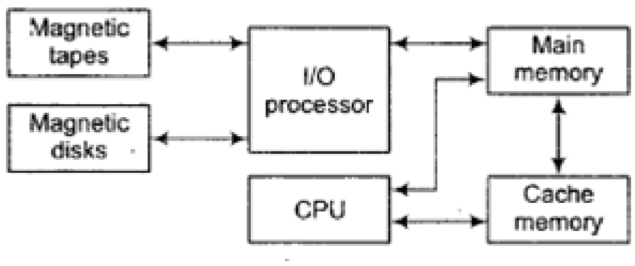


Figure 14.1 Memory Hierarchy in computer system

Figure 14.1 illustrates the components in a typical memory hierarchy. At the bottom of the hierarchy are the relatively slow magnetic tapes used to store removable files. Next are the Magnetic disks used as backup storage. The main memory occupies a central position by being able to communicate directly with CPU and with auxiliary memory devices through an I/O process. Program not currently needed in main memory are transferred into auxiliary memory to provide space for currently used programs and data.

The cache memory is used for storing segments of programs currently being executed in the CPU. The I/O processor manages data transfer between auxiliary memory and main memory. The auxiliary memory has a large storage capacity is relatively inexpensive, but has low access speed compared to main memory. The cache memory is very small, relatively expensive, and has very high access speed. The CPU has direct access to both cache and main memory but not to auxiliary memory.

Multiprogramming:

Many operating systems are designed to enable the CPU to process a number of independent programs concurrently.

Multiprogramming refers to the existence of 2 or more programs in different parts of the memory hierarchy at the same time.

Memory management System:

The part of the computer system that supervises the flow of information between auxiliary memory and main memory.

14.2 MAIN MEMORY:

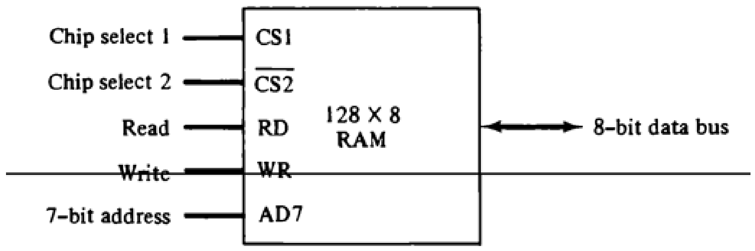
Main memory is the central storage unit in a computer system. It is a relatively large and fast memory used to store programs and data during the computer operation. The principal technology used for the main memory is based on semi conductor integrated circuits. Integrated circuits RAM chips are available in two possible operating modes, static and dynamic.

- Static RAM-Consists of internal flip-flops that store the binary information.
- Dynamic RAM-Stores the binary information in the form of electric charges that are applied to capacitors.
- Most of the main memory in a general purpose computer is made up of RAM integrated circuit chips, but a portion of the memory may be constructed with ROM chips.

- Read Only Memory-Store programs that are permanently resident in the computer and for tables of constants that do not change in value once the production of the computer is completed.
- The ROM portion of main memory is needed for storing an initial program called a Boot strap loader.
- Boots trap loader-function is start the computer software operating when power is turned on.
- Boot strap program loads a portion of operating system from disc to main memory and control is then transferred to operating system.

RAM and ROM CHIP:

RAM chip-utilizes bidirectional data bus with three state buffers to perform communication with CPU.



(a) Block diagram

CS1	$\overline{CS2}$	RD	WR	Memory function	State of data bus
0	0	×	×	Inhibit	High-impedance
0	1	×	×	Inhibit	High-impedance
1	0	0	0	Inhibit	High-impedance
1	0	0	1	Write	Input data to RAM
1	0	1	×	Read	Output data from RAM
1	1	×	×	Inhibit	High-impedance

(b) Function table

Figure 14.2 Typical RAM Chip

The block diagram of a RAM Chip is shown in Fig.14.2. The capacity of memory is 128 words of eight bits (one byte) per word. This requires a 7-bit address and an 8-bit bidirectional data bus. The read and write inputs specify the memory operation and the two chips select (CS) control inputs are enabling the chip only when it is selected by the microprocessor. The read and write inputs are sometimes combined into one line labelled R/W.

The function table listed in Fig. 14.2(b) specifies the operation of the RAM chip. The unit is in operation only when CS1=1 and CS2=0. The bar on top of the second select variable indicates that this input is enabled when it is equal to 0. If the chip select inputs are not enabled, or if they are enabled but the read or write inputs are not enabled, the memory is inhibited and its data bus is in a high-impedance state. When CS1=1 and CS2=0, the memory can be placed in a write or read mode. When the WR input is enabled, the memory stores a byte from the data bus into a location specified by the address input lines. When the

RD input is enabled, the content of the selected byte is placed into the data bus. The RD and WR signals control the memory operation as well as the bus buffers associated with the bidirectional data bus.

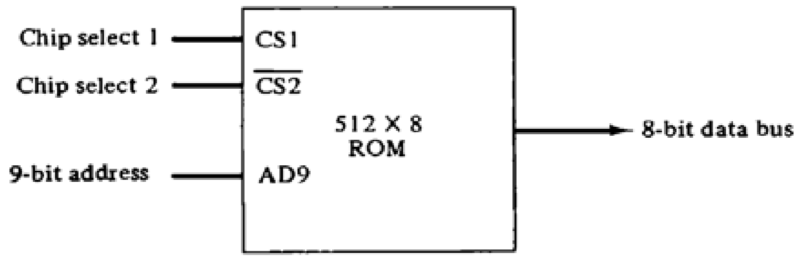


Figure 14.3 Typical ROM Chip

A ROM chip is organized externally in a similar manner. However, since a ROM can only read, the data bus can only be in an output mode. The block diagram of a ROM chip is shown in fig.14.3. The nine address lines in the ROM chip specify any one of the512 bytes stored in it. The two chip select inputs must be CS1=1 and CS2=0 for the unit to operate. Otherwise, the data bus is in a high-impedance state.

Memory Address Map:

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 specify the memory address assigned to each chip. The table called Memory address map, is a pictorial representation of assigned address space for each chip in the system.

Table 14.1 Memory Address Map for Microprocessors

Component	Hexadecimal address	Address bus									
		10	9	8	7	6	5	4	3	2	1
RAM 1	0000–007F	0	0	0	x	x	x	x	x	x	x
RAM 2	0080–00FF	0	0	1	x	x	x	x	x	x	x
RAM 3	0100–017F	0	1	0	x	x	x	x	x	x	x
RAM 4	0180–01FF	0	1	1	x	x	x	x	x	x	x
ROM	0200–03FF	1	x	x	x	x	x	x	x	x	x

The memory address map for this configuration is shown in table 14.1. 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. The RAM chips have 128 bytes and need seven address lines. The ROM chip has 512 bytes and needs 9 address lines.

Memory Connection 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.

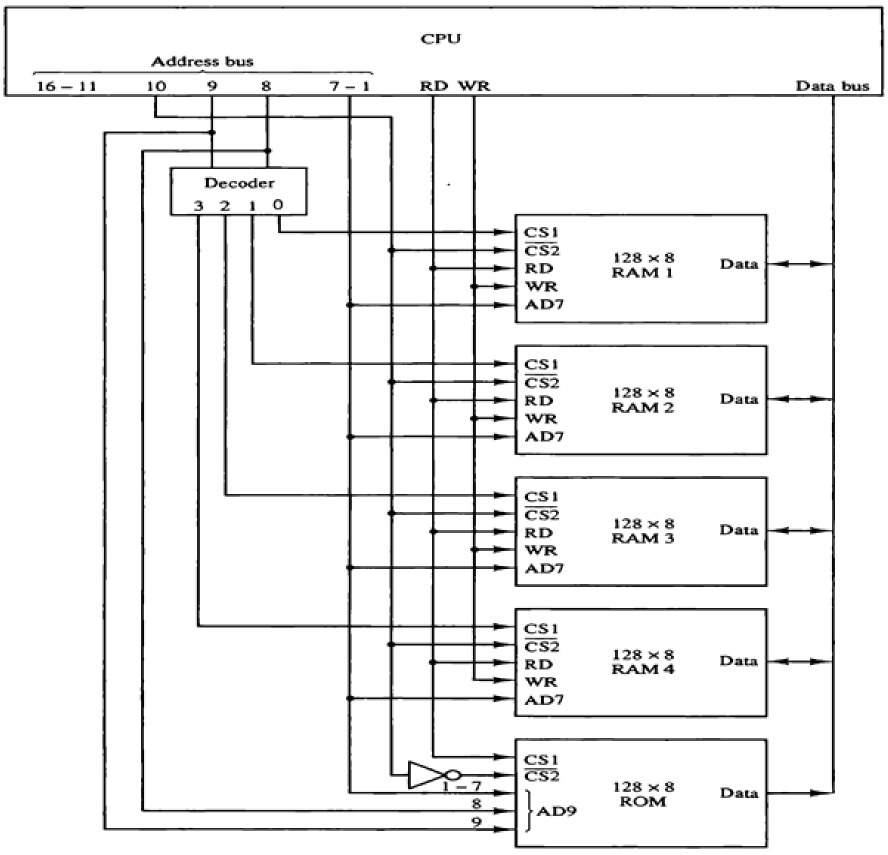


Figure 14.4 Memory connection to the CPU

The connection of memory chips to the CPU is shown in Fig.14.4. This configuration gives a memory capacity of 512 bytes of RAM and 512 bytes of ROM. Each RAM receives the seven low-order bits of the address bus to select one of 128 possible bytes. The particular RAM chip selected is determined rom lines 8 and 9 in the address bus. This is done through a 2X4 decoder whose outputs go to the CS1inputs in each RAM chip. Thus, when address lines 8 and 9 are equal to 00, the first RAM chip is selected. When 01, the second RAM chip is select, and so on. The RD and WR outputs from the microprocessor are applied to the inputs of each RAM chip. The selection between RAM and ROM is achieved through bus line 14. The RAMs are selected when the bit in this line is 0 and the ROM when the bit is 1. Address bus lines 1 to 9 are applied to the input address of ROM without going through the decoder. The data bus of the ROM has only an output capability, whereas the data bus connected to the RAMs can transfer information in both directions.

14.3 AUXILIARY MEMORY:

An Auxiliary memory is known as the lowest-cost, highest-capacity and slowest-access storage in a computer system. It is where programs and data are kept for long-term storage or when not in immediate use. The most common examples of auxiliary memories are magnetic tapes and magnetic disks.

Magnetic Disks:

A magnetic disk is a type of memory constructed using a circular plate of metal or plastic coated with magnetized materials. Usually, both sides of the disks are used to carry out read/write operations. However, several disks may be stacked on one spindle with read/write head available on each surface.

The following image shows the structural representation for a magnetic disk.

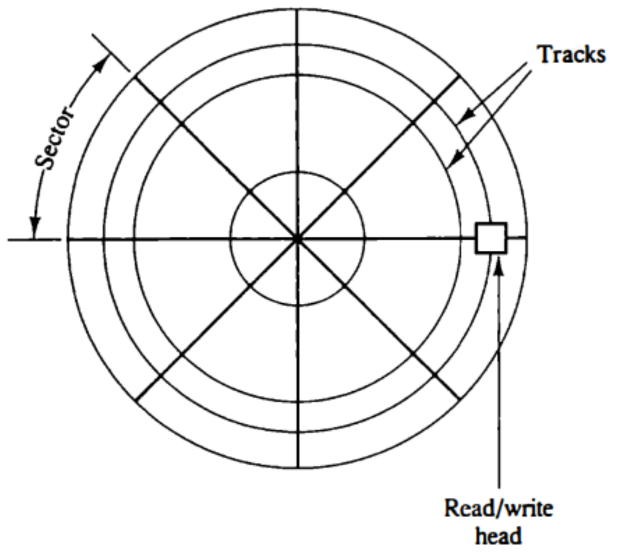


Figure 14.5 magnetic disk

The memory bits are stored in the magnetized surface in spots along the concentric circles called tracks.

The concentric circles (tracks) are commonly divided into sections called sectors.

Magnetic Tape:

Magnetic tape is a storage medium that allows data archiving, collection, and backup for different kinds of data. The magnetic tape is constructed using a plastic strip coated with a magnetic recording medium.

The bits are recorded as magnetic spots on the tape along several tracks. Usually, seven or nine bits are recorded simultaneously to form a character together with a parity bit.

Magnetic tape units can be halted, started to move forward or in reverse, or can be rewound. However, they cannot be started or stopped fast enough between individual characters. For this reason, information is recorded in blocks referred to as records.

14.4 ASSOCIATIVE MEMORY:

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).

- CAM is accessed simultaneously and in parallel on the basis of data content rather than by specific address or location.
- Associative memory is more expensive than a RAM because each cell must have storage capability as well as logic circuits.
- Argument register-holds an external argument for content matching.
- Key register-mask for choosing a particular field or key in the argument word.

Hardware Organization

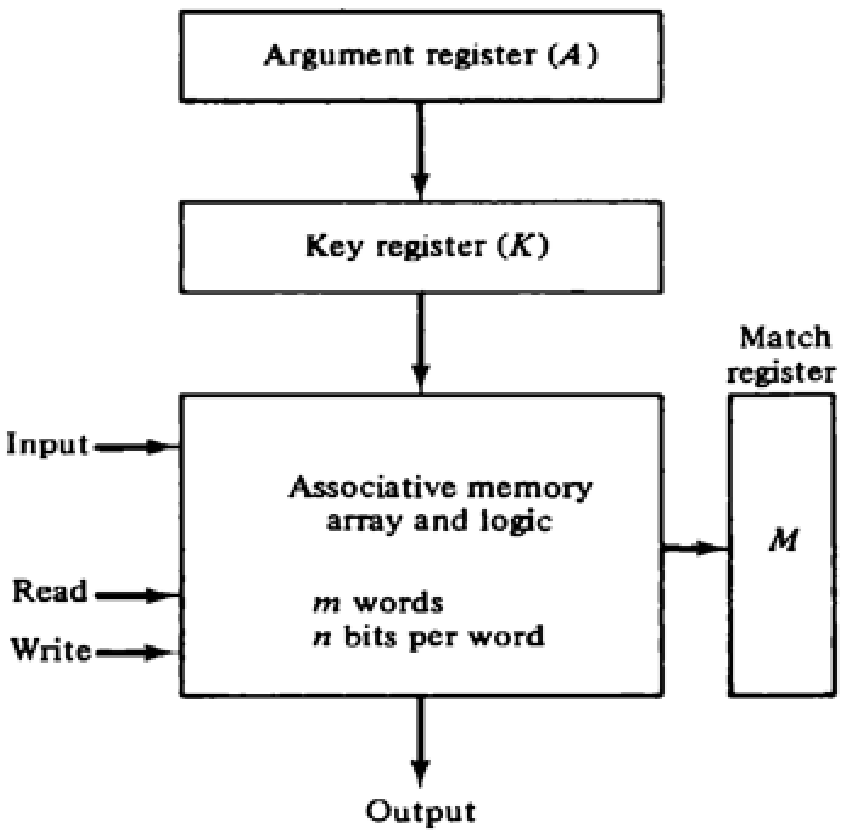


Figure 14.6 Block diagram of associative memory

It consists of a memory array and logic for m words with n bits per word. The argument register A and key register K each haven 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 b 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.

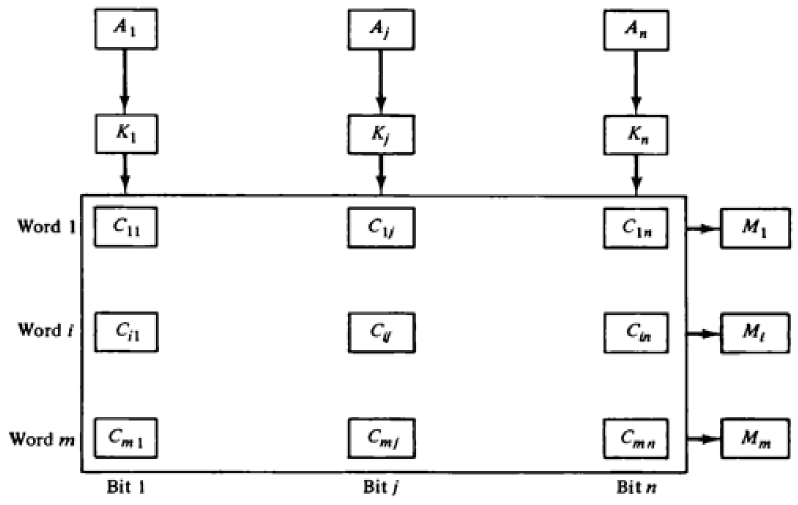


Figure 14.7 Associative memory of m word, n cells per word

The relation between the memory array and external registers in an associative memory is shown in Fig.14.7. The cells in the array are marked by the letter C with two subscripts. The first subscript gives the word number and second specifies the bit position in the word. Thus cell C_{ij} is the cell for bit j in word

i. A bit A_j in the argument register is compared with all the bits in column j of the array provided that k_j

=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 M_i in the match register is set to 1. If one or more unmasked bits of the argument and the word do not match, M_i is cleared to 0.

It consists of flip-flop storage element F_{ij} and the circuits for reading, writing, and matching the cell. The input bit is transferred into the storage cell during a write operation. The bit stored is read out during a read operation. The match logic compares the content of the storage cell with corresponding unmasked bit of the argument and provides an output for the decision logic that sets the bit in M_i.

Match Logic:

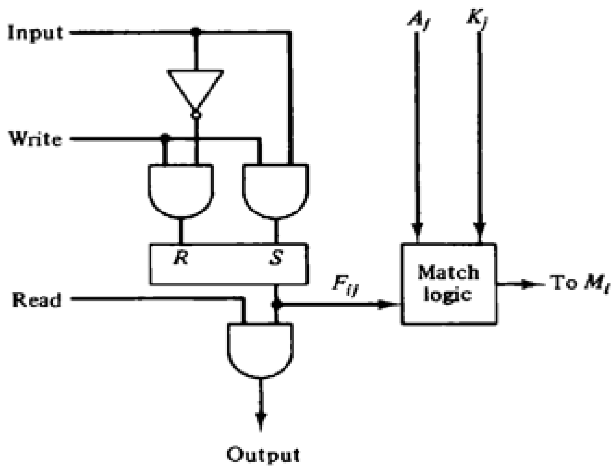


Figure 14.8 One cell of associative memory

The match logic for each word can be derived from the comparison algorithm for two binary numbers. First, neglect the key bits and compare the argument in A with the bits stored in the cells of the words.

Word i is equal to the argument in A if $A_j=F_{ij}$for j=1,2,…..,n. Two bits are equal if they are both 1 or both 0. The equality of two bits can be expressed logically by the Boolean function

$$x_j=A_jF_{ij} +A_j'F_{ij}'$$

where $x_j = 1$ if the pair of bits in position j are equal; otherwise, $x_j=0$. For a word i is equal to the argument in A we must have all x_j variables equal to 1. This is the condition for setting the corresponding match bit M_i to 1. The Boolean function for this condition is

$$M_i=x_1x_2x_3……x_n$$

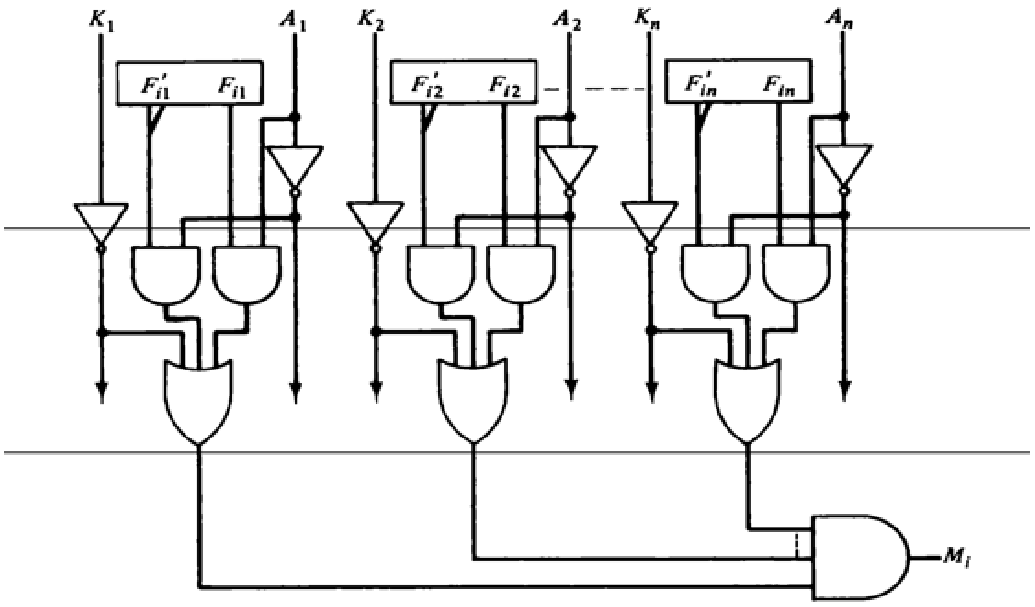


Figure 14.9 Mathc Logic for one word of associative memory

Each cell requires two AND gate and one OR gate. The inverters for A and K are needed once for each column and are used for all bits in the column. The output of all OR gates in the cells of the same word go to the input of a common AND gate to generate the match signal for M_i. M_i will be logic 1 if a match occurs and 0 if no match occurs.

Read and Write operation Read Operation

- If more than one word in memory matches the unmasked argument field , all the matched words will have 1's in the corresponding bit position of the match register.
- In read operation all matched words are reading sequence by applying are ad signal to each word line whose corresponding Mi bit is a logic 1
- In applications where no two identical items are stored in the memory, only one word may match, in which case we can use Mi output directly as a read signal for the corresponding word.

Write Operation:

Can take two different forms

1. Entire memory may be loaded with new information
2. Unwanted words to be deleted and new words to be inserted
3. Entire memory: writing can be done by addressing each location in sequence-This makes it random access memory for writing and content addressable memory for reading-number of lines needed for decoding is d Where m = 2 d , m is number of words.
4. Unwanted words to be deleted and new words to be inserted:
   - Tag register is used which has as many bits as there are words in memory.
   - For every active (valid) word in memory, the corresponding bit in tag register is set to 1.
   - When word is deleted the corresponding tag bit is reset to 0.
   - The word is stored in the memory by scanning the tag register until the first 0 bit is encountered. After storing the word the bit is set to 1.

14.5 CACHE MEMORY

- Effectiveness of cache mechanism is based on a property of computer programs called *“locality of reference”*
- The references to memory at any given time intervaltend to be confined within a localized areas
- Analysis of programs shows that most of their execution time is spent on routines in which instructions are executed repeatedly These instructions may be-loops, nested loops, or few procedures that call each other.
- Many instructions in localized areas of program are executed.
- Repeatedly during some time period and reminder of the program is accessed infrequently. This property is called “Locality of Reference”.

Locality of Reference:

Locality of reference is manifested in two ways:

1) Temporal-means that a recently executed instruction is likely to be executed again very soon.
2) The information which will be used in near future is likely to be in use already (e.g. reuse of information in loops)
3) Spatial-means that instructions in close proximity to a recently executed instruction are also likely to be executed soon.
4) If a word is accessed, adjacent (near) words are likely to be accessed soon (e.g. related data items (arrays) are usually stored together; instructions are executed sequentially)

5) If active segments of a program can be placed in a fast (cache) memory, then total execution time can be reduced significantly

6) Temporal Locality of Reference suggests whenever an information (instruction or data) is needed first , this item should be brought in to cache

7) Spatial aspect of Locality of Reference suggests that instead of bringing just one item from the main memory to the cache ,it is wise to bring several items that reside at adjacent addresses as well (i.e. a block of information).

Principles of Cache:

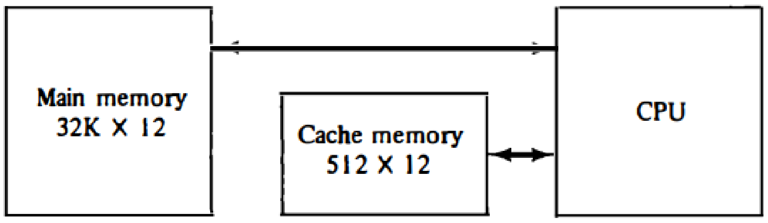


Figure 14.10 Example of Cache Memory

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, there is a duplicate copy in main memory. The CPU communicates with both memories. It first sends a 15 bit address to cahache. If there is a hit, the CPU accepts the 12 bit data from cache. If there is a miss, the CPU reads the word from main memory and the word is then transferred to cache.

- When a read request is received from CPU, contents of a block of memory words containing the location specified are transferred in to cache
- When the program references any of the locations in this block , the contents are read from the cache Number of blocks in cache is smaller than number of blocks in main memory
- Correspondence between main memory blocks and those in the cache is specified by a mapping function
- Assume cache is full and memory word not in cache is referenced
- Control hardware decides which block from cache is to be removed to create space for new block containing referenced word from memory
- Collection of rules form a king this decision is called *“Replacement algorithm”*.

Read/ Write Operations on cache:

- **Cache Hit Operation**
  - CPU issues Read/Write requests using addresses that refer to locations in main memory.

- Cache control circuitry determines whether requested word currently exists in cache.
- If it does, Read/Write operation is performed on the appropriate location in cache (Read/Write Hit)

Read/Write Operations on Cache incase of Hit

- In Read operation main memory is not involved.
- In Write operation two things can happen.

1) Cache and main memory locations are updated simultaneously *“Write Through”*.
2) Update only cache location and mark it as “Dirty or Modified Bit” and update main memory location at the time of cache block removal *(“Write Back” or “Copy Back”)*.

Read/Write operations on cache in case of Miss Read Operation:

- When addressed word is not in cache Read Miss occurs there are two ways this can be dealt with

1) Entire block of words that contain the requested word is copied from main memory to cache and the particular word requested is forwarded to CPU from the cache *(Load Through)*
2) The requested word from memory is sent to CPU first and then the cache is updated *(Early Restart)*

Write Operation

- If addressed word is not in cache Write Miss occurs
- If write through protocol is used information is directly written into main memory.
- In write back protocol, block containing the word is first brought in to cache, the desired word is then overwritten.

Mapping Functions:

Correspondence between main memory blocks and those in the cache is specified by a memory mapping function. There are three techniques in memory mapping.

- Direct Mapping
- Associative Mapping
- Set Associative Mapping

Direct mapping:

A particular block of main memory can be brought to a particular block of cache memory. So, it is not flexible.

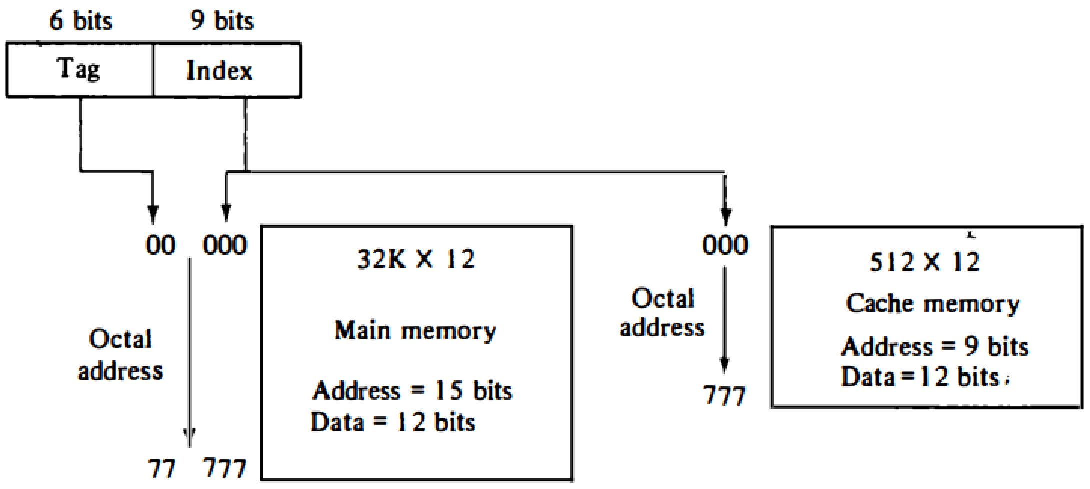


Figure 14.11 Addressing relationships between main and cache memories

In fig 14.11. The CPU address of 15 bits is divided into two fields. The nine least significant bits constitute the index field and remaining six bits form the tag field. The main memory needs an address that includes both the tag and the index bits. The number of bits in the index field is equal to the number of address bits required to access the cache memory.

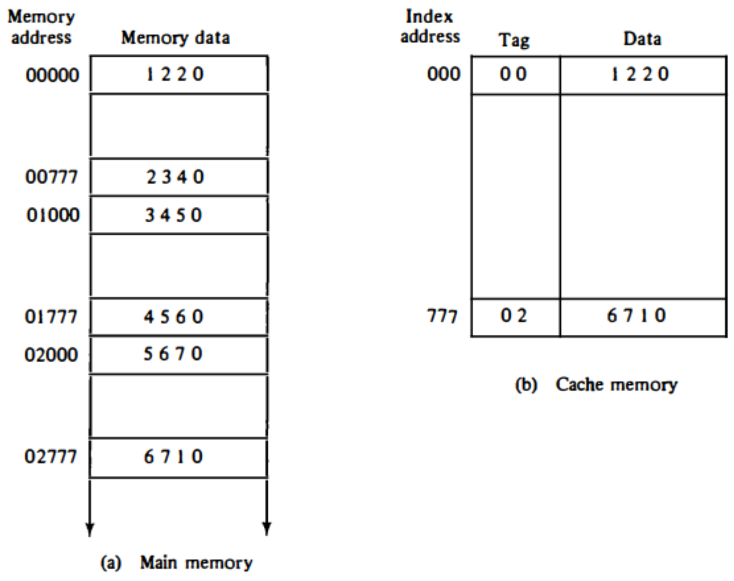


Figure 14.12 Direct mapping cache organization

The direct mapping cache organization uses the n- bit address to access the main memory and the k-bit index to access the cache. Each word in cache consists of the data word and 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 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. If there is no match, there is a miss and the required word is read from main memory.

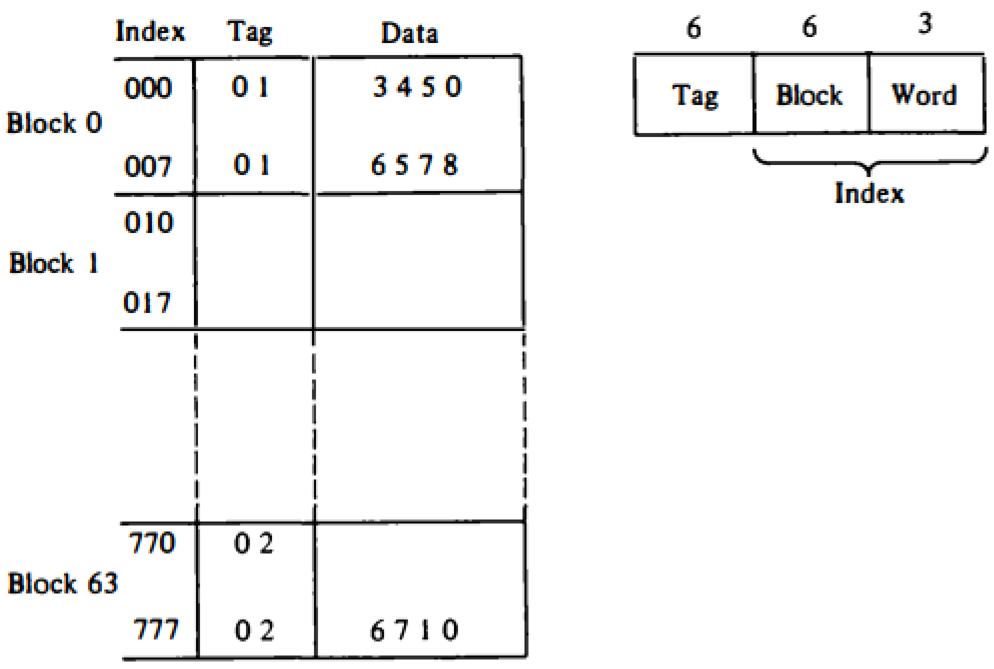


Figure 14.13 Direct mapping cache with block size of 8 words

In fig 14.13, The index field is now divided into two parts: Block field and The word field. In a 512 word cache there are 64 blocks of 8 words each, since 64 X 8 =514. The block number is specified with a 6 bit field and the word within the block is specified with a 3-bit field. The tag field stored within the the cache is common to all eight words of the same block.

Associative mapping:

In this mapping function, any block of Main memory can potentially reside in any cache block position. This is much more flexible mapping method.

The associative memory stores both address and content (data) of the memory word. This permits any location in cache to store any word from main memory. The diagram shows three words presently stored in the cache. The address value of 15 bits is shown as a five-digital number and its corresponding 14.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 address is found, the corresponding 14 Bit data is read and sent to the CPU. If no match occurs, the main memory is accessed for the word.

Set-Associative Mapping:

In this method, blocks of cache are grouped into sets, and the mapping allows a block of main memory to reside in any block of a specific set. From the flexibility point of view, it is in between to the other two methods.

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 14.14 Two way set associative mapping cache

The octal numbers listed in Fig.14.14 are with reference to the main memory contents. When the CPU generates a memory request, the index values of the address is used to access the cache. The tag field of the CPU address is then compared with both tags in the cache to determine if a match occurs. The comparison logic done by an associative search of the tags in the set similar to an associative memory search thus the name “Set Associative”.

Replacement Policies:

- When the cache is full and there is necessity to bring new data to cache, then a decision must be made as to which data from cache is to be removed.
- The guideline for taking a decision about which data is to be removed is called replacement policy Replacement policy depends on mapping.
- There is no specific policy in case of Direct mapping as we have no choice of block placement in cache Replacement Policies.

In case of associative mapping:

- A simple procedure is to replace cells of the cache in round robin order whenever a new word is requested from memory
- This constitutes a First-in First-out (FIFO) replacement policy

In case of set associative mapping:

- Random replacement
- First-in First-out (FIFO) (item chosen is the item that has been in the set longest)
- Least Recently Used (LRU) (item chosen is the item that has been least recently used by CPU).

14.6 VIRTUAL MEMORY:

Early day's memory was expensive–hence small. Programmers were using secondary storage for over laying Programmers were responsible for breaking programs in to overlays, decide where to keep in secondary memory, arranging for transfer of overlays between main memory and secondary memory In 1961 Manchester University proposed a method for performing overlay process automatically which has given rise to the concept of Virtual memory today

Virtual Memory-Background

Virtual memory is a separate concept to address space and memory locations. Programs reference instructions and data that is independent of available physical memory Addresses issued by processor for Instructions or Data are called Virtual or Logical addresses. Virtual addresses are translated in to physical addresses by a combination of Hardware and Software components

Address Space and Memory Space

Address used by a programmer is called virtual address and set of such addresses is called address space. Address in main memory is called a location or physical address and set of such locations is called the memory space

The Address Space is allowed to be larger than the memory space in computers with virtual memory

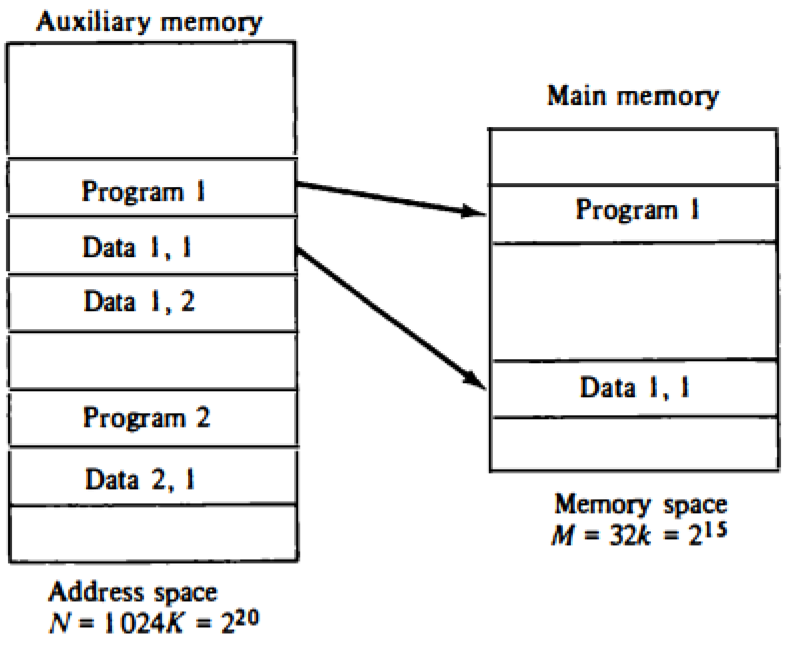


Figure 14.15 Relation between address and memory space in a virtual memory system

In a multi program computer system, programs and data are transferred to and from auxiliary memory and main memory based on demand simposed by the CPU. Suppose that program 1 is currently being executed in the CPU. Program1 and a portion of its associated data are moved from auxiliary memory into main memory as shown in fig. 14.15. Portions of programs and data need not be in contiguous locations in memory since information is being moved in out, and empty spaces may be available in scattered locations in memory.

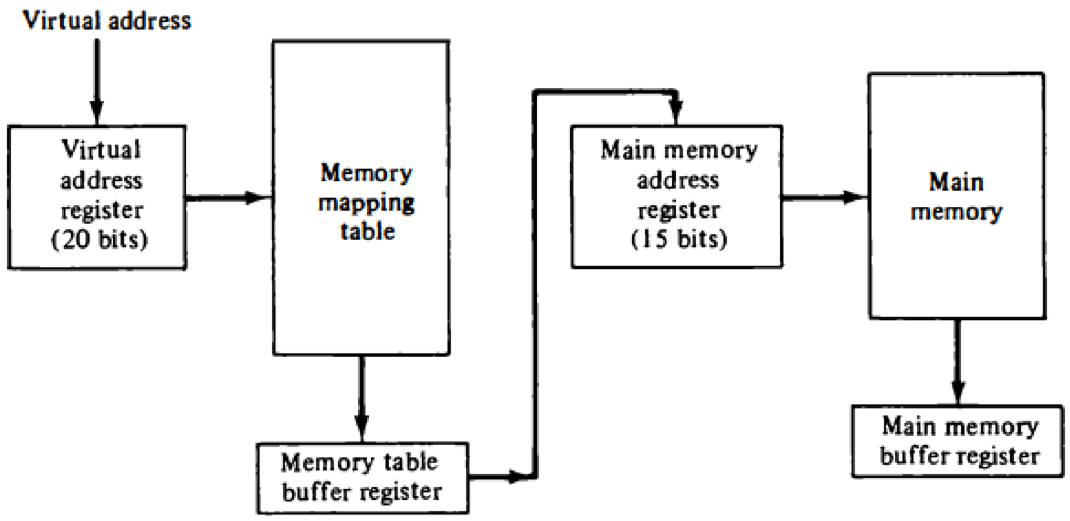

Virtual address
Virtual address register (20 bits)
Memory mapping table
Memory table buffer register
Main memory address register (15 bits)
Main memory
Main memory buffer register

Figure 14.16 Memory table for mapping a virtual address

In fig.14.16, To map a virtual address of 20 bits to a physical address of 15 bits. The mapping is a dynamic operation, which means that every address is translated immediately as a word is referenced by CPU. The mapping table maybe stored in a separate memory. In first case, an additional unit is required as well as one extra memory access time. In the second case, the table takes space from main memory and two accesses to memory are required with program running at half speed. A third alternative is to use an associative memory.

Address Mapping Using Pages:

The physical memory is broken down into groups of equal size called blocks, which may range from 64 to 4096 word each. The term page refers to groups of address space of the same size. Portions of programs are moved from auxiliary memory to main memory in records equal to the size of a page. The term “page frame” is sometimes used to denote a block.

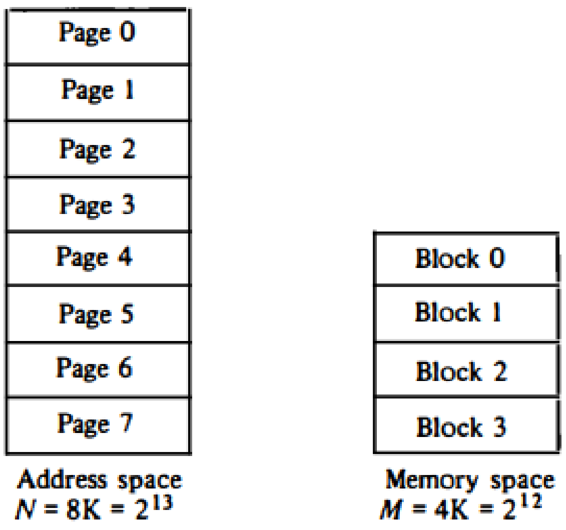


Figure 14.17 Address space and memory space split into groups of 1K words

In fig.14.17, a virtual address has 13 bits. Since each page consists of 1024 words, the high order three bits of virtual address will specify one of the eight pages and the low order 10 bits give the line address within the page.

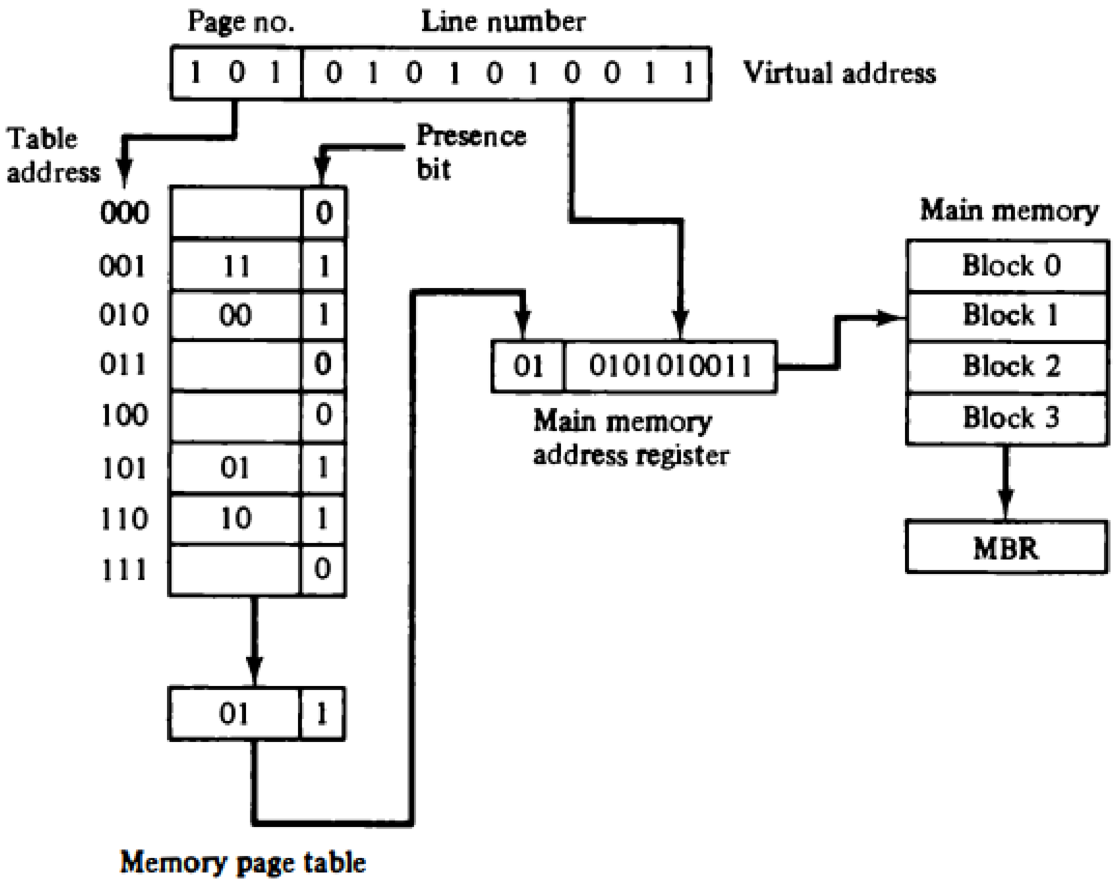


Figure 14.18 Memory table in a paged system

The organization of the memory mapping table in a paged system is shown in Fig.14.18. The memory page table consists of eight word, one for each page. The address in the page tabled notes the page number and the content of the word gives 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.

Associative Memory Page Table:

A random-access memory page table is inefficient with respect to storage utilization.

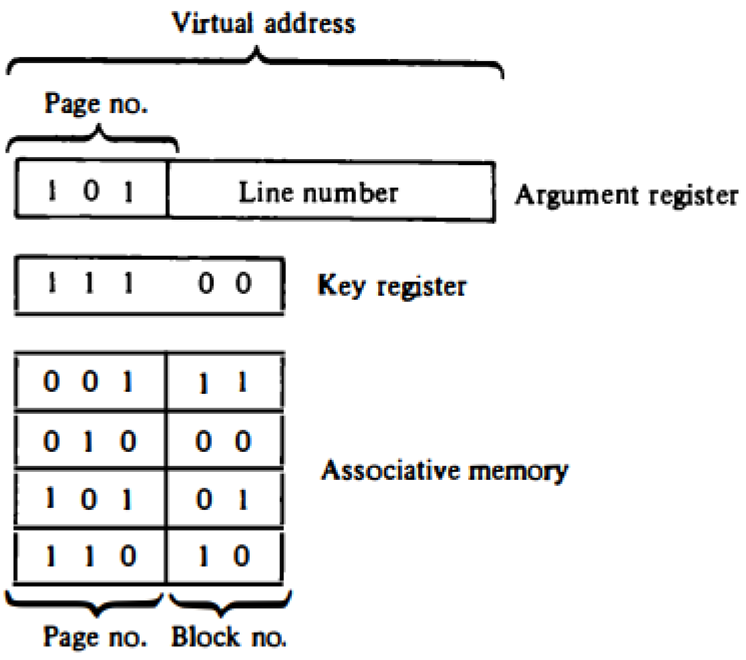


Figure 14.19 An Associative Memory Page Table

Replace the random access memory-page table with an associative memory of four words as shown in Fig14.19. Each entry in the associative memory array consists of two fields. The first three bits specify a field for storing the page number. The last two bits constitute a field for storing the block number. The virtual address is placed in the argument register.

14.6 SUMMARY:

The chapter on memory organization explores the structure and functionality of various types of computer memory, highlighting the memory hierarchy, which prioritizes speed and efficiency. It discusses main memory (RAM), essential for direct access by the CPU, and auxiliary memory (like hard drives), used for long-term data storage. The chapter delves into associative memory, also known as content-addressable memory, which allows for data retrieval based on content rather than address. Additionally, it covers cache memory, a smaller, faster type of volatile memory that stores copies of frequently accessed data to improve overall system performance.

14.7 TECHNICAL TERMS:

Memory Hierarchy, Main Memory, Auxiliary memory, Associative Memory, Cache Memory.

14.8 SELF ASSESSMENT QUESTIONS:

Essay questions:

1) Define auxiliary memory and its primary functions in a computer system?
2) Draw and explain the various types of secondary storage devices?

3) What is virtual memory? Why is it necessary to implement virtual memory? Explain the virtual memory address translation?

4) Discuss different cache mapping techniques such as direct mapping, associative mapping, and set-associative mapping?

Short Questions:

1) Explain the concept of memory hierarchy and its importance in computer architecture?

2) Compare and contrast the different levels of the memory hierarchy?

3) Discuss the differences between SRAM and DRAM?

4) Describe the purpose of cache memory?

14.9 FURTHER READINGS:

1) M.Morris Mano, "Computer System Architecture", 3rd Edition, Pearson Education (2008).
2) V. Rajaraman, T. Radha Krishnan, "Computer Organization and Architecture", PHI.
3) Behrooz Parhami, "Computer Architecture", Oxford (2007).
4) ISRD group, "Computer Organization", ace series, TMH (2007).
5) William Stallings, "Computer Organization and Architecture-Designing for Performance", Pearson Education (2005).

Dr. Vasantha Rudramalla