

ATME COLLEGE OF ENGINEERING

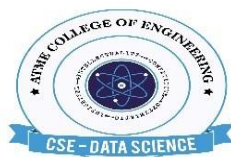
13th KM Stone, Bannur Road, Mysore - 560 028



A T M E

College of Engineering

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING (DATA SCIENCE)



(ACADEMIC YEAR 2024-25)

COURSE: Digital Design and Computer Organization

COURSE CODE: BCS302

SEMESTER: III-2022 CBCS Scheme

INSTITUTIONAL MISSION AND VISION

Objectives

- ☐ To provide quality education and groom top-notch professionals, entrepreneurs and leaders for different fields of engineering, technology and management.
- ☐ To open a Training-R & D-Design-Consultancy cell in each department, gradually introduce doctoral and postdoctoral programs, encourage basic & applied research in areas of social relevance, and develop the institute as a center of excellence.
- ☐ To develop academic, professional and financial alliances with the industry as well as the academia at national and transnational levels
- ☐ To develop academic, professional and financial alliances with the industry as well as the academia at national and transnational levels.
- ☐ To cultivate strong community relationships and involve the students and the staff in local community service.
- ☐ To constantly enhance the value of the educational inputs with the participation of students, faculty, parents and industry.

Vision

- ☐ Development of academically excellent, culturally vibrant, socially responsible and globally competent human resources.

Mission

- To keep pace with advancements in knowledge and make the students competitive and capable at the global level.
- To create an environment for the students to acquire the right physical, intellectual, emotional and moral foundations and shine as torch bearers of tomorrow's society.

- To strive to attain ever-higher benchmarks of educational excellence.

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING AND ENGINEERING
(DATA SCIENCE &ENGINEERING)

Vision of The Department

- To impart technical education in the field of data science of excellent quality with a high level of professional competence, social responsibility, and global awareness among the students

Mission

- To impart technical education that is up to date, relevant and makes students competitive and employable at global level
- To provide technical education with a high sense of discipline, social relevance in an intellectually, ethically and socially challenging environment for better tomorrow
- Educate to the global standards with a benchmark of excellence and to kindle the spirit of innovation.

Program Outcomes(PO)

- **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
- **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

- **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
- **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
- **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
- **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice
- **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
- **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
- **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
- **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

- **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
- **Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Outcomes (PSOs)

- PSO1: Develop relevant programming skills to become a successful data scientist
- PSO2: Apply data science concepts and algorithms to solve real world problems of the society
- PSO3: Apply data science techniques in the various domains like agriculture, education healthcare for better society

Program Educational Objectives (PEOs):

PEO1: Develop cutting-edge skills in data science and its related technologies, such as machine learning, predictive analytic, and data engineering.

PEO2: Design and develop data-driven solutions to real-world problems in a business, research, or social environment.

PEO3: Apply data engineering and data visualization techniques to discover, investigate, and interpret data.

PEO4: Demonstrate ethical and responsible data practices in problem solving

PEO5: Integrate fields within computer science, optimization, and statistics to develop better solutions

Course Code	Course Title	Core / Elective	Prerequisite	Contact Hours			Total Hrs/ Sessions
				L	T	P	
BCS302	Digital Design & Computer Organization	Core	Basics of Algebra	3	0	2	40T+20P
Course Objectives	1. To demonstrate the functionalities of binary logic system 2. To explain the working of combinational and sequential logic system 3. To realize the basic structure of computer system 4. To illustrate the working of I/O operations and processing unit						

Topics Covered as per Syllabus

Module-1

Introduction to Digital Design: Binary Logic, Basic Theorems and Properties of Boolean Algebra, Boolean Functions, Digital Logic Gates, Introduction, The Map Method, Four-Variable Map, Don't-Care Conditions, NAND and NOR Implementation, Other Hardware Description Language – Verilog Model of a simple circuit.

Module-2

Combinational Logic: Introduction, Combinational Circuits, Design Procedure, Binary Adder-Subtractor, Decoders, Encoders, Multiplexers. HDL Models of Combinational Circuits – Adder, Multiplexer, Encoder. **Sequential Logic:** Introduction, Sequential Circuits, Storage Elements: Latches, Flip-Flops.

Module-3

Basic Structure of Computers: Functional Units, Basic Operational Concepts, Bus structure, Performance – Processor Clock, Basic Performance Equation, Clock Rate, Performance Measurement. **Machine Instructions and Programs:** Memory Location and Addresses, Memory Operations, Instruction and Instruction sequencing, Addressing Modes.

Module-4

Input/output Organization: Accessing I/O Devices, Interrupts – Interrupt Hardware, Enabling and Disabling Interrupts, Handling Multiple Devices, Direct Memory Access: Bus Arbitration, Speed, size and Cost of memory systems. Cache Memories – Mapping Functions.

Module-5

Basic Processing Unit: Some Fundamental Concepts: Register Transfers, Performing ALU operations, fetching a word from Memory, Storing a word in memory. Execution of a Complete Instruction. Pipelining: Basic concepts, Role of Cache memory, Pipeline Performance.

Laboratory Component:

1. Given a 4-variable logic expression, simplify it using appropriate technique and simulate the same using basic gates.
2. Design a 4-bit full adder and subtractor and simulate the same using basic gates.
3. Design Verilog HDL to implement simple circuits using structural, Data flow and Behavioural model.

4. Design Verilog HDL to implement Binary Adder-Subtractor – Half and Full Adder, Half and Full Subtractor.
5. Design Verilog HDL to implement Decimal adder.
6. Design Verilog program to implement Different types of multiplexers like 2:1, 4:1 and 8:1.
7. Design Verilog program to implement types of De-Multiplexer.
8. Design Verilog program for implementing various types of Flip-Flops such as SR, JK and D.

List of Textbooks

1. M. Morris Mano & Michael D. Ciletti, Digital Design with an Introduction to Verilog Design, 5e, Pearson Education.
2. Carl Hamacher, Zvonko Vranesic, Safwat Zaky, Computer Organization, 5th Edition, Tata McGraw Hill.

Course Outcomes	<ol style="list-style-type: none"> 1. Apply the K–Map techniques to simplify various Boolean expressions. 2. Design different types of combinational and sequential circuits along with Verilog programs. 3. Describe the fundamentals of machine instructions, addressing modes and Processor performance. 4. Explain the approaches involved in achieving communication between processor and I/O devices. 5. Analyze internal Organization of Memory and Impact of cache/Pipelining on Processor Performance.
------------------------	---

MODULE-1

Introduction to Digital Design

1.1 Binary Logic

Binary logic deals with variables that take on two discrete values and with operations that assume logical meaning. The two values the variables assume may be called by different names (true and false, yes and no, etc.), but for our purpose, it is convenient to think in terms of bits and assign the values 1 and 0. The binary logic is equivalent to an algebra called Boolean algebra.

Binary logic consists of binary variables and a set of logical operations. The variables are designated by letters of the alphabet, such as A, B, C, x, y, z, etc., with each variable having two and only two distinct possible values: 1 and 0. There are three basic logical operations: AND, OR, and NOT. Each operation produces a binary result, denoted by z.

1. AND: This operation is represented by a dot or by the absence of an operator. For example, $x \cdot y = z$ or $xy = z$ is read “x AND y is equal to z.” The logical operation AND is interpreted to mean that $z = 1$ if and only if $x = 1$ and $y = 1$; otherwise, $z = 0$. (Remember that x, y, and z are binary variables and can be equal either to 1 or 0, and nothing else.) The result of the operation $x \cdot y$ is z.

2. OR: This operation is represented by a plus sign. For example, $x + y = z$ is read “x OR y is equal to z,” meaning that $z = 1$ if $x = 1$ or if $y = 1$ or if both $x = 1$ and $y = 1$. If both $x = 0$ and $y = 0$, then $z = 0$.

3. NOT: This operation is represented by a prime (sometimes by an overbar). For example, $x' = z$ (or $x = z$) is read “not x is equal to z,” meaning that z is what x is not. In other words, if $x = 1$, then $z = 0$, but if $x = 0$, then $z = 1$. The NOT operation is also referred to as the complement operation, since it changes a 1 to 0 and a 0 to 1, i.e., the result of complementing 1 is 0, and vice versa.

Logic gates are elementary building block of a digital circuit. Most logic gates have two inputs and only one output. The basic logic gates are AND, OR and NOT. Binary logic deals with true and false.

1.2 Basic Theorems and Properties of Boolean Algebra

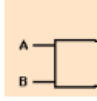
De Morgan's First Theorem: The complement of a sum equals the product of the complements.

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

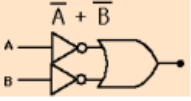
A	B	A+B	$\overline{A+B}$	\bar{A}	\bar{B}	$\bar{A} \cdot \bar{B}$
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

De Morgan's Second Theorem: The complement of a sum equals the product of the complements. $\overline{A+B} = \bar{A} \cdot \bar{B}$

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



NAND Gate



Bubbled OR Gate

Duality Theorem: Starting with a Boolean relation, can derive another Boolean relation by

1. Changing each OR sign to an AND sign
2. Changing each AND sign to an OR sign
3. Complementing any 0 or 1 appearing in the expression.

Example: 1. We say that, $A+0 = A$; the dual is, $A \cdot 1 = A$

2. Consider, $A(B+C) = AB + AC$

By changing the OR and AND operation, we get the dual relation:

$$A + BC = (A+B)(A+C)$$

Laws of Boolean Algebra

The following laws are of immense use in the simplification of Boolean expressions. Note that, if A is a variable, then either $A = 0$ or $A = 1$. Also, when $A = 0$, $A \neq 1$; and when $A = 1$, $A \neq 0$.

- 1) Commutative Law: $A + B = B + A$ and $A \cdot B = B \cdot A$
- 2) Associative Law: $A + (B + C) = (A + B) + C$ and $A \cdot (BC) = (AB) \cdot C$
- 3) Distributive Law: $A(B + C) = AB + AC$
- 4) In relation to OR operation, the following laws hold good:
 - $A + 0 = A$
 - $A + A = A$
 - $A + 1 = 1$ and $A + A' = 1$
- 5) In relation to AND operation, the following laws hold good:
 - $A \cdot 1 = A$
 - $A \cdot A = A$
 - $A \cdot 0 = 0$
 - $A \cdot A' = 0$ and $A'' = A$
- 6) Some more useful Boolean relations:
 - $A + AB = A$
 - $A + A'B = A + B$
 - $A(A + B) = A$
 - $A(A' + B) = AB$
 - $A + (B \cdot C) = (A + B)(A + C)$

1.3 Boolean Functions

Boolean algebra is an algebra that deals with binary variables and logic operations. A Boolean function described by an algebraic expression consists of binary variables, the constants 0 and 1, and the logic operation symbols. For a given value of the binary variables, the function can be equal to either 1 or 0.

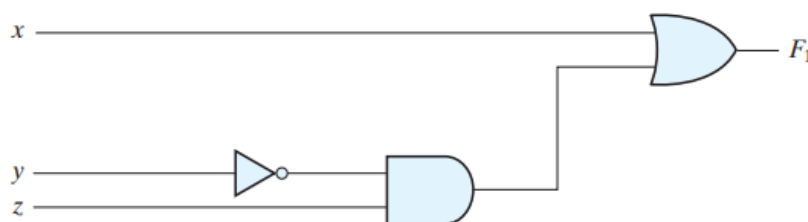
As an example, consider the Boolean function $F1 = x + y'z$.

The function $F1$ is equal to 1 if x is equal to 1 or if both y' and z are equal to 1. $F1$ is equal to 0 otherwise. The complement operation dictates that when $y' = 1$, $y = 0$. Therefore, $F1 = 1$ if $x = 1$ or if $y = 0$ and $z = 1$. A Boolean function expresses the logical relationship between binary variables and is evaluated by determining the binary value of the expression for all possible values of the variables.

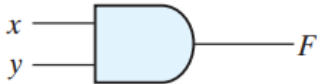

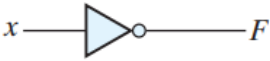


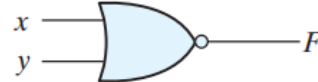
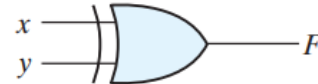

A Boolean function can be represented in a truth table. The number of rows in the truth table is 2^n , where n is the number of variables in the function. The binary combinations for the truth table are obtained from the binary numbers by counting from 0 through $2^n - 1$. Table below shows the truth table for the function $F1$. There are eight possible binary combinations for assigning bits to the three variables x , y , and z . The column labelled $F1$ contains either 0 or 1 for each of these combinations. The table shows that the function is equal to 1 when $x = 1$ or when $yz = 01$ and is equal to 0 otherwise.

x	y	z	F_1
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 Boolean function can be transformed from an algebraic expression into a circuit diagram composed of logic gates connected in a particular structure. The logic-circuit diagram (also called a schematic) for $F1$ is shown in Fig. There is an inverter for input y to generate its complement. There is an AND gate for the term $y'z$ and an OR gate that combines x with $y'z$. In logic-circuit diagrams, the variables of the function are taken as the inputs of the circuit and the binary variable $F1$ is taken as the output of the circuit. The schematic expresses the relationship between the output of the circuit and its inputs.

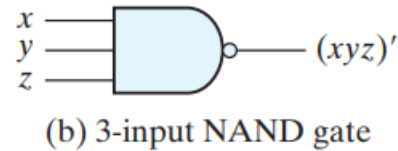
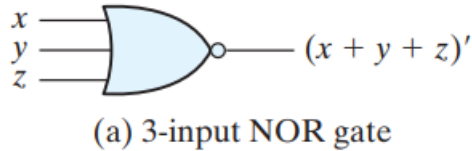


1.4 Digital Logic Gates

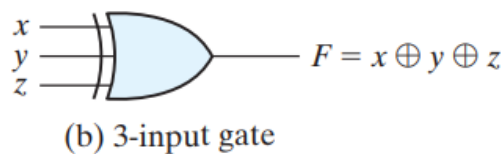
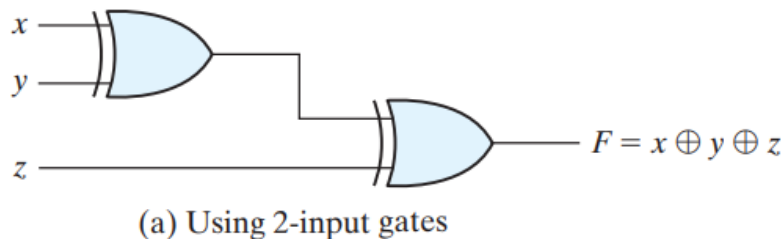
Name	Graphic symbol	Algebraic function	Truth table															
AND		$F = x \cdot y$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	x	y	F	0	0	0	0	1	0	1	0	0	1	1	1
x	y	F																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR		$F = x + y$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	x	y	F	0	0	0	0	1	1	1	0	1	1	1	1
x	y	F																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
Inverter		$F = x'$	<table><tr><th>x</th><th>F</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	x	F	0	1	1	0									
x	F																	
0	1																	
1	0																	
Buffer		$F = x$	<table><tr><th>x</th><th>F</th></tr><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td></tr></table>	x	F	0	0	1	1									
x	F																	
0	0																	
1	1																	
NAND		$F = (xy)'$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	x	y	F	0	0	1	0	1	1	1	0	1	1	1	0
x	y	F																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
NOR		$F = (x + y)'$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	x	y	F	0	0	1	0	1	0	1	0	0	1	1	0
x	y	F																
0	0	1																
0	1	0																
1	0	0																
1	1	0																
Exclusive-OR (XOR)		$F = xy' + x'y$ $= x \oplus y$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	x	y	F	0	0	0	0	1	1	1	0	1	1	1	0
x	y	F																
0	0	0																
0	1	1																
1	0	1																
1	1	0																
Exclusive-NOR or equivalence		$F = xy + x'y'$ $= (x \oplus y)'$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	x	y	F	0	0	1	0	1	0	1	0	0	1	1	1
x	y	F																
0	0	1																
0	1	0																
1	0	0																
1	1	1																

Extension to Multiple Inputs

The graphic symbols for the three-input gates are shown in Fig. below. In writing cascaded NOR and NAND operations, one must use the correct parentheses to signify the proper sequence of the gates.



The construction of a three-input exclusive-OR function is shown in Fig. below. This function is normally implemented by cascading two-input gates, as shown in (a). Graphically, it can be represented with a single three-input gate, as shown in (b). The truth table in (c) clearly indicates that the output F is equal to 1 if only one input is equal to 1 or if all three inputs are equal to 1 (i.e., when the total number of 1's in the input variables is odd).

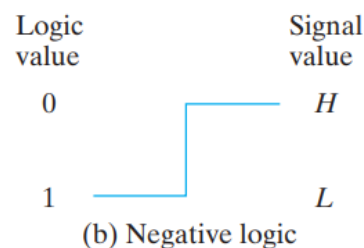
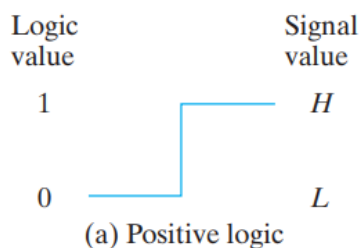


x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

(c) Truth table

Positive and Negative Logic

The binary signal at the inputs and outputs of any gate has one of two values, except during transition. One signal value represents logic 1 and the other logic 0. Since two signal values are assigned to two logic values, there exist two different assignments of signal level to logic value, as shown in Fig. below. The higher signal level is designated by H and the lower signal level by L . Choosing the high-level H to represent logic 1 defines a positive logic system. Choosing the low-level L to represent logic 1 defines a negative logic system. The terms positive and negative are somewhat misleading since both signals may be positive or both may be negative. It is not the actual values of the signals that determine the type of logic, but rather the assignment of logic values to the relative amplitudes of the two signal levels.



1.5 Introduction

Gate-level minimization is the design task of finding an optimal gate-level implementation of the Boolean functions describing a digital circuit. This task is well understood but is difficult to execute by manual methods when the logic has more than a few inputs. Fortunately, computer-based logic synthesis tools can minimize a large set of Boolean equations efficiently and quickly. Nevertheless, it is important that a designer understand the underlying mathematical description and solution of the problem.

Minimum Forms of Switching Functions

A minimum sum-of-products expression

- A minimum sum-of-products expression for a function is defined as a sum of product terms which (a) has a minimum number of terms and (b) of all those expressions which have the same minimum number of terms, has a minimum number of literals.
- The minimum sum of products corresponds directly to a minimum two-level gate circuit which has (a) a minimum number of gates and (b) a minimum number of gate inputs.

Given a minterm expansion, the minimum sum-of-products form can often be obtained by the following procedure:

1. Combine terms by using the uniting theorem $XY' + XY = X$. Do this repeatedly to eliminate as many literals as possible. A given term may be used more than once because $X + X = X$.
2. Eliminate redundant terms by using the consensus theorem or other theorems. Unfortunately, the result of this procedure may depend on the order in which terms are combined or eliminated so that the final expression obtained is not necessarily minimum.

A minimum products-of-sum expression

- A minimum product-of-sums expression for a function is defined as a product of sum terms which (a) has a minimum number of terms, and (b) of all those expressions which have the same number of terms, has a minimum number of literals.

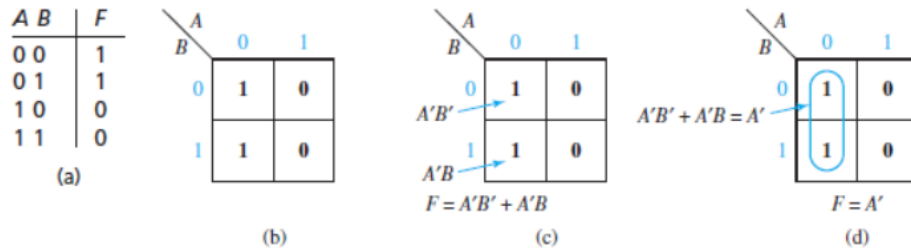
Unlike the maxterm expansion, the minimum product-of-sums form of a function is not necessarily unique. Given a maxterm expansion, the minimum product of sums can often be obtained by a procedure similar to that used in the minimum sum of-products case, except that the uniting theorem $(X + Y)(X + Y') = X$ is used to combine terms.

1.6 The Map Method

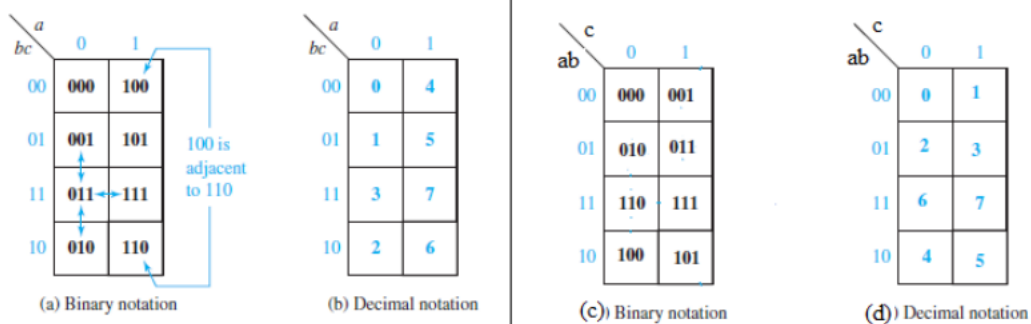
Karnaugh Map-A drawing that shows all the fundamental products and the corresponding output values of a truth table.

Just like a truth table, the Karnaugh map of a function specifies the value of the function for every combination of values of the independent variables.

The values of one variable are listed across the top of the map, and the values of the other variable are listed on the left side. Each square of the map corresponds to a pair value for A and B as indicated. The following Figure shows the truth table for a function F and the corresponding Karnaugh map:

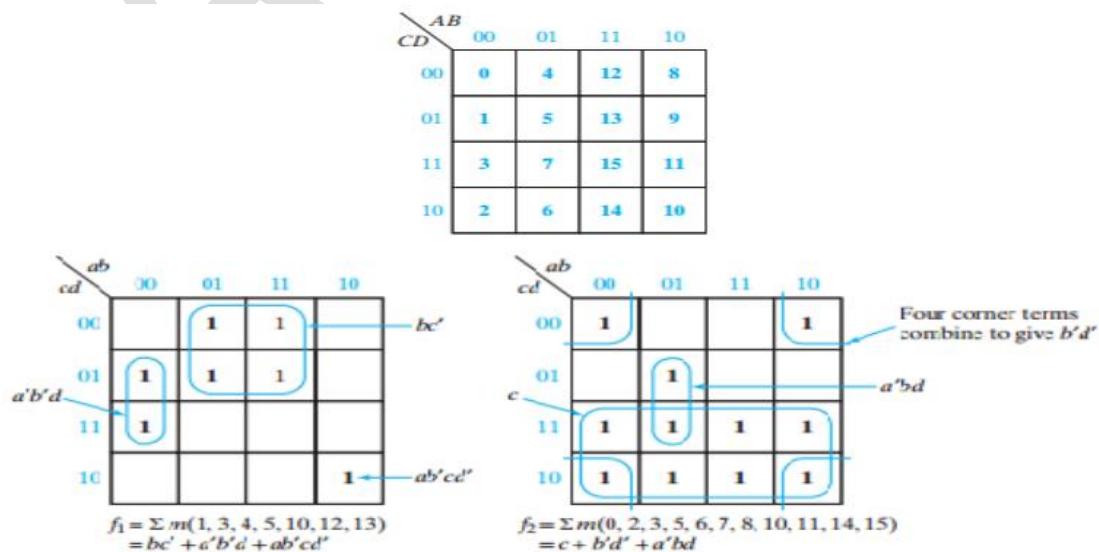


The following Figure shows a three-variable truth table and the corresponding Karnaugh map: The value of one variable (A) is listed across the top of the map, and the values of the other two variables (B, C) are listed along the side of the map. The rows are labelled in the sequence 00, 01, 11, 10 so that values in adjacent rows differ in only one variable.



1.7 Four-Variable Map

A four -variable Karnaugh map is shown below Each minterm is located adjacent to the four terms with which it can combine.



Pairs: The following K-map contains a pair of 1s that are horizontally adjacent. Two adjacent 1s, such as these are called a pair. A pair eliminates one variable and its complement.

Y	$\bar{A}\bar{B}$	$\bar{A}B$	AB	$A\bar{B}$
$\bar{C}\bar{D}$	0	0	0	0
$\bar{C}D$	0	0	0	0
CD	0	0	1	0
$C\bar{D}$	0	0	1	0

The sum-of-product equation is:

$$Y = ABCD + ABCD' = ABC(D + D') = ABC$$

Quad: A quad is a group of four 1s that are horizontally or vertically adjacent. A quad eliminates two variables and their complements.

Y	$\bar{A}\bar{B}$	$\bar{A}B$	AB	$A\bar{B}$
$\bar{C}\bar{D}$	0	0	1	0
$\bar{C}D$	0	0	1	0
CD	0	0	1	0
$C\bar{D}$	0	0	1	0

The sum-of-product equation is:

$$Y = ABC' + ABC = AB(C + C') = AB$$

The Octet: The octet is a group of eight 1s, as shown in the following Fig. An octet eliminates three variables and their complements.

Y	$\bar{A}\bar{B}$	$\bar{A}B$	AB	$A\bar{B}$
$\bar{C}\bar{D}$	0	0	1	1
$\bar{C}D$	0	0	1	1
CD	0	0	1	1
$C\bar{D}$	0	0	1	1

The sum-of-product equation is:

$$Y = AB + AB' = A(B + B') = A$$

Prime Implicants

In choosing adjacent squares in a map, we must ensure that.

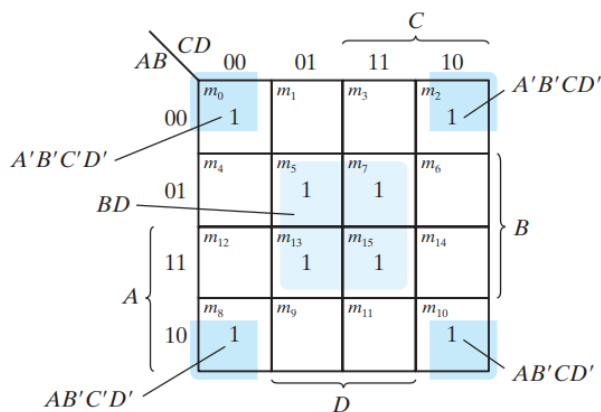
(1) all the minterms of the function are covered when we combine the squares,

- (2) the number of terms in the expression is minimized, and
- (3) there are no redundant terms (i.e., minterms already covered by other terms).

Sometimes there may be two or more expressions that satisfy the simplification criteria. The procedure for combining squares in the map may be made more systematic if we understand the meaning of two special types of terms. A prime implicant is a product term obtained by combining the maximum possible number of adjacent squares in the map. If a minterm in a square is covered by only one prime implicant, that prime implicant is said to be essential.

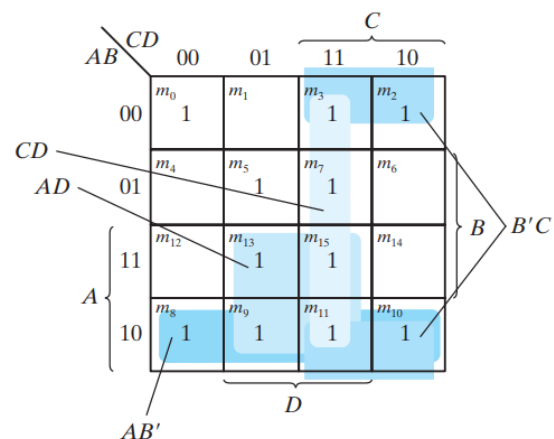
The prime implicants of a function can be obtained from the map by combining all possible maximum numbers of squares. This means that a single 1 on a map represents a prime implicant if it is not adjacent to any other 1's. Two adjacent 1's form a prime implicant, provided that they are not within a group of four adjacent squares. Four adjacent 1's form a prime implicant if they are not within a group of eight adjacent squares, and so on. The essential prime implicants are found by looking at each square marked with a 1 and checking the number of prime implicants that cover it. The prime implicant is essential if it is the only prime implicant that covers the minterm.

Consider the following four-variable Boolean function: $F(A, B, C, D) = \sum(0, 2, 3, 5, 7, 8, 9, 10, 11, 13, 15)$



Note: $A'B'C'D' + A'B'CD' = A'B'D'$
 $AB'C'D' + AB'CD' = AB'D'$
 $A'B'D' + AB'D' = B'D'$

(a) Essential prime implicants BD and B'D'

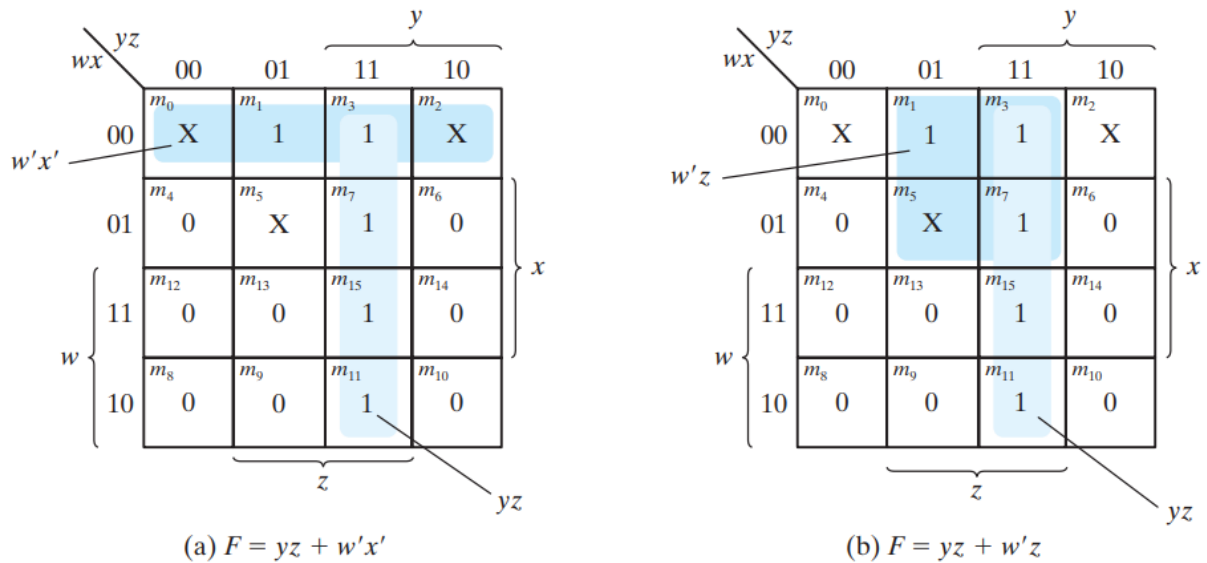


(b) Prime implicants CD, B'C, AD, and AB'

1.8 Don't-Care Conditions

A don't-care minterm is a combination of variables whose logical value is not specified. Such a minterm cannot be marked with a 1 in the map, because it would require that the function always be a 1 for such a combination. Likewise, putting a 0 on the square requires the function to be 0. To distinguish the don't-care condition from 1's and 0's, an X is used. Thus, an X inside a square in the map indicates that we don't care whether the value of 0 or 1 is assigned to F for the minterm.

Eg: Simplify the Boolean function $F(w, x, y, z) = \sum(1, 3, 7, 11, 15)$ which has the don't-care conditions $d(w, x, y, z) = \sum(0, 2, 5)$



In Fig. (a), don't-care minterms 0 and 2 are included with the 1's, resulting in the simplified function

$$F = yz + w'x'$$

In Fig. (b), don't-care minterm 5 is included with the 1's, and the simplified function is now

$$F = yz + w'z$$

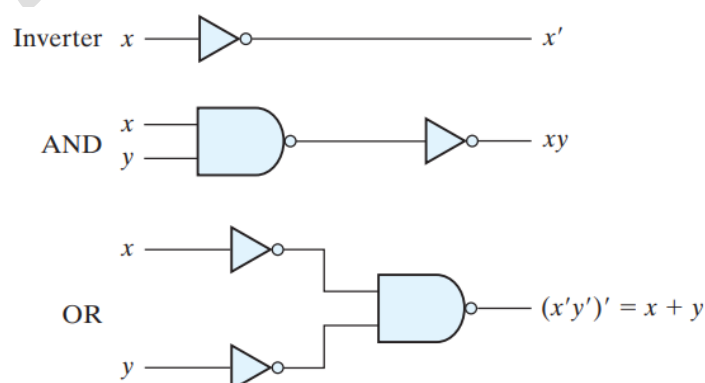
Either one of the preceding two expressions satisfies the conditions stated for this example.

1.9 NAND and NOR Implementation

Digital circuits are frequently constructed with NAND or NOR gates rather than with AND and OR gates. NAND and NOR gates are easier to fabricate with electronic components and are the basic gates used in all IC digital logic families. Because of the prominence of NAND and NOR gates in the design of digital circuits, rules and procedures have been developed for the conversion from Boolean functions given in terms of AND, OR, and NOT into equivalent NAND and NOR logic diagrams.

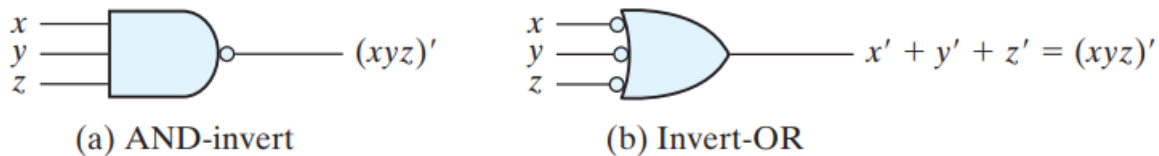
NAND Circuits

The NAND gate is said to be a universal gate because any logic circuit can be implemented with it. To show that any Boolean function can be implemented with NAND gates, we need only show that the logical operations of AND, OR, and complement can be obtained with NAND gates alone. This is indeed shown in Fig below.



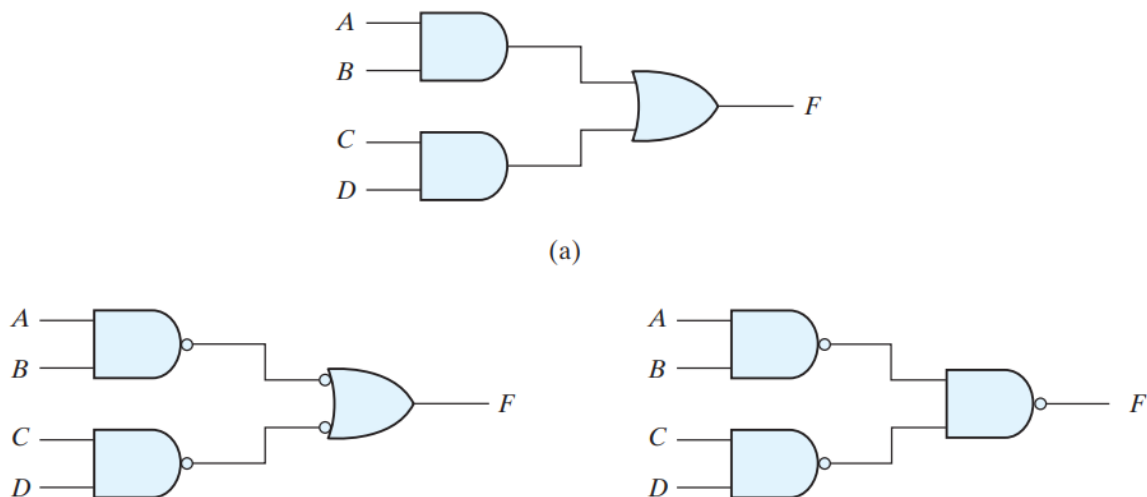
A convenient way to implement a Boolean function with NAND gates is to obtain the simplified Boolean function in terms of Boolean operators and then convert the function to NAND logic. The conversion of an algebraic expression from AND, OR, and complement to NAND can be done by simple circuit manipulation techniques that change AND-OR diagrams to NAND diagrams.

To facilitate the conversion to NAND logic, it is convenient to define an alternative graphic symbol for the gate. Two equivalent graphic symbols for the NAND gate are shown in Fig.



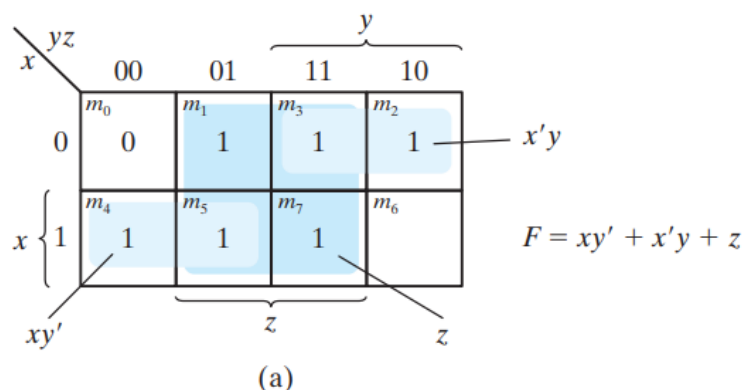
Two-Level Implementation

The implementation of Boolean functions with NAND gates requires that the functions be in sum-of-products form. To see the relationship between a sum-of-products expression and its equivalent NAND implementation, consider the logic diagrams drawn in Fig. All three diagrams are equivalent and implement the function $F = AB + CD$

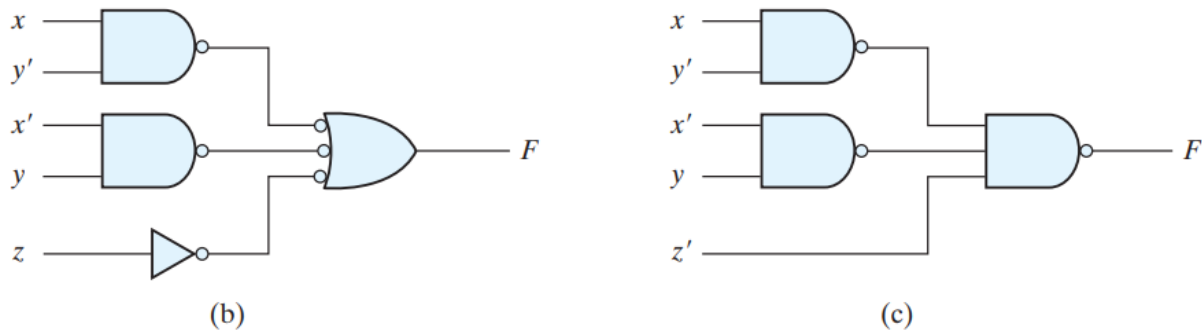


Eg: Implement the following Boolean function with NAND gates: $F(x, y, z) = (1, 2, 3, 4, 5, 7)$

The first step is to simplify the function into sum-of-products form. This is done by means of the map of Fig. below, from which the simplified function is obtained: $F = xy' + x'y + z$

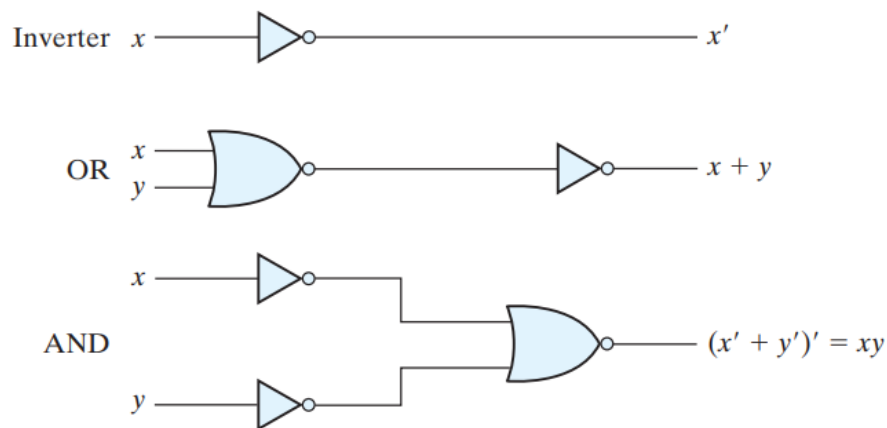


The two-level NAND implementation is shown in Fig. (b) in mixed notation. Note that input z must have a one-input NAND gate (an inverter) to compensate for the bubble in the second-level gate. An alternative way of drawing the logic diagram is given in Fig. (c). Here, all the NAND gates are drawn with the same graphic symbol. The inverter with input z has been removed, but the input variable is complemented and denoted by z' .

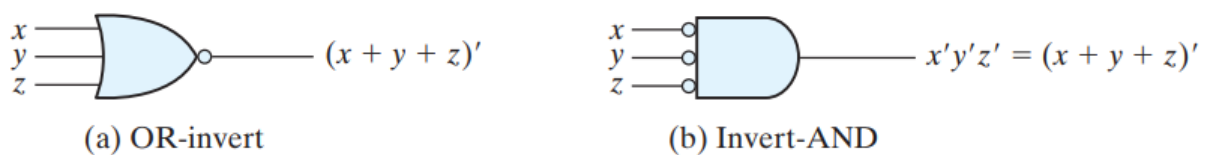


NOR Implementation

The NOR operation is the dual of the NAND operation. Therefore, all procedures and rules for NOR logic are the duals of the corresponding procedures and rules developed for NAND logic. The NOR gate is another universal gate that can be used to implement any Boolean function. The implementation of the complement, OR, and AND operations with NOR gates is shown in Fig below.



The two graphic symbols for the mixed notation are shown in below Fig. The OR-invert symbol defines the NOR operation as an OR followed by a complement. The invert-AND symbol complements each input and then performs an AND operation. The two symbols designate the same NOR operation and are logically identical because of DeMorgan's theorem.



1.10 Other Hardware Description Language – Verilog Model of a simple circuit.

Manual methods for designing logic circuits are feasible only when the circuit is small. For anything else (i.e., a practical circuit), designers use computer-based design tools. Coupled with the correct-by-construction methodology, computer-based design tools leverage the creativity and the effort of a designer and reduce the risk of producing a flawed design. Prototype integrated circuits are too expensive and time consuming to build, so all modern design tools rely on a hardware description language to describe, design, and test a circuit in software before it is ever manufactured.

A hardware description language (HDL) is a computer-based language that describes the hardware of digital systems in a textual form. It resembles an ordinary computer programming language, such as C, but is specifically oriented to describing hardware structures and the behavior of logic circuits. It can be used to represent logic diagrams, truth tables, Boolean expressions, and complex abstractions of the behavior of a digital system. One way to view an HDL is to observe that it describes a relationship between signals that are the inputs to a circuit and the signals that are the outputs of the circuit. For example, an HDL description of an AND gate describes how the logic value of the gate's output is determined by the logic values of its inputs.

As a documentation language, an HDL is used to represent and document digital systems in a form that can be read by both humans and computers and is suitable as an exchange language between designers. The language content can be stored, retrieved, edited, and transmitted easily and processed by computer software in an efficient manner.

HDLs are used in several major steps in the design flow of an integrated circuit: design entry, functional simulation or verification, logic synthesis, timing verification, and fault simulation.

Companies that design integrated circuits use proprietary and public HDLs. In the public domain, there are two standard HDLs that are supported by the IEEE: VHDL and Verilog.

1. VHDL is a Department of Defense–mandated language. (The V in VHDL stands for the first letter in VHSIC, an acronym for very high-speed integrated circuit.)
2. Verilog began as a proprietary HDL of Cadence Design Systems, but Cadence transferred control of Verilog to a consortium of companies and universities known as Open Verilog International (OVI) as a step leading to its adoption as an IEEE standard.

VHDL is more difficult to learn than Verilog.

Module Declaration

The language reference manual for the Verilog HDL presents a syntax that describes precisely the constructs that can be used in the language. A Verilog model is composed of text using keywords, of which there are about 100.

Keywords are predefined lowercase identifiers that define the language constructs. Examples of keywords are module, endmodule, input, output, wire, and, or, and not.

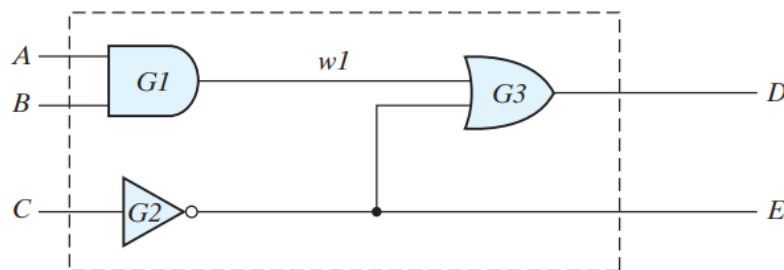
Any text between two forward slashes (//) and the end of the line is interpreted as a comment and will have no effect on a simulation using the model. Multiline comments begin with / * and terminate with * /.

Blank spaces are ignored, but they may not appear within the text of a keyword, a user-specified identifier, an operator, or the representation of a number.

Verilog is case sensitive, which means that uppercase and lowercase letters are distinguishable (e.g., not is not the same as NOT).

The term module refers to the text enclosed by the keyword pair module . . . endmodule. A module is the fundamental descriptive unit in the Verilog language. It is declared by the keyword module and must always be terminated by the keyword endmodule.

Combinational logic can be described by a schematic connection of gates, by a set of Boolean equations, or by a truth table. Each type of description can be developed in Verilog.



The HDL description of the circuit of Fig. above is shown in HDL Example below. The first line of text is a comment (optional) providing useful information to the reader. The second line begins with the keyword module and starts the declaration (description) of the module; the last line completes the declaration with the keyword endmodule. The keyword module is followed by a name and a list of ports. The name (Simple_Circuit in this example) is an identifier. Identifiers are names given to modules, variables (e.g., a signal), and other elements of the language so that they can be referenced in the design. In general, we choose meaningful names for modules. Identifiers are composed of alphanumeric characters and the underscore (_), and are case sensitive. Identifiers must start with an alphabetic character or an underscore, but they cannot start with a number.

```
module Simple_Circuit (A, B, C, D, E);  
  output      D, E;  
  input       A, B, C;  
  wire        w1;  
  
  and         G1 (w1, A, B); // Optional gate instance name  
  not         G2 (E, C);  
  or          G3 (D, w1, E);  
endmodule
```

The port list of a module is the interface between the module and its environment. In this example, the ports are the inputs and outputs of the circuit. The port list is enclosed in parentheses, and commas are used to separate elements of the list. The statement is terminated

with a semicolon (;). In our examples, all keywords (which must be in lowercase) are printed in bold for clarity, but that is not a requirement of the language. Next, the keywords **input** and **output** specify which of the ports are inputs and which are outputs. Internal connections are declared as wires.

The circuit in this example has one internal connection, at terminal w1, and is declared with the keyword **wire**. The structure of the circuit is specified by a list of (predefined) primitive gates, each identified by a descriptive keyword (**and**, **not**, **or**). The elements of the list are referred to as instantiations of a gate, each of which is referred to as a gate instance. Each gate instantiation consists of an optional name (such as G1, G2, etc.) followed by the gate output and inputs separated by commas and enclosed within parentheses.

The output of a primitive gate is always listed first, followed by the inputs. For example, the OR gate of the schematic is represented by the **or** primitive, is named G3, and has output D and inputs w1 and E. (Note : The output of a primitive must be listed first, but the inputs and outputs of a module may be listed in any order.) The module description ends with the keyword **endmodule**. Each statement must be terminated with a semicolon, but there is no semicolon after **endmodule**.

MODULE-2

Chapter 1: Combinational Logic

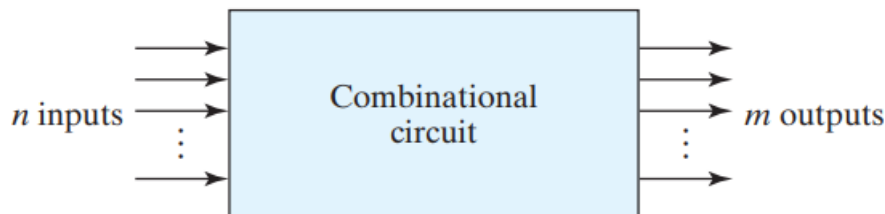
2.1 Introduction

Logic circuits for digital systems may be combinational or sequential. A combinational circuit consists of logic gates whose outputs at any time are determined from only the present combination of inputs. A combinational circuit performs an operation that can be specified logically by a set of Boolean functions.

In contrast, sequential circuits employ storage elements in addition to logic gates. Their outputs are a function of the inputs and the state of the storage elements. Because the state of the storage elements is a function of previous inputs, the outputs of a sequential circuit depend not only on present values of inputs, but also on past inputs, and the circuit behavior must be specified by a time sequence of inputs and internal states.

2.2 Combinational Circuits

A combinational circuit consists of an interconnection of logic gates. Combinational logic gates react to the values of the signals at their inputs and produce the value of the output signal, transforming binary information from the given input data to a required output data. A block diagram of a combinational circuit is shown in Fig. below.



The n input binary variables come from an external source; the m output variables are produced by the internal combinational logic circuit and go to an external destination. Each input and output variable exists physically as an analog signal whose values are interpreted to be a binary signal that represents logic 1 and logic 0. In many applications, the source and destination are storage registers. If the registers are included with the combinational gates, then the total circuit must be a sequential circuit.

For n input variables, there are 2^n possible combinations of the binary inputs. For each possible input combination, there is one possible value for each output variable. Thus, a combinational circuit can be specified with a truth table that lists the output values for each combination of input variables.

A combinational circuit also can be described by m Boolean functions, one for each output variable. Each output function is expressed in terms of the n input variables.

2.3 Design Procedure

The design of combinational circuits starts from the specification of the design objective and culminates in a logic circuit diagram or a set of Boolean functions from which the logic diagram can be obtained. The procedure involves the following steps:

1. From the specifications of the circuit, determine the required number of inputs and outputs and assign a symbol to each.
2. Derive the truth table that defines the required relationship between inputs and outputs.
3. Obtain the simplified Boolean functions for each output as a function of the input variables.
4. Draw the logic diagram and verify the correctness of the design (manually or by simulation).

A truth table for a combinational circuit consists of input columns and output columns. The input columns are obtained from the 2^n binary numbers for the n input variables. The binary values for the outputs are determined from the stated specifications. The output functions specified in the truth table give the exact definition of the combinational circuit. The output binary functions listed in the truth table are simplified by any available method, such as algebraic manipulation, the map method, or a computer-based simplification program.

Frequently, there is a variety of simplified expressions from which to choose. In a particular application, certain criteria will serve as a guide in the process of choosing an implementation. A practical design must consider such constraints as the number of gates, number of inputs to a gate, propagation time of the signal through the gates, number of interconnections, limitations of the driving capability of each gate (i.e., the number of gates to which the output of the circuit may be connected), and various other criteria that must be taken into consideration when designing integrated circuits. Since the importance of each constraint is dictated by the application, it is difficult to make a general statement about what constitutes an acceptable implementation.

In most cases, the simplification begins by satisfying an elementary objective, such as producing the simplified Boolean functions in a standard form. Then the simplification proceeds with further steps to meet other performance criteria.

Code Conversion Example

The availability of a large variety of codes for the same discrete elements of information results in the use of different codes by different digital systems. It is sometimes necessary to use the output of one system as the input to another. A conversion circuit must be inserted between the two systems if each uses different codes for the same information. Thus, a code converter is a circuit that makes the two systems compatible even though each uses a different binary code.

To convert from binary code A to binary code B, the input lines must supply the bit combination of elements as specified by code A and the output lines must generate the corresponding bit combination of code B. A combinational circuit performs this transformation by means of logic gates.

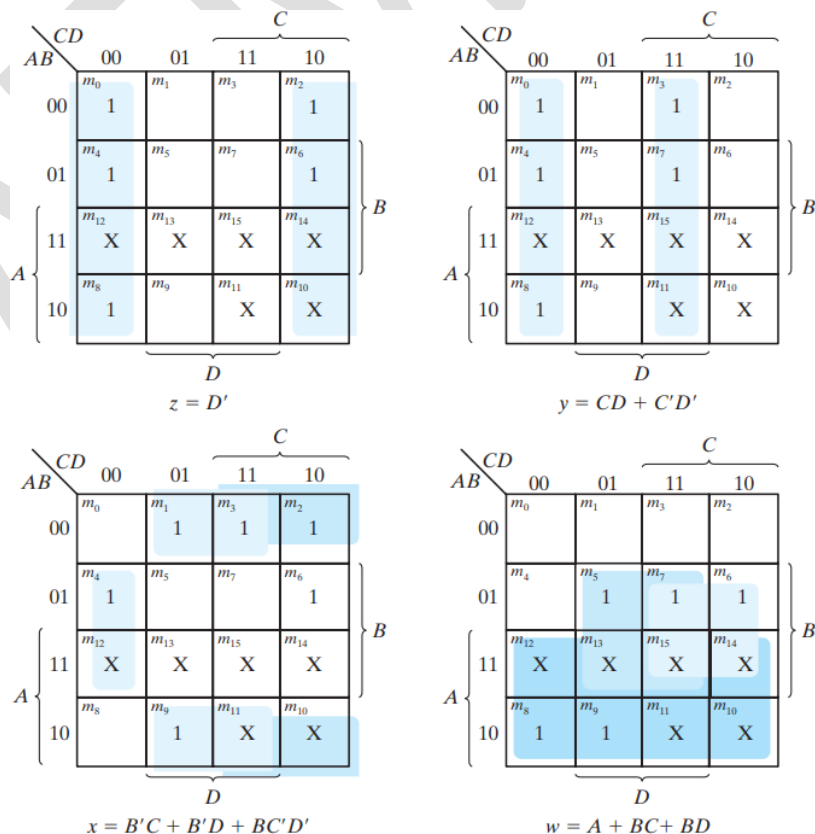
The design procedure will be illustrated by an example that converts binary coded decimal (BCD) to the excess-3 code for the decimal digits. Since each code uses four bits to represent a decimal digit, there must be four input variables and four output variables. Let the four input binary variables be the symbols A, B, C, and D, and the four output variables be w, x, y, and z. The truth table relating the input and output variables is shown in Table below. Note that four binary variables may have 16-bit combinations, but only 10 are listed in the truth table.

Truth Table for Code Conversion Example

Input BCD				Output Excess-3 Code			
A	B	C	D	w	x	y	z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

The six-bit combinations not listed for the input variables are don't-care combinations. These values have no meaning in BCD and we assume that they will never occur in actual operation of the circuit. Therefore, we are at liberty to assign to the output variables either a 1 or a 0, whichever gives a simpler circuit.

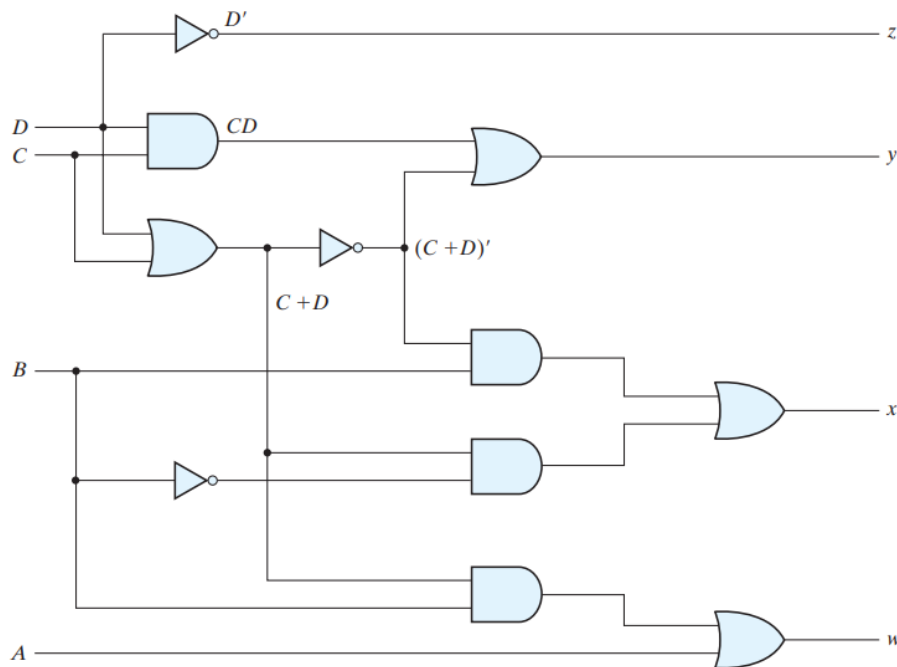
The maps in below Fig. are plotted to obtain simplified Boolean functions for the outputs. Each one of the four maps represents one of the four outputs of the circuit as a function of the four input variables. The 1's marked inside the squares are obtained from the minterms that make the output equal to 1. The 1's are obtained from the truth table by going over the output columns one at a time. For example, the column under output z has five 1's; therefore, the map for z has five 1's, each being in a square corresponding to the minterm that makes z equal to 1. The six don't-care minterms 10 through 15 are marked with an X.



A two-level logic diagram for each output may be obtained directly from the Boolean expressions derived from the maps. There are various other possibilities for a logic diagram that implements this circuit. The expressions obtained in Fig. may be manipulated algebraically for the purpose of using common gates for two or more outputs. This manipulation, shown next, illustrates the flexibility obtained with multiple-output systems when implemented with three or more levels of gates:

$$\begin{aligned} z &= D' \\ y &= CD + C'D' = CD + (C + D)' \\ x &= B'C + B'D + BC'D' = B'(C + D) + BC'D' \\ &= B'(C + D) + B(C + D)' \\ w &= A + BC + BD = A + B(C + D) \end{aligned}$$

The logic diagram that implements these expressions is shown in Fig. below.



2.4 Binary Adder- Subtractor

Digital computers perform a variety of information-processing tasks. Among the functions encountered are the various arithmetic operations. The most basic arithmetic operation is the addition of two binary digits. This simple addition consists of four possible elementary operations: $0 + 0 = 0$, $0 + 1 = 1$, $1 + 0 = 1$, and $1 + 1 = 10$. The first three operations produce a sum of one digit, but when both augend and addend bits are equal to 1, the binary sum consists of two digits. The higher significant bit of this result is called a carry. When the augend and addend numbers contain more significant digits, the carry obtained from the addition of two bits is added to the next higher order pair of significant bits.

A combinational circuit that performs the addition of two bits is called a half adder. One that performs the addition of three bits (two significant bits and a previous carry) is a full adder. The names of the circuits stem from the fact that two half adders can be employed to implement a full adder. A binary adder-subtractor is a combinational circuit that performs the arithmetic operations of addition and subtraction with binary numbers.

Half Adder

From the verbal explanation, half adder circuit needs two binary inputs and two binary outputs. The input variables designate the augend and addend bits; the output variables produce the sum and carry. Let symbols x and y be the two inputs and S (for sum) and C (for carry) to the outputs. The truth table for the half adder is listed in Table.

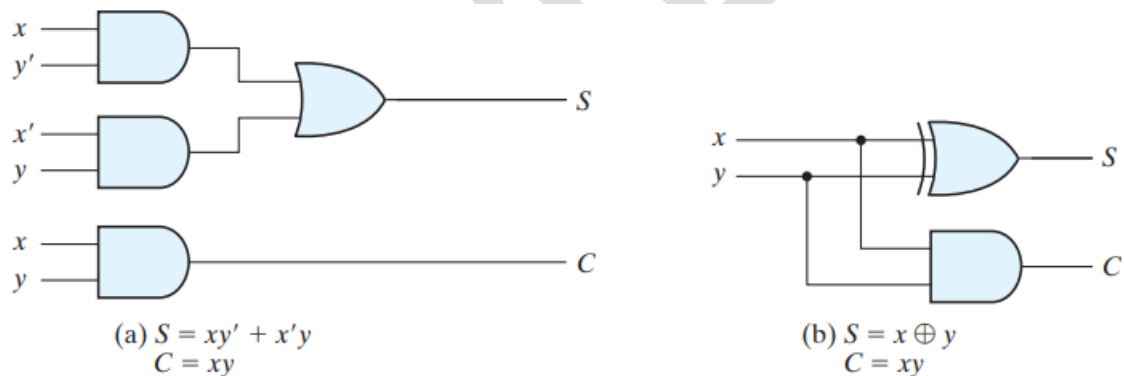
x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

The C output is 1 only when both inputs are 1. The S output represents the least significant bit of the sum. The simplified Boolean functions for the two outputs can be obtained directly from the truth table. The simplified sum-of-products expressions are

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

$$C = xy$$

The logic diagram of the half adder implemented in sum of products is shown in Fig. (a). It can be also implemented with an exclusive-OR and an AND gate as shown in Fig.(b). This form is used to show that two half adders can be used to construct a full adder.



Full Adder

A full adder is a combinational circuit that forms the arithmetic sum of three 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 representation of 2 or 3 needs two bits. 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 formed by adding the input carry and the bits of the words. The truth table of the full adder is listed in Table below.

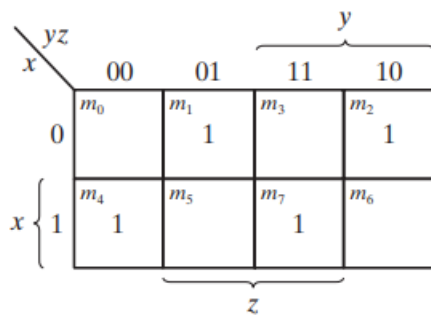
x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

The eight rows under the input variables designate all possible combinations of the three variables. 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.

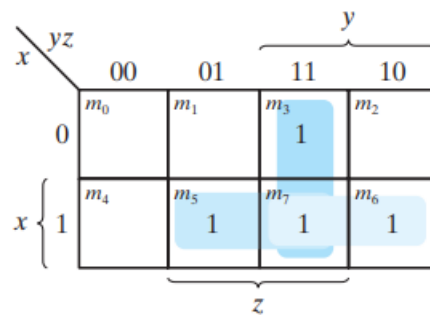
The maps for the outputs of the full adder are shown in Fig. below. The simplified expressions are,

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

$$C = xy + xz + yz$$

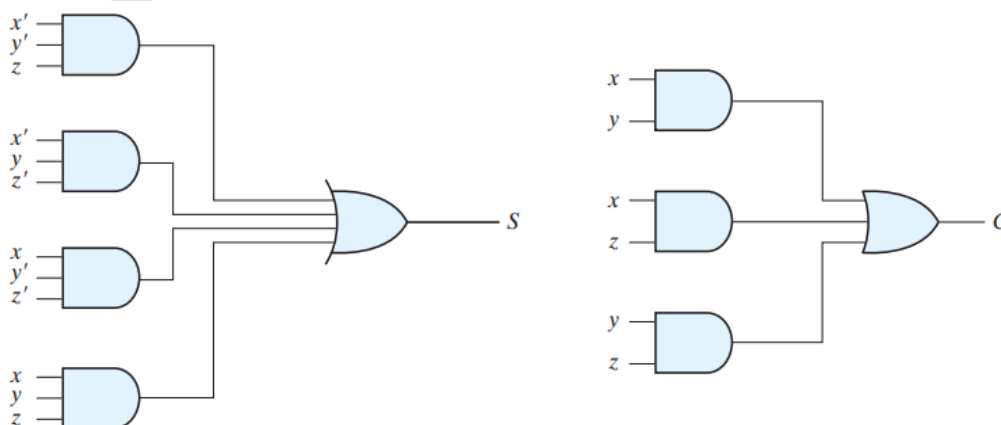


(a) $S = x'y'z + x'yz' + xy'z' + xyz$

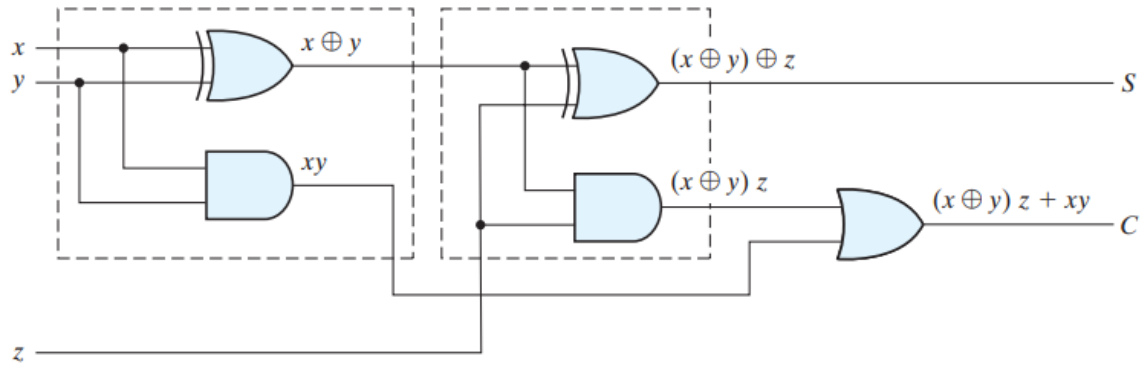


(b) $C = xy + xz + yz$

The logic diagram for the full adder implemented in sum-of-products form is shown in Fig. below.



It can also be implemented with two half adders and one OR gate, as shown below.



The S output from the second half adder is the exclusive-OR of z and the output of the first half adder, giving

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

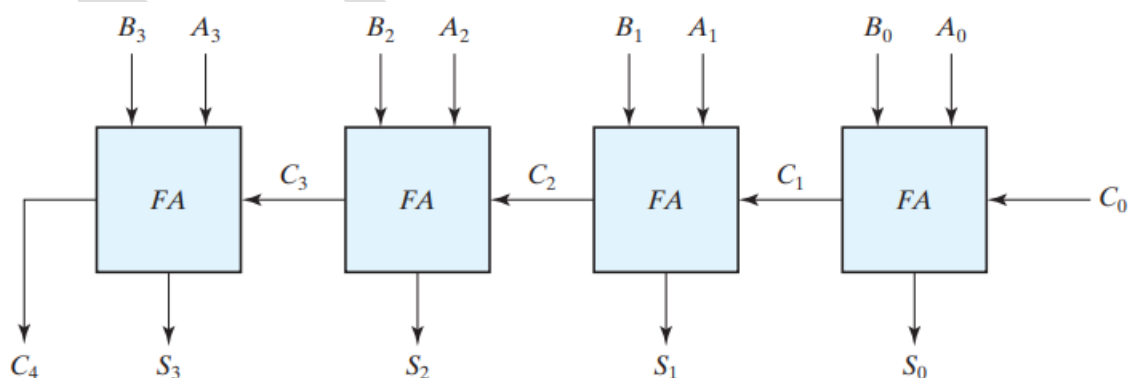
The carry output is

$$C = z(xy' + x'y) + xy = xy'z + x'yz + xy$$

Binary Adder

A binary adder is a digital circuit that produces the arithmetic sum of two binary numbers. It can be constructed with full adders connected in cascade, with the output carry from each full adder connected to the input carry of the next full adder in the chain.

Addition of n-bit numbers requires a chain of n full adders or a chain of one-half adder and n-1 full adders. In the former case, the input carry to the least significant position is fixed at 0. Figure below shows the interconnection of four full-adder (FA) circuits to provide a four-bit binary ripple carry adder.



The augend bits of A and the addend bits of B are designated by subscript numbers from right to left, with subscript 0 denoting the least significant bit. The carries are connected in a chain through the full adders. The input carry to the adder is C0, and it ripples through the full adders to the output carry C4. The S outputs generate the required sum bits. An n-bit adder requires n full adders, with each output carry connected to the input carry of the next higher order full adder. To demonstrate with a specific example, consider the two binary numbers A = 1011 and B = 0011. Their sum S = 1110 is formed with the four-bit adder as follows:

Subscript i :	3	2	1	0	
Input carry	0	1	1	0	C_i
Augend	1	0	1	1	A_i
Addend	0	0	1	1	B_i
Sum	1	1	1	0	S_i
Output carry	0	0	1	1	C_{i+1}

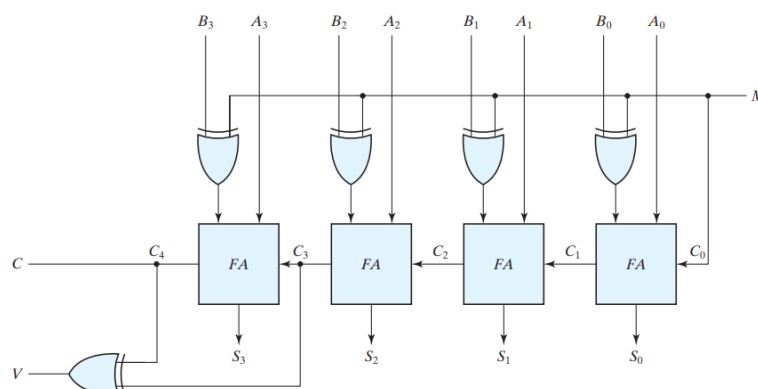
The bits are added with full adders, starting from the least significant position (subscript 0), to form the sum bit and carry bit. The input carry C_0 in the least significant position must be 0. The value of C_{i+1} in a given significant position is the output carry of the full adder. This value is transferred into the input carry of the full adder that adds the bits one higher significant position to the left. The sum bits are thus generated starting from the rightmost position and are available as soon as the corresponding previous carry bit is generated. All the carries must be generated for the correct sum bits to appear at the outputs.

Binary Subtractor

The subtraction of unsigned binary numbers can be done most conveniently by means of complements. Remember that the subtraction $A - B$ can be done by taking the 2's complement of B and adding it to A . The 2's complement can be obtained by taking the 1's complement and adding 1 to the least significant pair of bits. The 1's complement can be implemented with inverters, and a 1 can be added to the sum through the input carry.

The circuit for subtracting $A - B$ consists of an adder with inverters placed between each data input B and the corresponding input of the full adder. The input carry C_0 must be equal to 1 when subtraction is performed. The operation thus performed becomes A , plus the 1's complement of B , plus 1. This is equal to A plus the 2's complement of B . For unsigned numbers, that gives $A - B$ if $A \geq B$ or the 2's complement of $(B - A)$ if $A < B$. For signed numbers, the result is $A - B$, provided that there is no overflow.

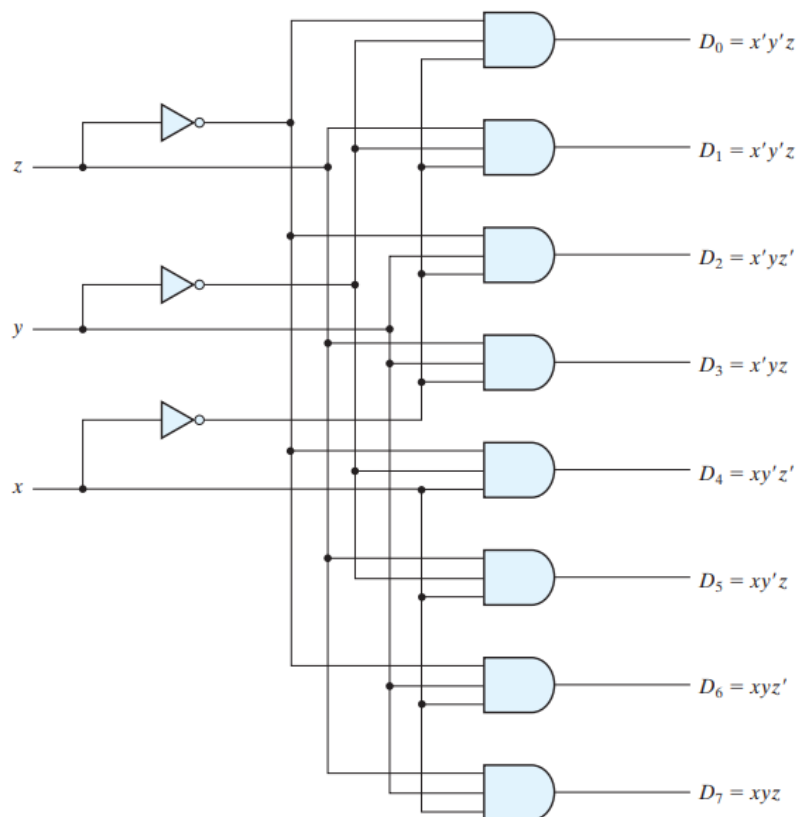
The addition and subtraction operations can be combined into one circuit with one common binary adder by including an exclusive-OR gate with each full adder. A four-bit adder-subtractor circuit is shown in Fig. below. The mode input M controls the operation. When $M = 0$, the circuit is an adder, and 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 $B \text{ ex-or } 0 = B$. 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 $B \text{ ex-or } 1 = B'$ 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 .



2.5 Decoders

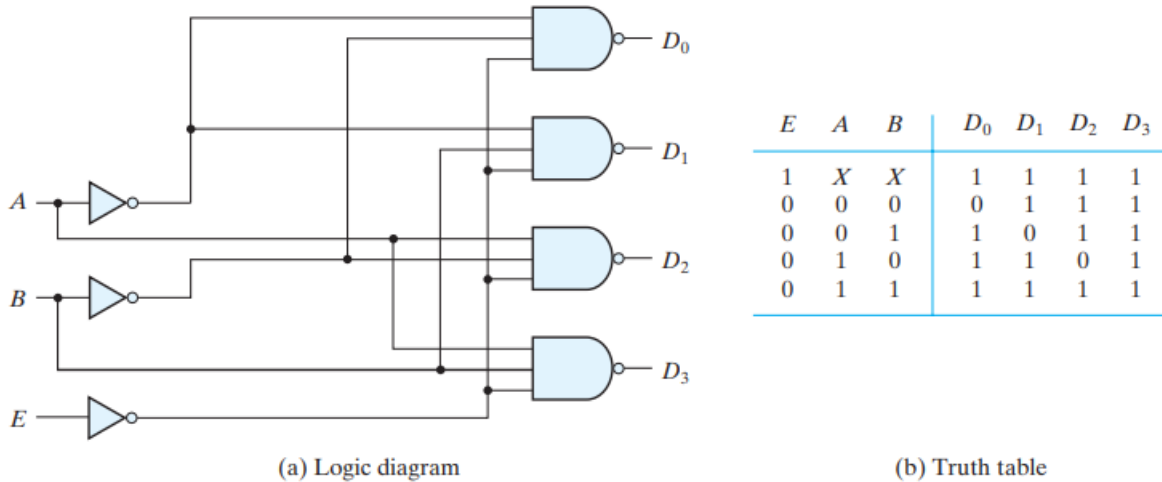
A decoder is a combinational circuit that converts binary information from n input lines to a maximum of 2^n unique output lines. If the n -bit coded information has unused combinations, the decoder may have fewer than 2^n outputs. The decoders presented here are called n -to- m -line decoders, where $m \leq 2^n$. Their purpose is to generate the 2^n (or fewer) minterms of n input variables.

Each combination of inputs will assert a unique output. The name decoder is also used in conjunction with other code converters, such as a BCD-to-seven-segment decoder. As an example, consider the three-to-eight-line decoder circuit of Fig. below. The three inputs are decoded into eight outputs, each representing one of the minterms of the three input variables. The three inverters provide the complement of the inputs, and each one of the eight AND gates generate one of the minterms. A particular application of this decoder is binary-to-octal conversion. The input variables represent a binary number, and the outputs represent the eight digits of a number in the octal number system. However, a three-to-eight-line decoder can be used for decoding any three-bit code to provide eight outputs, one for each element of the code.

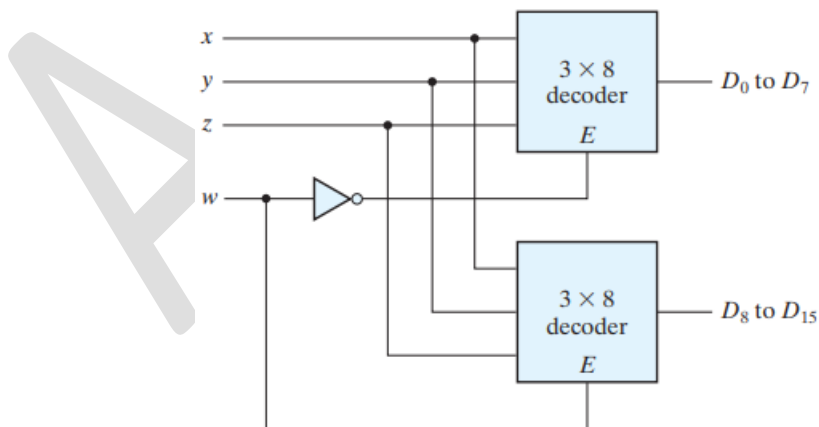


Inputs			Outputs							
x	y	z	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Furthermore, decoders include one or more enable inputs to control the circuit operation. A two-to-four-line decoder with an enable input constructed with NAND gates is shown in Fig. below. The circuit operates with complemented outputs and a complement enable input. The decoder is enabled when E is equal to 0 (i.e., active-low enable).



Decoders with enable inputs can be connected together to form a larger decoder circuit. Figure below shows two 3-to-8-line decoders with enable inputs connected to form a 4-to-16-line decoder. When $w=0$, the top decoder is enabled and the other is disabled. The bottom decoder outputs are all 0's, and the top eight outputs generate minterms 0000 to 0111. When $w=1$, the enable conditions are reversed: The bottom decoder outputs generate minterms 1000 to 1111, while the outputs of the top decoder are all 0's. This example demonstrates the usefulness of enable inputs in decoders and other combinational logic components. In general, enable inputs are a convenient feature for interconnecting two or more standard components for the purpose of combining them into a similar function with more inputs and outputs.



2.6 Encoders

An encoder is a digital circuit that performs the inverse operation of a decoder. An encoder has 2^n (or fewer) input lines and n output lines. The output lines, as an aggregate, generate the binary code corresponding to the input value.

An example of an encoder is the octal-to-binary encoder whose truth table is given in Table. It has eight inputs (one for each of the octal digits) and three outputs that generate the corresponding binary number. It is assumed that only one input has a value of 1 at any given time. The encoder can be implemented with OR gates whose inputs are determined directly from the truth table. Output z is equal to 1 when the input octal digit is 1, 3, 5, or 7. Output y is 1 for octal digits 2, 3, 6, or 7, and output x is 1 for digits 4, 5, 6, or 7. These conditions can be expressed by the following Boolean output functions:

$$z = D_1 + D_3 + D_5 + D_7$$

$$y = D_2 + D_3 + D_6 + D_7$$

$$x = D_4 + D_5 + D_6 + D_7$$

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

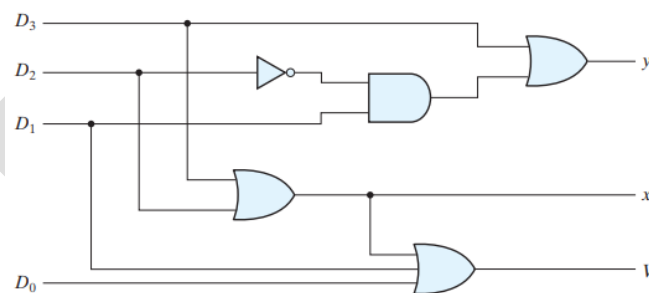
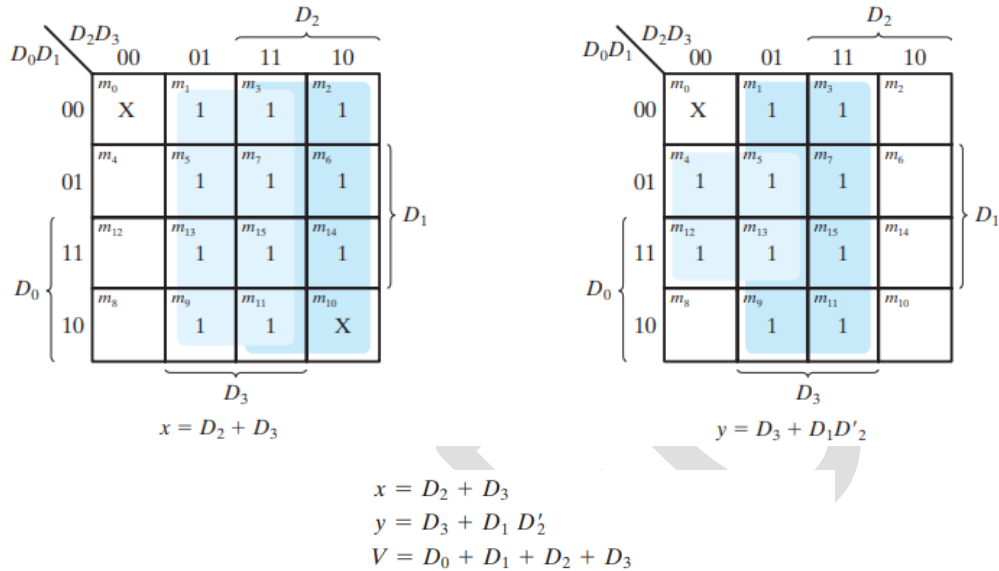
The encoder defined in Table has the limitation that only one input can be active at any given time. If two inputs are active simultaneously, the output produces an undefined combination. For example, if D_3 and D_6 are 1 simultaneously, the output of the encoder will be 111 because all three outputs are equal to 1. The output 111 does not represent either binary 3 or binary 6. To resolve this ambiguity, encoder circuits must establish an input priority to ensure that only one input is encoded.

Another ambiguity in the octal-to-binary encoder is that an output with all 0's is generated when all the inputs are 0; but this output is the same as when D_0 is equal to 1. The discrepancy can be resolved by providing one more output to indicate whether at least one input is equal to 1.

Priority Encoder

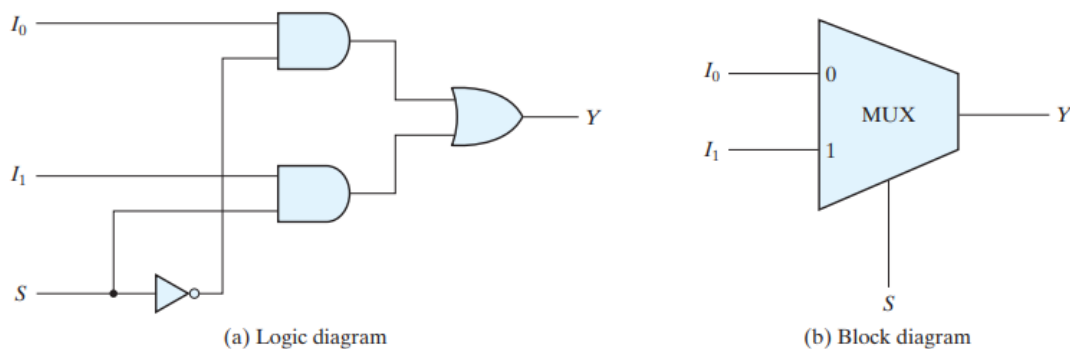
A priority encoder is an encoder circuit that includes the priority function. The operation of the priority encoder is such that if two or more inputs are equal to 1 at the same time, the input having the highest priority will take precedence. The truth table of a four-input priority encoder is given in Table. In addition to the two outputs x and y, the circuit has a third output designated by V; this is a valid bit indicator that is set to 1 when one or more inputs are equal to 1. If all inputs are 0, there is no valid input and V is equal to 0. The other two outputs are not inspected when V equals 0 and are specified as don't-care conditions. Note that whereas X's in output columns represent don't-care conditions, the X's in the input columns are useful for representing a truth table in condensed form. Instead of listing all 16 minterms of four variables, the truth table uses an X to represent either 1 or 0. For example, X 100 represents the two minterms 0100 and 1100.

Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

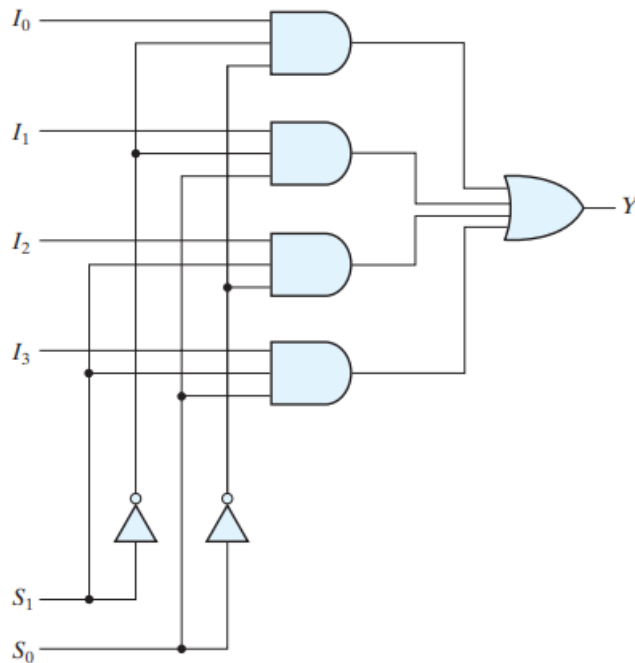


2.7 Multiplexers

A multiplexer is a combinational circuit that selects binary information from one of many input lines and directs it to a single output line. The selection of a particular input line is controlled by a set of selection lines. Normally, there are 2^n input lines and n selection lines whose bit combinations determine which input is selected.



A two-to-one-line multiplexer connects one of two 1-bit sources to a common destination, as shown in Fig. below. The circuit has two data input lines, one output line, and one selection line S . When $S = 0$, the upper AND gate is enabled and I_0 has a path to the output. When $S = 1$, the lower AND gate is enabled and I_1 has a path to the output. The multiplexer acts like an electronic switch that selects one of two sources. The block diagram of a multiplexer is sometimes depicted by a wedge-shaped symbol, as shown in Fig.(b). It suggests visually how a selected one of multiple data sources is directed into a single destination. The multiplexer is often labeled “MUX” in block diagrams.



(a) Logic diagram

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

(b) Function table

2.8 HDL Models of Combinational Circuits – Adder, Multiplexer, Encoder.

The module is the basic building block for modeling hardware with the Verilog HDL. The logic of a module can be described in anyone (or a combination) of the following modeling styles:

- Gate-level modeling using instantiations of predefined and user-defined primitive gates.
- Dataflow modeling using continuous assignment statements with the keyword assign.
- Behavioral modeling using procedural assignment statements with the keyword always.

Gate-level (structural) modeling describes a circuit by specifying its gates and how they are connected with each other. Dataflow modeling is used mostly for describing the Boolean equations of combinational logic.

Gate level modeling

A circuit of this type is identified by its logic gates and how they are connected. A schematic diagram's textual description is provided through gate-level modeling. Twelve fundamental gates are preset primitives in the Verilog HDL. They consist of and, NAND, OR, NOR, XOR, XNOR, NOT, & Buffer.

Below HDL Example shows the gate-level description of a two-to-four-line decoder. This decoder has two data inputs A and B and an enable input E. The four outputs are specified with the vector D. The wire declaration is for internal connections. Three not gates produce the complement of the inputs, and four nand gates provide the outputs for D. Remember that the output is always listed first in the port list of a primitive, followed by the inputs.

```

module decoder_2x4_gates (D, A, B, enable);
  output      [0: 3]      D;
  input       A, B;
  input       enable;
  wire        A_not,B_not, enable_not;

  not
    G1 (A_not, A),
    G2 (B_not, B),
    G3 (enable_not, enable);
  nand
    G4 (D[0], A_not, B_not, enable_not),
    G5 (D[1], A_not, B, enable_not),
    G6 (D[2], A, B_not, enable_not),
    G7 (D[3], A, B, enable_not);
endmodule

```

Data flow modeling

Combinational logic dataflow modeling employs a variety of operators that work on operands to yield desired outcomes. Around 30 distinct operators are offered by Verilog HDL. Continuous assignments and the phrase assign are used in dataflow modeling. A statement that gives a value to a net is known as a continuous assignment. Using the data type family net, a physical link between circuit components can be represented.

The dataflow description of a two-to-four-line decoder with active-low output enable and inverted output is shown in HDL Example below.

```

module decoder_2x4_df (                                     // Verilog 2001, 2005 syntax
  output      [0: 3]      D,
  input       A, B,
               enable
);
  assign      D[0] = !((!A) && (!B) && (!enable)),
               D[1] = !(*!A) && B && (!enable)),
               D[2] = !(A && B && (!enable)
               D[3] = !(A && B && (!enable))
endmodule

```

Behavioral modeling

A functional and algorithmic level representation of digital circuits is provided by behavioral modeling. Although it can also be used to describe combinational circuits, it is most frequently employed to explain sequential circuits. A list of procedural assignment statements is followed by an optional event control expression and the word always in behavioral descriptions.

HDL Example below describes the function of a four-to-one-line multiplexer. The select input is defined as a two-bit vector, and output y is declared to have type reg. The always statement, in this example, has a sequential block enclosed between the keywords case and endcase. The

block is executed whenever any of the inputs listed after the @ symbol changes in value. The case statement is a multiway conditional branch construct. Whenever in_0, in_1, in_2, in_3 or select change, the case expression (select) is evaluated and its value compared, from top to bottom, with the values in the list of statements that follow, the so-called case items. The statement associated with the first case item that matches the case expression is executed. In the absence of a match, no statement is executed. Since select is a two-bit number, it can be equal to 00, 01, 10, or 11. The case items have an implied priority because the list is evaluated from top to bottom.

```
// Behavioral description of four-to-one line multiplexer
// Verilog 2001, 2005 port syntax

module mux_4x1_beh
( output reg m_out,
  input   in_0, in_1, in_2, in_3,
  input [1: 0] select
);
always @ (in_0, in_1, in_2, in_3, select) // Verilog 2001, 2005 syntax
case (select)
  2'b00:      m_out = in_0;
  2'b01:      m_out = in_1;
  2'b10:      m_out = in_2;
  2'b11:      m_out = in_3;
endcase
endmodule
```

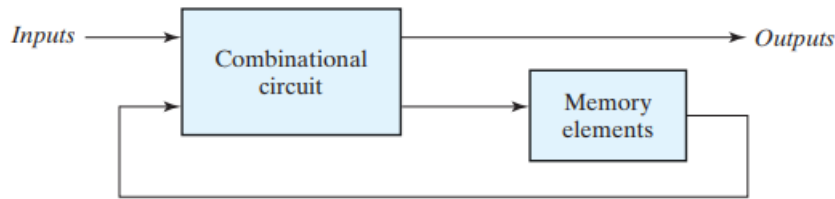
Chapter-2: Sequential Logic

2.9 Introduction

Hand-held devices, cell phones, navigation receivers, personal computers, digital cameras, personal media players, and virtually all electronic consumer products can send, receive, store, retrieve, and process information represented in a binary format. The technology enabling and supporting these devices is critically dependent on electronic components that can store information, i.e., have memory. The digital circuits considered thus far have been combinational—their output depends only and immediately on their inputs—they have no memory, i.e., dependence on past values of their inputs. Sequential circuits, however, act as storage elements and have memory. They can store, retain, and then retrieve information when needed later.

2.10 Sequential Circuits

A block diagram of a sequential circuit is shown in Fig. below. It consists of a combinational circuit to which storage elements are connected to form a feedback path. The storage elements are devices capable of storing binary information. The binary information stored in these elements at any given time defines the state of the sequential circuit at that time. The sequential circuit receives binary information from external inputs that, together with the present state of the storage elements, determine the binary value of the outputs. These external inputs also determine the condition for changing the state in the storage elements.



The block diagram demonstrates that the outputs in a sequential circuit are a function not only of the inputs, but also of the present state of the storage elements. The next state of the storage elements is also a function of external inputs and the present state. Thus, a sequential circuit is specified by a time sequence of inputs, outputs, and internal states. In contrast, the outputs of combinational logic depend only on the present values of the inputs. There are two main types of sequential circuits, and their classification is a function of the timing of their signals. A synchronous sequential circuit is a system whose behavior can be defined from the knowledge of its signals at discrete instants of time. The behavior of an asynchronous sequential circuit depends upon the input signals at any instant of time and the order in which the inputs change. The storage elements commonly used in asynchronous sequential circuits are time-delay devices.

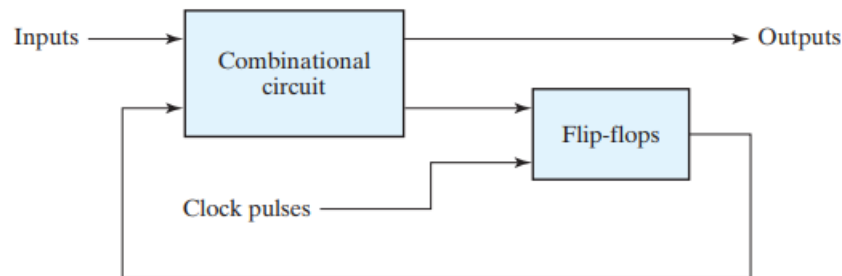
The storage capability of a time-delay device varies with the time it takes for the signal to propagate through the device. In practice, the internal propagation delay of logic gates is of sufficient duration to produce the needed delay, so that actual delay units may not be necessary. In gate-type asynchronous systems, the storage elements consist of logic gates whose propagation delay provides the required storage. Thus, an asynchronous sequential circuit may be regarded as a combinational circuit with feedback. Because of the feedback among logic gates, an asynchronous sequential circuit may become unstable at times.

A synchronous sequential circuit employs signals that affect the storage elements at only discrete instants of time. Synchronization is achieved by a timing device called a clock generator, which provides a clock signal having the form of a periodic train of clock pulses. The clock signal is commonly denoted by the identifiers *clock* and *clk*. The clock pulses are distributed throughout the system in such a way that storage elements are affected only with the arrival of each pulse. In practice, the clock pulses determine when computational activity will occur within the circuit, and other signals (external inputs and otherwise) determine what changes will take place affecting the storage elements and the outputs.

For example, a circuit that is to add and store two binary numbers would compute their sum from the values of the numbers and store the sum at the occurrence of a clock pulse. Synchronous sequential circuits that use clock pulses to control storage elements are called clocked sequential circuits and are the type most frequently encountered in practice. They are called synchronous circuits because the activity within the circuit and the resulting updating of stored values is synchronized to the occurrence of clock pulses. The design of synchronous circuits is feasible because they seldom manifest instability problems and their timing is easily broken down into independent discrete steps, each of which can be considered separately.

The storage elements (memory) used in clocked sequential circuits are called flipflops. A flip-flop is a binary storage device capable of storing one bit of information. In a stable state, the output of a flip-flop is either 0 or 1. A sequential circuit may use many flip-flops to store as many bits as necessary. The block diagram of a synchronous clocked sequential circuit is shown

in Fig. below. The outputs are formed by a combinational logic function of the inputs to the circuit, or the values stored in the flip-flops (or both). The value that is stored in a flip-flop when the clock pulse occurs is also determined by the inputs to the circuit or the values presently stored in the flip-flop (or both). The new value is stored (i.e., the flip-flop is updated) when a pulse of the clock signal occurs.



(a) Block diagram



(b) Timing diagram of clock pulses

Prior to the occurrence of the clock pulse, the combinational logic forming the next value of the flip-flop must have reached a stable value. Consequently, the speed at which the combinational logic circuits operate is critical. If the clock (synchronizing) pulses arrive at a regular interval, as shown in the timing diagram, the combinational logic must respond to a change in the state of the flip-flop in time to be updated before the next pulse arrives.

Propagation delays play an important role in determining the minimum interval between clock pulses that will allow the circuit to operate correctly. A change in state of the flip-flops is initiated only by a clock pulse transition—for example, when the value of the clock signals changes from 0 to 1. When a clock pulse is not active, the feedback loop between the value stored in the flip-flop and the value formed at the input to the flip-flop is effectively broken because the flip-flop outputs cannot change even if the outputs of the combinational circuit driving their inputs change in value. Thus, the transition from one state to the next occurs only at predetermined intervals dictated by the clock pulses.

2.11 Storage Elements

Latches

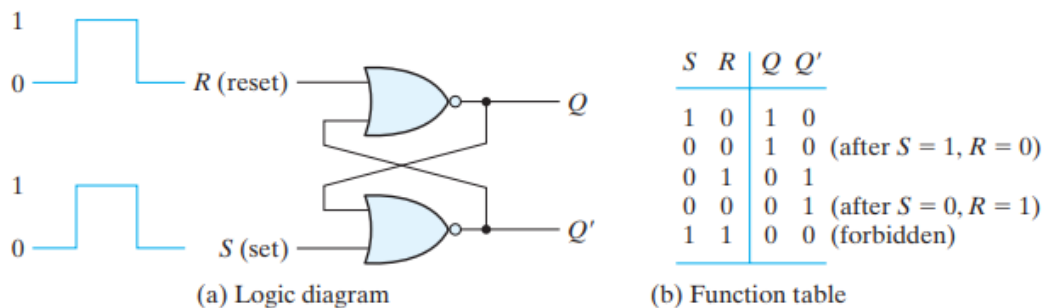
A storage element in a digital circuit can maintain a binary state indefinitely, until directed by an input signal to switch states. The major differences among various types of storage elements are in the number of inputs they possess and in the way the inputs affect the binary state.

Storage elements that operate with signal levels (rather than signal transitions) are referred to as latches; those controlled by a clock transition are flip-flops. Latches are said to be level sensitive devices; flip-flops are edge-sensitive devices. The two types of storage elements are related because latches are the basic circuits from which all flip-flops are constructed. Although latches are useful for storing binary information and for the design of asynchronous sequential circuits, they are not practical for use as storage elements in synchronous sequential circuits.

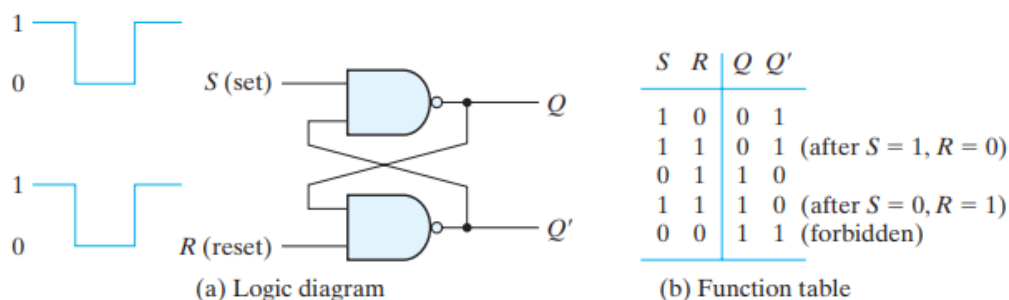
SR Latch

The SR latch is a circuit with two cross-coupled NOR gates or two cross-coupled NAND gates, and two inputs labeled S for set and R for reset. The SR latch constructed with two cross-coupled NOR gates is shown in Fig. below. The latch has two useful states.

When output $Q = 1$ and $Q' = 0$, the latch is said to be in the set state. When $Q = 0$ and $Q' = 1$, it is in the reset state. Outputs Q and Q' are normally the complement of each other. However, when both inputs are equal to 1 at the same time, a condition in which both outputs are equal to 0 (rather than be mutually complementary) occurs. If both inputs are then switched to 0 simultaneously, the device will enter an unpredictable or undefined state or a meta-stable state. Consequently, in practical applications, setting both inputs to 1 is forbidden. Under normal conditions, both inputs of the latch remain at 0 unless the state has to be changed. The application of a momentary 1 to the S input causes the latch to go to the set state. The S input must go back to 0 before any other changes take place, in order to avoid the occurrence of an undefined next state that results from the forbidden input condition.



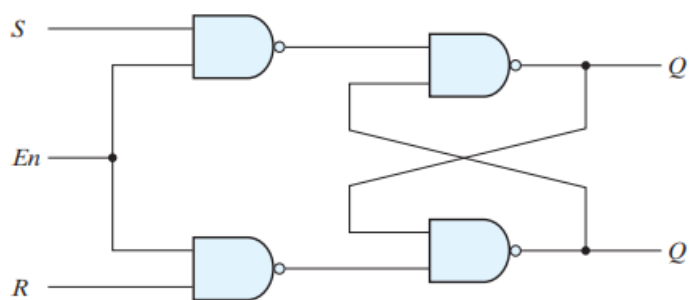
As shown in the function table of Fig. (b), two input conditions cause the circuit to be in the set state. The first condition ($S = 1, R = 0$) is the action that must be taken by input S to bring the circuit to the set state. Removing the active input from S leaves the circuit in the same state. After both inputs return to 0, it is then possible to shift to the reset state by momentarily applying a 1 to the R input. The 1 can then be removed from R, whereupon the circuit remains in the reset state. Thus, when both inputs S and R are equal to 0, the latch can be in either the set or the reset state, depending on which input was most recently a 1. If a 1 is applied to both the S and R inputs of the latch, both outputs go to 0. This action produces an undefined next state because the state that results from the input transitions depends on the order in which they return to 0. It also violates the requirement that outputs be the complement of each other. In normal operation, this condition is avoided by making sure that 1's are not applied to both inputs simultaneously.



The SR latch with two cross-coupled NAND gates is shown in Fig. above. It operates with both inputs normally at 1 unless the state of the latch has to be changed. The application of 0 to the S input causes output Q to go to 1, putting the latch in the set state. When the S input goes back to 1, the circuit remains in the set state. After both inputs go back to 1, we are allowed to change the state of the latch by placing a 0 in the R input. This action causes the circuit to go to the reset state and stay there even after both inputs return to 1. The condition that is forbidden for the NAND latch is both inputs being equal to 0 at the same time, an input combination that should be avoided.

In comparing the NAND with the NOR latch, note that the input signals for the NAND require the complement of those values used for the NOR latch. Because the NAND latch requires a 0 signal to change its state, it is sometimes referred to as an S'R' latch. The primes (or, sometimes, bars over the letters) designate the fact that the inputs must be in their complement form to activate the circuit. The operation of the basic SR latch can be modified by providing an additional input signal that determines (controls) when the state of the latch can be changed by determining whether S and R (or S' and R') can affect the circuit.

An SR latch with a control input is shown in Fig. below. It consists of the basic SR latch and two additional NAND gates. The control input En acts as an enable signal for the other two inputs. The outputs of the NAND gates stay at the logic-1 level as long as the enable signal remains at 0. This is the quiescent condition for the SR latch. When the enable input goes to 1, information from the S or R input is allowed to affect the latch. The set state is reached with S = 1, R = 0, and En = 1 (active-high enabled). To change to the reset state, the inputs must be S = 0, R = 1, and En = 1. In either case, when En returns to 0, the circuit remains in its current state. The control input disables the circuit by applying 0 to En, so that the state of the output does not change regardless of the values of S and R. Moreover, when En = 1 and both the S and R inputs are equal to 0, the state of the circuit does not change. These conditions are listed in the function table accompanying the diagram. An indeterminate condition occurs when all three inputs are equal to 1. This condition places 0's on both inputs of the basic SR latch, which puts it in the undefined state. When the enable input goes back to 0, one cannot conclusively determine the next state, because it depends on whether the S or R input goes to 0 first. This indeterminate condition makes this circuit difficult to manage, and it is seldom used in practice.



(a) Logic diagram

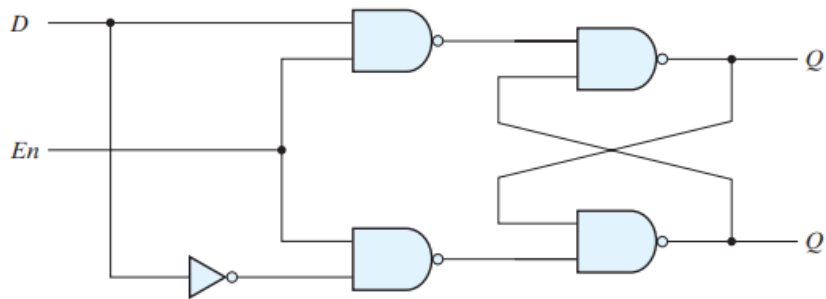
En	S	R	Next state of Q
0	X	X	No change
1	0	0	No change
1	0	1	Q = 0; reset state
1	1	0	Q = 1; set state
1	1	1	Indeterminate

(b) Function table

D Latch (Transparent Latch)

One way to eliminate the undesirable condition of the indeterminate state in the SR latch is to ensure that inputs S and R are never equal to 1 at the same time. This is done in the D latch, shown in Fig. below. This latch has only two inputs: D (data) and En (enable). The D input goes directly to the S input, and its complement is applied to the R input. As long as the enable

input is at 0, the cross-coupled SR latch has both inputs at the 1 level and the circuit cannot change state regardless of the value of D. The D input is sampled when $En = 1$. If $D = 1$, the Q output goes to 1, placing the circuit in the set state. If $D = 0$, output Q goes to 0, placing the circuit in the reset state.



(a) Logic diagram

En	D	Next state of Q
0	X	No change
1	0	$Q = 0$; reset state
1	1	$Q = 1$; set state

(b) Function table

The D latch receives that designation from its ability to hold data in its internal storage. It is suited for use as a temporary storage for binary information between a unit and its environment. The binary information present at the data input of the D latch is transferred to the Q output when the enable input is asserted. The output follows changes in the data input as long as the enable input is asserted. This situation provides a path from input D to the output, and for this reason, the circuit is often called a transparent latch. When the enable input signal is deasserted, the binary information that was present at the data input at the time the transition occurred is retained (i.e., stored) at the Q output until the enable input is asserted again. Note that an inverter could be placed at the enable input. Then, depending on the physical circuit, the external enabling signal will be a value of 0 (active low) or 1 (active high).

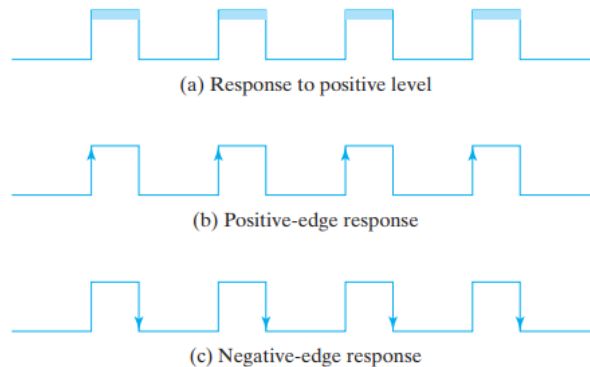
Flip-flops

The state of a latch or flip-flop is switched by a change in the control input. This momentary change is called a trigger, and the transition it causes is said to trigger the flip-flop. The D latch with pulses in its control input is essentially a flip-flop that is triggered every time the pulse goes to the logic-1 level. As long as the pulse input remains at this level, any changes in the data input will change the output and the state of the latch.

When latches are used for the storage elements, a serious difficulty arises. The state transitions of the latches start as soon as the clock pulse changes to the logic-1 level. The new state of a latch appears at the output while the pulse is still active. This output is connected to the inputs of the latches through the combinational circuit. If the inputs applied to the latches change while the clock pulse is still at the logic-1 level, the latches will respond to new values and a new output state may occur. The result is an unpredictable situation, since the state of the latches may keep changing for as long as the clock pulse stays at the active level. Because of this unreliable operation, the output of a latch cannot be applied directly or through combinational logic to the input of the same or another latch when all the latches are triggered by a common clock source.

Flip-flop circuits are constructed in such a way as to make them operate properly when they are part of a sequential circuit that employs a common clock. The problem with the latch is that it responds to a change in the level of a clock pulse. As shown in Fig. (a), a positive level

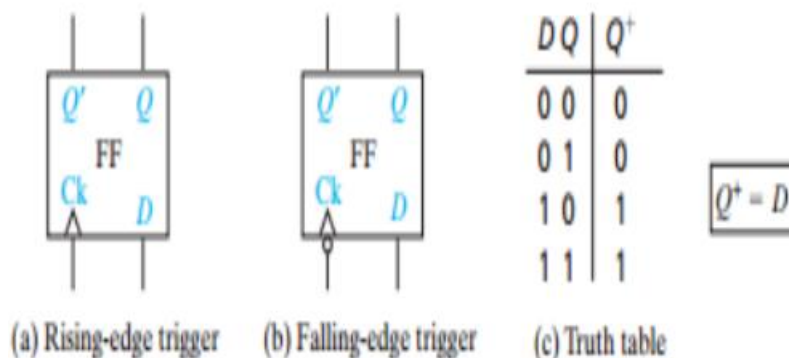
response in the enable input allows changes in the output when the D input changes while the clock pulse stays at logic 1. The key to the proper operation of a flip-flop is to trigger it only during a signal transition. This can be accomplished by eliminating the feedback path that is inherent in the operation of the sequential circuit using latches. A clock pulse goes through two transitions: from 0 to 1 and the return from 1 to 0. As shown in Fig. below, the positive transition is defined as the positive edge and the negative transition as the negative edge. There are two ways that a latch can be modified to form a flip-flop. One way is to employ two latches in a special configuration that isolates the output of the flip-flop and prevents it from being affected while the input to the flip-flop is changing.

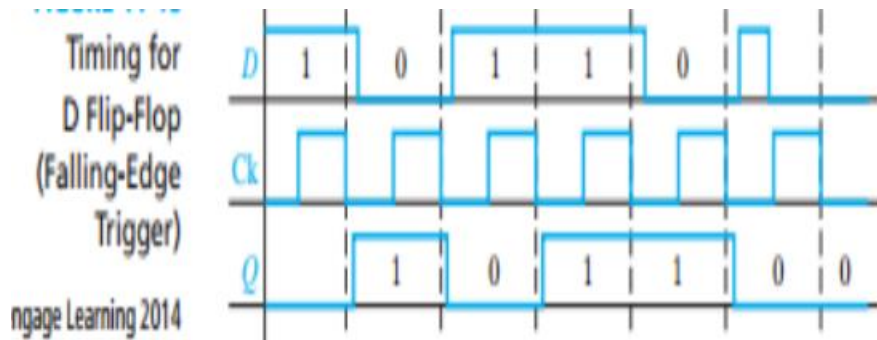


Another way is to produce a flip-flop that triggers only during a signal transition (from 0 to 1 or from 1 to 0) of the synchronizing signal (clock) and is disabled during the rest of the clock pulse. We will now proceed to show the implementation of both types of flip-flops.

Edge-Triggered D Flip Flop

A D flip-flop has two inputs, D (data) and Ck (clock). The small arrowhead on the flip-flop symbol identifies the clock input. Unlike the D latch, the flip-flop output changes only in response to the clock, not to a change in D. If the output can change in response to a 0 to 1 transition on the clock input, we say that the flip-flop is triggered on the rising edge (or positive edge) of the clock. If the output can change in response to a 1 to 0 transition on the clock input, we say that the flip-flop is triggered on the falling edge (or negative edge) of the clock. An inversion bubble on the clock input indicates a falling-edge trigger (Figure (b)), and no bubble indicates a rising-edge trigger (Figure (a)). The term active edge refers to the clock edge (rising or falling) that triggers the flip-flop state change.





MODULE-3

Basic Structure of Computers

Introduction

Computer Architecture (CA) is concerned with the structure and behaviour of the computer. CA includes the information formats, the instruction set and techniques for addressing memory.

In general covers, CA covers 3 aspects of computer-design namely: 1) Computer Hardware, 2) Instruction set Architecture and 3) Computer Organization.

1. Computer Hardware: It consists of electronic circuits, displays, magnetic and optical storage media and communication facilities.

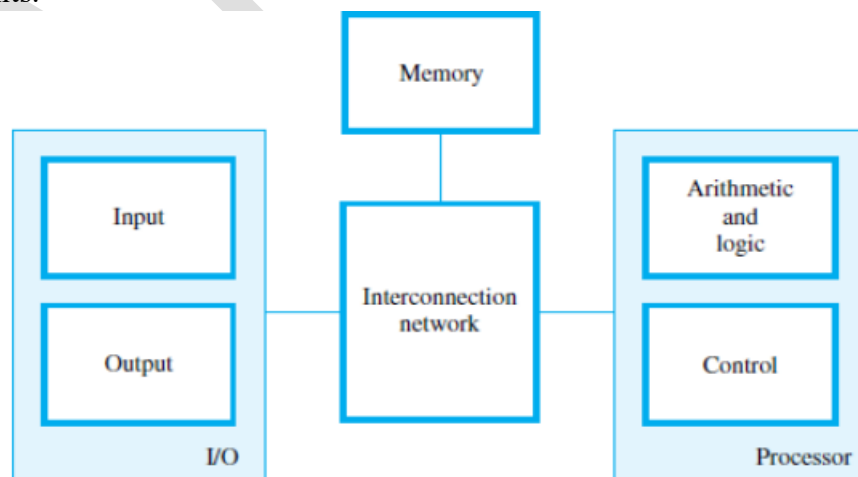
2. Instruction Set Architecture: It is programmer visible machine interface such as instruction set, registers, memory organization and exception handling. Two main approaches are 1) CISC and 2) RISC. (CISC Complex Instruction Set Computer, RISC Reduced Instruction Set Computer)

3. Computer Organization: It includes the high-level aspects of a design, such as memory-system, bus-structure & design of the internal CPU. It refers to the operational units and their interconnections that realize the architectural specifications. It describes the function of and design of the various units of digital computer that store and process information.

3.1 Functional Units

A computer consists of 5 functionally independent main parts:

- 1) Input
- 2) Memory
- 3) ALU
- 4) Output&
- 5) Control units.



Functional unit of computer

3.2 Basic Operational Concepts

An Instruction consists of 2 parts, 1) Operation code (Opcode) and 2) Operands.

The data/operands are stored in memory. The individual instruction is brought from the memory to the processor. Then, the processor performs the specified operation.

Let us see a typical instruction ADD LOCA, R0

This instruction is an addition operation. The following are the steps to execute the instruction:

Step 1: Fetch the instruction from main-memory into the processor.

Step 2: Fetch the operand at location LOCA from main-memory into the processor.

Step 3: Add the memory operand (i.e. fetched contents of LOCA) to the contents of register R0.

Step 4: Store the result (sum) in R0.

The same instruction can be realized using 2 instructions as: Load LOCA, R1

Add R1, R0

The following are the steps to execute the instruction:

Step 1: Fetch the instruction from main-memory into the processor.

Step 2: Fetch the operand at location LOCA from main-memory into the register R1.

Step 3: Add the content of Register R1 and the contents of register R0.

Step 4: Store the result (sum) in R0.

Main Parts of Processor

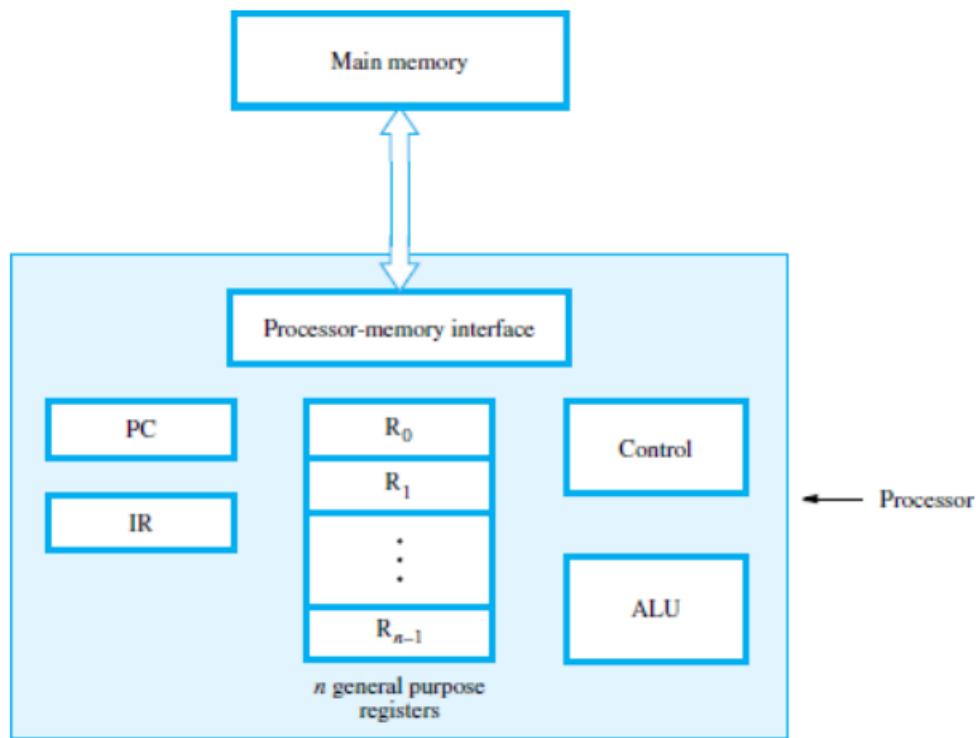
The processor contains ALU, control-circuitry and many registers. The processor contains 'n' general-purpose registers R0 through Rn-1. The IR holds the instruction that is currently being executed. The control-unit generates the timing-signals that determine when a given action is to take place. The PC contains the memory-address of the next-instruction to be fetched & executed. During the execution of an instruction, the contents of PC are updated to point to next instruction. The MAR holds the address of the memory-location to be accessed. The MDR contains the data to be written into or read out of the addressed location. MAR and MDR facilitates the communication with memory.

(IR Instruction-Register, PC-Program Counter, MAR-Memory Address Register, MDR-Memory Data Register)

Steps To Execute an Instruction

- 1) The address of first instruction (to be executed) gets loaded into PC.
- 2) The contents of PC (i.e. address) are transferred to the MAR & control-unit issues Read signal to memory.
- 3) After certain amount of elapsed time, the first instruction is read out of memory and placed into MDR.
- 4) Next, the contents of MDR are transferred to IR. At this point, the instruction can be decoded & executed.
- 5) To fetch an operand, its address is placed into MAR & control-unit issues Read signal. As a result, the operand is transferred from memory into MDR, and then it is transferred from MDR to ALU.

- 6) Likewise required number of operands is fetched into processor.
- 7) Finally, ALU performs the desired operation.
- 8) If the result of this operation is to be stored in the memory, then the result is sent to the MDR.
- 9) The address of the location where the result is to be stored is sent to the MAR and a Write cycle is initiated.
- 10) At some point during execution, contents of PC are incremented to point to next instruction in the program.

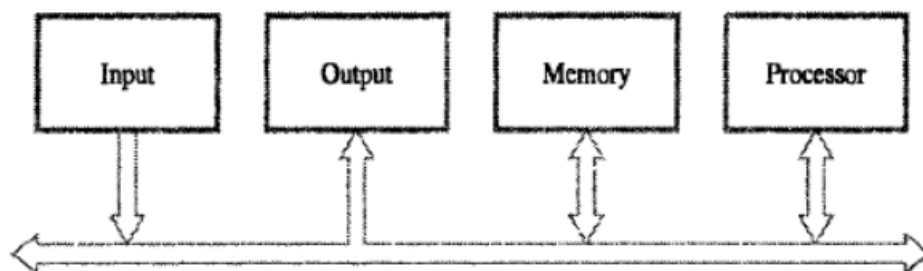


Connection between processor and the main memory

3.3 Bus structure

A bus is a group of lines that serves as a connecting path for several devices. A bus may be lines or wires. The lines carry data or address or control signal. There are 2 types of Bus structures:

- 1) Single Bus Structure: Because the bus can be used for only one transfer at a time, only 2 units can actively use the bus at any given time. Bus control lines are used to arbitrate multiple requests for use of the bus. Advantages: 1. Low cost & 2. Flexibility for attaching peripheral devices.



Single bus architecture

2) Multiple Bus Structure: Systems that contain multiple buses achieve more concurrency in operations. Two or more transfers can be carried out at the same time. Advantage: Better performance. Disadvantage: Increased cost.

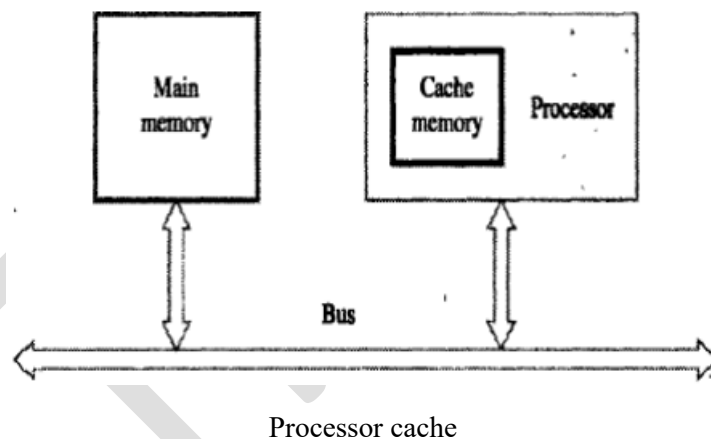
The devices connected to a bus vary widely in their speed of operation.

- To synchronize their operational-speed, buffer-registers can be used. Buffer Registers are included with the devices to hold the information during transfers. prevent a high-speed processor from being locked to a slow I/O device during data transfers.

3.4 Performance

The most important measure of performance of a computer is how quickly it can execute programs. The speed of a computer is affected by the design of 1) Instruction-set. 2) Hardware & the technology in which the hardware is implemented. 3) Software including the operating system.

Because programs are usually written in an HLL, performance is also affected by the compiler that translates programs into machine language. (HLL High Level Language). For best performance, it is necessary to design the compiler, machine instruction set and hardware in a co-ordinated way.



Let us examine the flow of program instructions and data between the memory & the processor.

- At the start of execution, all program instructions are stored in the main-memory.
- As execution proceeds, instructions are fetched into the processor, and a copy is placed in the cache.
- Later, if the same instruction is needed a second time, it is read directly from the cache.
- A program will be executed faster if movement of instruction/data between the main-memory and the processor is minimized which is achieved by using the cache.

Processor Clock

Processor circuits are controlled by a timing signal called a Clock. The clock defines regular time intervals called Clock Cycles. To execute a machine instruction, the processor divides the action to be performed into a sequence of basic steps such that each step can be completed in one clock cycle.

Let P = Length of one clock cycle R = Clock rate. Relation between P and R is given by $R=(1/P)$, R is measured in cycles per second. Cycles per second is also called Hertz(Hz)

Basic Performance Equation

Let T = Processor time required to executed a program.

N = Actual number of instruction executions.

S = Average number of basic steps needed to execute one machine instruction.

R = Clock rate in cycles per second.

The program execution time is given by $T=(N*S)/R$ -----(1)

Equ1 is referred to as the basic performance equation. To achieve high performance, the computer designer must reduce the value of T , which means reducing N and S , and increasing R .

The value of N is reduced if source program is compiled into fewer machine instructions. The value of S is reduced if instructions have a smaller number of basic steps to perform. The value of R can be increased by using a higher frequency clock. Care has to be taken while modifying values since changes in one parameter may affect the other.

Clock Rate

There are 2 possibilities for increasing the clock rate R :

- 1) Improving the IC technology makes logic-circuits faster. This reduces the time needed to compute a basic step. (IC integrated circuits). This allows the clock period P to be reduced and the clock rate R to be increased.
- 2) Reducing the amount of processing done in one basic step also reduces the clock period P . In presence of a cache, the percentage of accesses to the main memory is small. Hence, much of performance-gain expected from the use of faster technology can be realized. The value of T will be reduced by same factor as R is increased ' '. S & N are not affected.

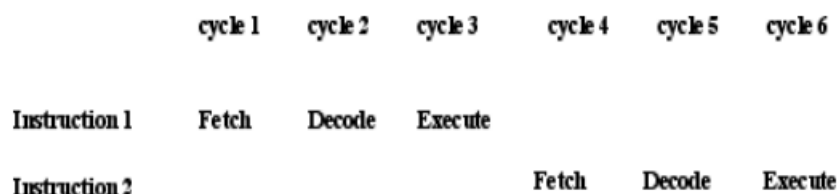
Pipelining & Superscalar Operation

Normally, the computer executes the instruction in sequence one by one. An improvement in performance can be achieved by overlapping the execution of successive instructions using a technique called Pipelining.

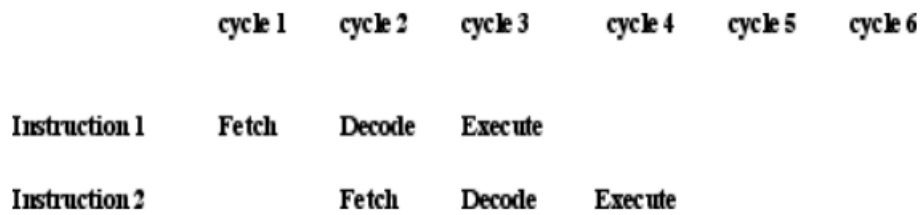
Consider the instruction Add R1,R2,R3 ;

Move R4,R5 ;

Let us assume that both operations take 3 clock cycles each for completion.



As shown in above figure, 6 clock cycles are required to complete two operations.



As shown in above figure, if we use pipelining & pre-fetching, only 4 cycles are required to complete same two operations. While executing the Add instruction, the processor can read the Move instruction from memory. In the ideal case, if all instructions are overlapped to the maximum degree possible, execution proceeds at the rate of one instruction completed in each clock cycle.

A higher degree of concurrency can be achieved if multiple instruction pipelines are implemented in the processor i.e. multiple functional units can be used to execute different instructions parallelly. This mode of operation is known as Superscalar Execution. With Superscalar arrangement, it is possible to complete the execution of more than one instruction in every clock cycle.

Performance Measurement

Benchmark refers to standard task used to measure how well a processor operates. The Performance Measure is the time taken by a computer to execute a given benchmark. SPEC selects & publishes the standard programs along with their test results for different application domains. (SPEC System Performance Evaluation Corporation).

SPEC Rating is given by $\text{SPEC rating} = (\text{Running time on the reference computer}) / (\text{Running time on the computer under test})$

SPEC rating = 50 The computer under test is 50 times as fast as reference-computer. The test is repeated for all the programs in the SPEC suite. Then, the geometric mean of the results is computed.

Let SPEC_i = Rating for program 'i' in the suite. Overall SPEC rating for the computer is given by,

$$\text{SPEC rating} = \left(\prod_{i=1}^n \text{SPEC}_i \right)^{\frac{1}{n}}$$

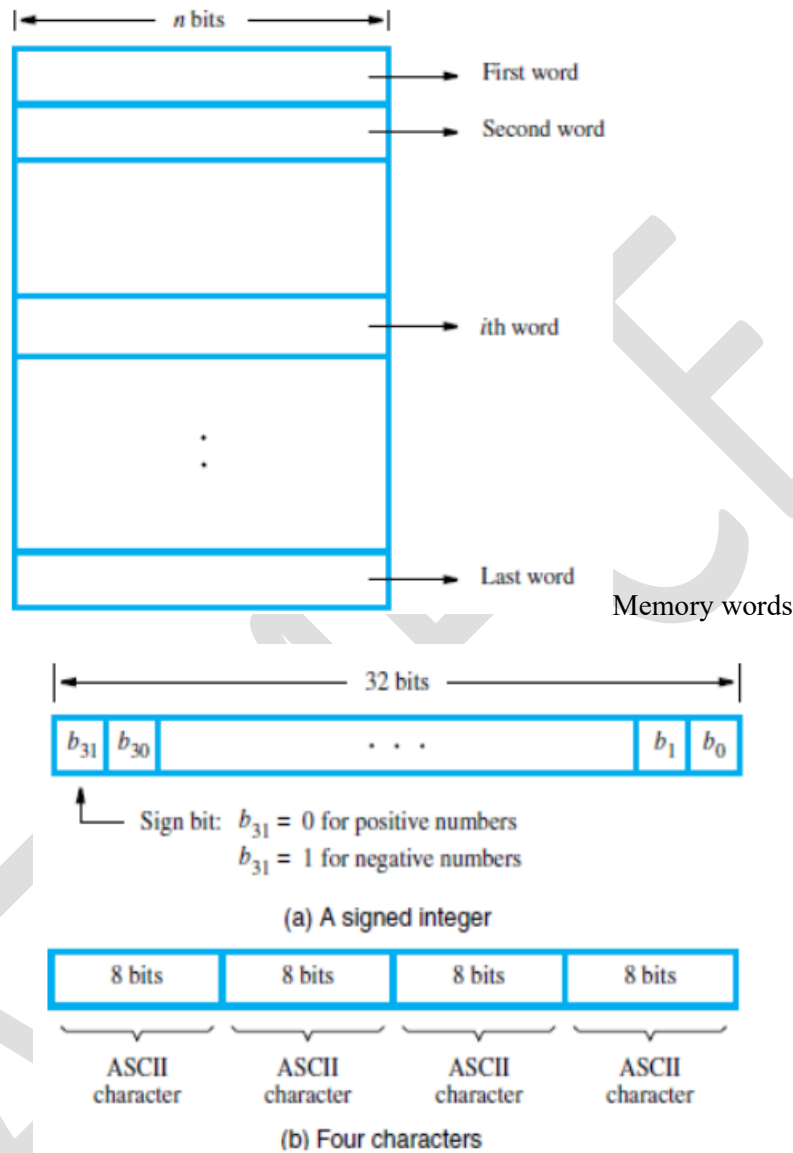
where n = no. of programs in the suite.

Machine Instructions and Programs

3.5 Memory Location and Addresses

The memory consists of many millions of storage cells(flip-flops). Each cell can store a bit of information i.e., 0 or 1. Each group of n bits is referred to as a word of information, and n is called the word length. The word length can vary from 8 to 64bits. A unit of 8 bits is called a byte. Accessing the memory to store or retrieve a single item of information (word/byte)

requires distinct addresses for each item location. (It is customary to use numbers from 0 through $2^k - 1$ as the addresses of successive locations in the memory). If $2^k = \text{no. of addressable locations}$ then 2^k addresses constitute the address-space of the computer. For example, a 24-bit address generates an address-space of 2^{24} locations (16 MB).



Byte-Addressability

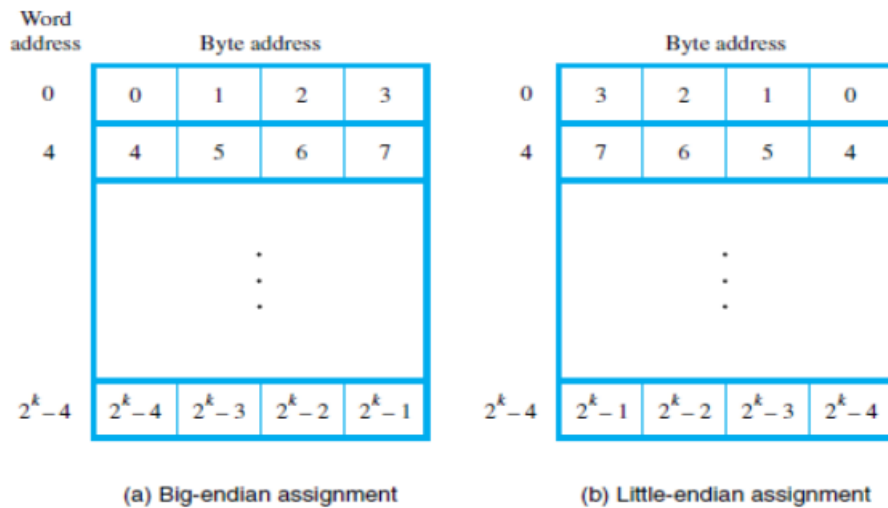
In byte-addressable memory, successive addresses refer to successive byte locations in the memory. Byte locations have addresses 0, 1, 2, If the word-length is 32 bits, successive words are located at addresses 0, 4, 8, . . . with each word having 4bytes.

Big-Endian & Little-Endian Assignments

There are two ways in which byte-addresses are arranged.

- 1) Big-Endian: Lower byte-addresses are used for the more significant bytes of the word.
- 2) Little-Endian: Lower byte-addresses are used for the less significant bytes of the word

In both cases, byte-addresses 0, 4, 8, . . . are taken as the addresses of successive words in the memory. Consider a 32-bit integer (in hex): 0x12345678 which consists of 4 bytes: 12, 34, 56, and 78. Hence this integer will occupy 4 bytes in memory. Assume, we store it at memory address starting 1000.



Word Alignment

Words are said to be Aligned in memory if they begin at a byte-address that is a multiple of the number of bytes in a word. For example, If the word length is 16(2 bytes), aligned words begin at byte-addresses 0, 2, 4 If the word length is 64(2 bytes), aligned words begin at byte-addresses 0, 8, 16 Words are said to have Unaligned Addresses, if they begin at an arbitrary byte-address.

Accessing Numbers, Characters & Characters Strings

A number usually occupies one word. It can be accessed in the memory by specifying its word address. Similarly, individual characters can be accessed by their byte-address. There are two ways to indicate the length of the string:

- 1) A special control character with the meaning "end of string" can be used as the last character in the string.
- 2) A separate memory word location or register can contain a number indicating the length of the string in byte

3.6 Memory Operations

Two memory operations are: 1) Load (Read/Fetch) & 2) Store (Write).

The Load operation transfers a copy of the contents of a specific memory-location to the processor. The memory contents remain unchanged.

- Steps for Load operation:
- 1) Processor sends the address of the desired location to the memory.
 - 2) Processor issues 'read' signal to memory to fetch the data.
 - 3) Memory reads the data stored at that address.
 - 4) Memory sends the read data to the processor.

The Store operation transfers the information from the register to the specified memory-location. This will destroy the original contents of that memory-location. Steps for Store operation is:

- 1) Processor sends the address of the memory-location where it wants to store data.
- 2) Processor issues 'write' signal to memory to store the data.
- 3) Content of register (MDR) is written into the specified memory-location.

3.7 Instruction and Instruction sequencing

A computer must have instructions capable of performing 4 types of operations:

- 1) Data transfers between the memory and the registers (MOV, PUSH, POP,XCHG).
- 2) Arithmetic and logic operations on data (ADD, SUB, MUL, DIV, AND, OR,NOT).
- 3) Program sequencing and control (CALL.RET, LOOP,INT).
- 4) I/O transfers (IN,OUT).

Register Transfer Notation (RTN)

The possible locations in which transfer of information occurs are: 1) Memory-location 2) Processor register & 3) Registers in I/O device.

Location	Hardware Binary Address	Example	Description
Memory	LOC, PLACE, NUM	$R1 \leftarrow [LOC]$	Contents of memory-location LOC are transferred into register R1.
Processor	R0, R1 ,R2	$[R3] \leftarrow [R1] + [R2]$	Add the contents of register R1 & R2 and places their sum into R3.
I/O Registers	DATAIN, DATAOUT	$R1 \leftarrow DATAIN$	Contents of I/O register DATAIN are transferred into register R1.

Assembly Language Notation

To represent machine instructions and programs, assembly language format is used.

Assembly Language Format	Description
Move LOC, R1	Transfer data from memory-location LOC to register R1. The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R1 are overwritten.
Add R1, R2, R3	Add the contents of registers R1 and R2, and places their sum into register R3.

Basic Instruction Types

Instruction Type	Syntax	Example	Description	Instructions for Operation
Three Address	Opcode Source1,Source2,Destination	Add A,B,C	Add the contents of memory-locations A & B. Then, place the result into location C.	
Two Address	Opcode Source, Destination	Add A,B	Add the contents of memory-locations A & B. Then, place the result into location B, replacing the original contents of this location. Operand B is both a source and a destination.	Move B, C Add A, C
One Address	Opcode Source/Destination	Load A	Copy contents of memory-location A into accumulator.	Load A Add B Store C
		Add B	Add contents of memory-location B to contents of accumulator register & place sum back into accumulator.	
		Store C	Copy the contents of the accumulator into location C.	
Zero Address	Opcode [no Source/Destination]	Push	Locations of all operands are defined implicitly. The operands are stored in a pushdown stack.	Not possible

Access to data in the registers is much faster than to data stored in memory-locations.

Let Ri represent a general-purpose register. The instructions: Load A,Ri

Store Ri,A

Add A,Ri

are generalizations of the Load, Store and Add Instructions for the single-accumulator case, in which register Ri performs the function of the accumulator. In processors, where arithmetic operations as allowed only on operands that are in registers, the task $C \leftarrow [A] + [B]$ can be performed by the instruction sequence:

Move A,Ri

Move B,Rj

Add Ri,Rj

Move Rj,C

Instruction Execution & Straight-Line Sequencing

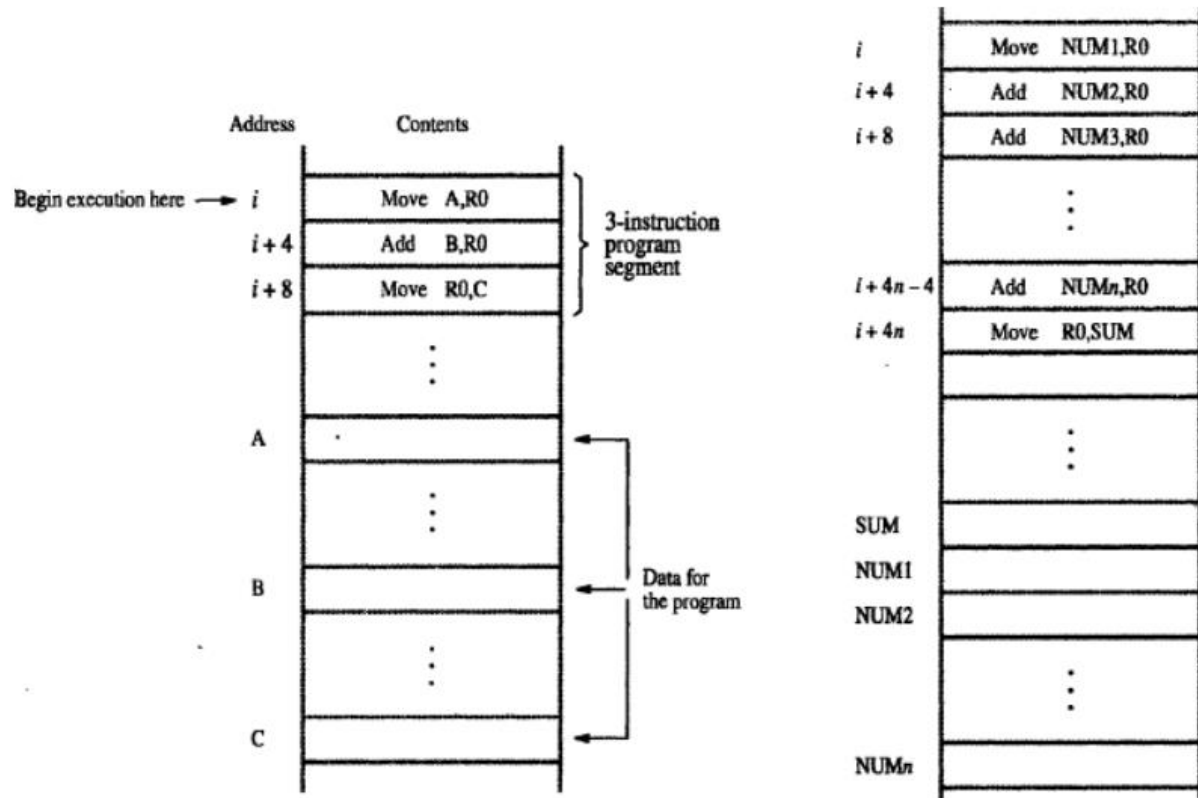
The program is executed as follows: 1) Initially, the address of the first instruction is loaded into PC.

2) Then, the processor control circuits use the information in the PC to fetch and execute instructions, one at a time, in the order of increasing addresses. This is called Straight-Line sequencing.

3) During the execution of each instruction, PC is incremented by 4 to point to next instruction.

There are 2 phases for Instruction Execution:

- 1) Fetch Phase: The instruction is fetched from the memory-location and placed in the IR.
- 2) Execute Phase: The contents of IR are examined to determine which operation is to be performed. The specified operation is then performed by the processor.



Program Explanation

Consider the program for adding a list of n numbers.

The Address of the memory-locations containing the n numbers are symbolically given as NUM1, NUM2.....NUMn. Separate Add instruction is used to add each number to the contents of register R0. After all the numbers have been added, the result is placed in memory-location SUM.

Branching

Consider the task of adding a list of ' n ' numbers. Number of entries in the list ' n ' is stored in memory-location N. Register R1 is used as a counter to determine the number of times the loop is executed. Content-location N is loaded into register R1 at the beginning of the program. The Loop is a straight-line sequence of instructions executed as many times as needed. The loop starts at location LOOP and ends at the instruction Branch>0.

During each pass, address of the next list entry is determined, and that entry is fetched and added to R0. The instruction Decrement R1 reduces the contents of R1 by 1 each time through the loop. Then Branch Instruction loads a new value into the program counter. As a result, the processor fetches and executes the instruction at this new address called the Branch Target. A Conditional Branch Instruction causes a branch only if a specified condition is satisfied. If the condition is not satisfied, the PC is incremented in the normal way, and the next instruction in sequential address order is fetched and executed.

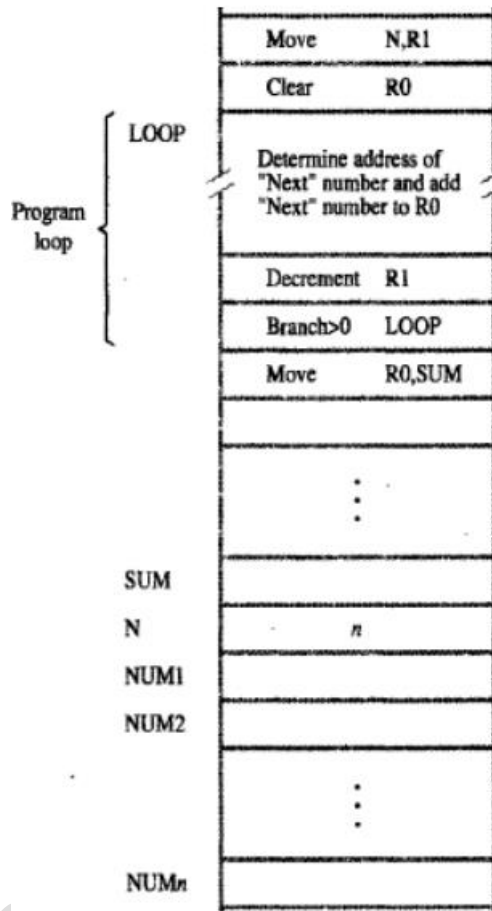


Fig: Loop to add n numbers.

Condition Codes

The processor keeps track of information about the results of various operations. This is accomplished by recording the required information in individual bits, called condition code flags. These flags are grouped together in a special processor-register called the condition code register (or statue register). Four commonly used flags are:

- 1) N (negative) set to 1 if the result is negative, otherwise cleared to 0.
- 2) Z (zero) set to 1 if the result is 0; otherwise, cleared to 0.
- 3) V (overflow) set to 1 if arithmetic overflow occurs; otherwise, cleared to 0.
- 4) C (carry) set to 1 if a carry-out results from the operation; otherwise cleared to 0.

3.8 Addressing Modes

The different ways in which the location of an operand is specified in an instruction are referred to as Addressing Modes.

Table 2.1 Generic addressing modes

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R _i	EA = R _i
Absolute (Direct)	LOC	EA = LOC
Indirect	(R _i)	EA = [R _i]
	(LOC)	EA = [LOC]
Index	X(R _i)	EA = [R _i] + X
Base with index	(R _i , R _j)	EA = [R _i] + [R _j]
Base with index and offset	X(R _i , R _j)	EA = [R _i] + [R _j] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(R _i) ⁺	EA = [R _i]; Increment R _i
Autodecrement	-(R _i)	Decrement R _i ; EA = [R _i]

EA = effective address
Value = a signed number

Implementation of Variable and Constants

A variable is represented by allocating a memory-location to hold its value. Thus, the value can be changed as needed using appropriate instructions. There are 2 accessing modes to access the variables:

1. Register Mode: The operand is the contents of a register. The name (or address) of the register is given in the instruction. Registers are used as temporary storage locations where the data in a register are accessed. For example, the instruction Move R1, R2 ; Copy content of register R1 into register R2.
2. Absolute (Direct) Mode: The operand is in a memory-location. The address of memory-location is given explicitly in the instruction. The absolute mode can represent global variables in the program. For example, the instruction Move LOC, R2 ; Copy content of memory-location LOC into register R2.

Address and data constants can be represented in assembly language using Immediate mode.

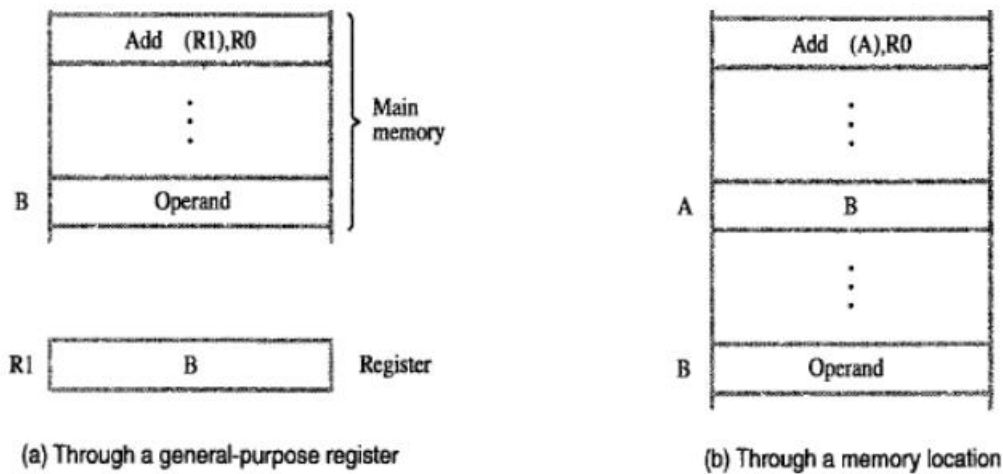
Immediate Mode: The operand is given explicitly in the instruction. For example, the instruction Move #200, R0 ; Place the value 200 in register R0. Clearly, the immediate mode is only used to specify the value of a source-operand.

Indirection And Pointers

Instruction does not give the operand or its address explicitly. Instead, the instruction provides information from which the new address of the operand can be determined. This address is called Effective Address (EA) of the operand.

Indirect Mode: The EA of the operand is the contents of a register (or memory-location). The register (or memory-location) that contains the address of an operand is called a Pointer.

We denote the indirection by \rightarrow name of the register or \rightarrow new address given in the instruction. E.g.: Add (R1),R0; The operand is in memory. Register R1 gives the effective address (B) of the operand. The data is read from location B and added to contents of register R0.



To execute the Add instruction in fig (a), the processor uses the value which is in register R1, as the EA of the operand. It requests a read operation from the memory to read the contents of location B. The value read is the desired operand, which the processor adds to the contents of register R0. Indirect addressing through a memory-location is also possible as shown in fig (b). In this case, the processor first reads the contents of memory-location A, then requests a second read operation using the value B as an address to obtain the operand.

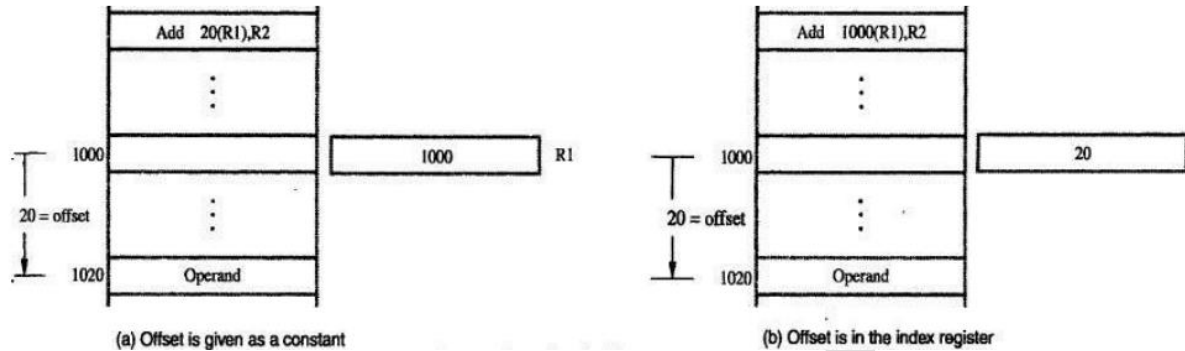
Address	Contents	
	Move	N,R1
	Move	#NUM1,R2
	Clear	R0
	} Initialization	
→ LOOP	Add	(R2),R0
	Add	#4,R2
	Decrement	R1
	Branch>0	LOOP
	Move	R0,SUM

Explanation: In above program, Register R2 is used as a pointer to the numbers in the list, and the operands are accessed indirectly through R2. The initialization-section of the program loads the counter-value n from memory-location N into R1 and uses the immediate addressing-mode to place the address value NUM1, which is the address of the first number in the list, into R2. Then it clears R0 to 0. The first two instructions in the loop implement the unspecified instruction block starting at LOOP. The first time through the loop, the instruction Add (R2), R0 fetches the operand at location NUM1 and adds it to R0. The second Add instruction adds 4 to the contents of the pointer R2, so that it will contain the address value NUM2 when the above instruction is executed in the second pass through the loop.

Indexing And Arrays

A different kind of flexibility for accessing operands is useful in dealing with lists and arrays. Index mode: the operation is indicated as X(Ri) where X=the constant value which defines an offset (also called a displacement). Ri=the name of the index registers which contains address of a new location. The effective address of the operand is given by $EA = X + [Ri]$. The contents of the index-register are not changed in the process of generating the effective- address. The

constant X may be given either → as an explicit number or → as a symbolic name representing a numerical value.



Fig(a) illustrates two ways of using the Index mode. In fig(a), the index register, R1, contains the address of a memory-location, and the value X defines an offset (also called a displacement) from this address to the location where the operand is found.

To find EA of operand: Eg: Add 20(R1), R2

$$EA \Rightarrow 1000 + 20 = 1020$$

An alternative use is illustrated in fig(b). Here, the constant X corresponds to a memory address, and the contents of the index register define the offset to the operand. In either case, the effective address is the sum of two values; one is given explicitly in the instruction, and the other is stored in a register.

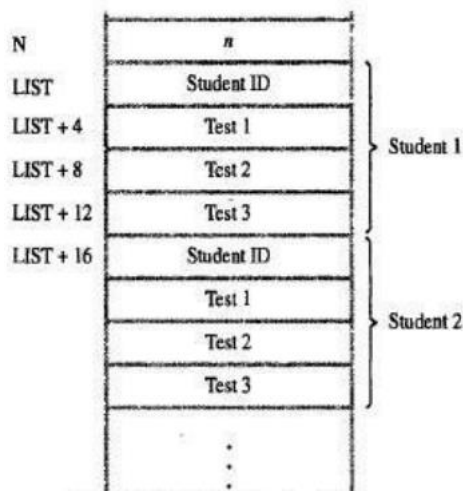


Figure 2.14 A list of students' marks.

	Move	#LIST,R0
	Clear	R1
	Clear	R2
	Clear	R3
	Move	N,R4
→	LOOP	Add 4(R0),R1
		Add 8(R0),R2
		Add 12(R0),R3
		Add #16,R0
	Decrement	R4
	Branch>0	LOOP
	Move	R1,SUM1
	Move	R2,SUM2
	Move	R3,SUM3

Figure 2.15 Indexed addressing used in accessing test scores in the list in Figure 2.14.

Base with Index Mode: Another version of the Index mode uses 2 registers which can be denoted as (Ri,Rj). Here, a second register may be used to contain the offset X. The second register is usually called the base register. The effective address of the operand is given by $EA = [Ri] + [Rj]$. This form of indexed addressing provides more flexibility in accessing operands because both components of the effective address can be changed. **Base with Index & Offset Mode.** Another version of the Index mode uses 2 registers plus a constant, which can be denoted as X(Ri,Rj). The effective address of the operand is given by $EA = X + [Ri] + [Rj]$. This added flexibility is useful in accessing multiple components inside each item in a record, where the beginning of an item is specified by the (Ri, Rj) part of the addressing-mode. In other words, this mode implements a 3-dimensional array.

Relative Mode: This is similar to index-mode with one difference: The effective-address is determined using the PC in place of the general-purpose register Ri. The operation is indicated as X(PC). X(PC) denotes an effective address of the operand which is X locations above or below the current contents of PC. Since the addressed location is identified "relative" to the PC, the name Relative mode is associated with this type of addressing. This mode is used commonly in conditional branch instructions. An instruction such as Branch >0 LOLOOP; Causes program execution to go to the branch target location identified by name LOOP if branch condition is satisfied.

Additional Addressing Modes

1) Auto Increment Mode: Effective address of operand is contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to point to the next item in a list. Implicitly, the increment amount is 1. This mode is denoted as (Ri)+; where Ri=pointer-register.

2) Auto Decrement Mode: The contents of a register specified in the instruction are first automatically decremented and are then used as the effective address of the operand. This mode is denoted as -(Ri) ; where Ri=pointer-register. These 2 modes can be used together to implement an important data structure called a stack.

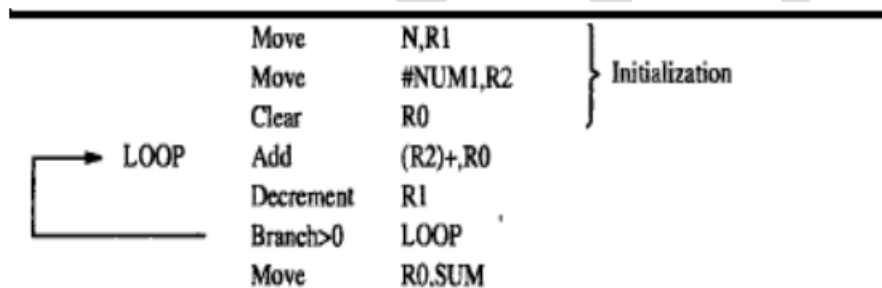


Figure 2.16 The Autoincrement addressing mode used in the program of Figure 2.12.

MODULE-4

Input/output Organization

Input/output architecture is an interface to the outside world that provide systematic means of controlling interaction with outside world. Input device such as keyboard, scanner, digital camera, etc. Output devices such as display, printer etc. Input/output devices cannot directly connect to the system bus because Data transfer rate of input/output devices is slower than (memory & processor) and verse versa. Input/output devices are used different data format and word length.

4.1 Accessing I/O Devices

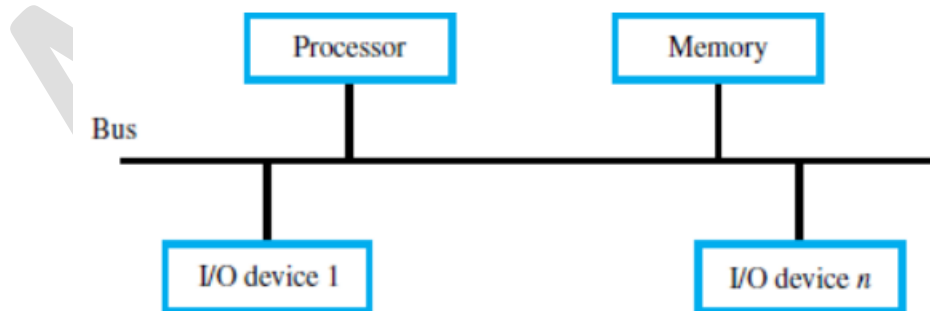
A single bus-structure can be used for connecting I/O-devices to a computer. Each I/O device is assigned a unique set of address. Bus consists of 3 sets of lines to carry address, data & control signals. Each IO devices are assigned with unique set of addresses. When processor places an address on address-lines, the intended-device responds to the command using control signal line. The processor requests either a read or write-operation. The requested-data are transferred over the data-lines.

There are 2 ways to deal with I/O-devices:

1) Memory-Mapped I/O: Memory and I/O-devices share a common address-space. Any data-transfer instruction (like Move, Load) can be used to exchange information.

For example, Move DATAIN, R0; This instruction sends the contents of location DATAIN to register R0. Here, DATAIN address of the input-buffer of the keyboard.

2) I/O-Mapped I/O — Memory and I/O address-spaces are different. A special instruction named IN and OUT are used for data-transfer. Ex: Processor in Intel family. Advantage of separate I/O space: I/O-devices deal with fewer address-lines.



Single bus structure

I/O Interface for an Input Device

1) Address Decoder: enables the device to recognize its address when this address appears on the address-lines.

2) Status Register: contains information relevant to operation of I/O-device.

3) Data Register: holds data being transferred to or from processor. There are 2 types:

i) DATAIN Input-buffer associated with keyboard.

ii) DATAOUT Output data buffer of a display/printer. Both data and status register are connected to the data bus and assigned unique address. The address decoder, the data and status register and the control circuitry required to coordinate I/O transfers constitute the device's interface circuit.

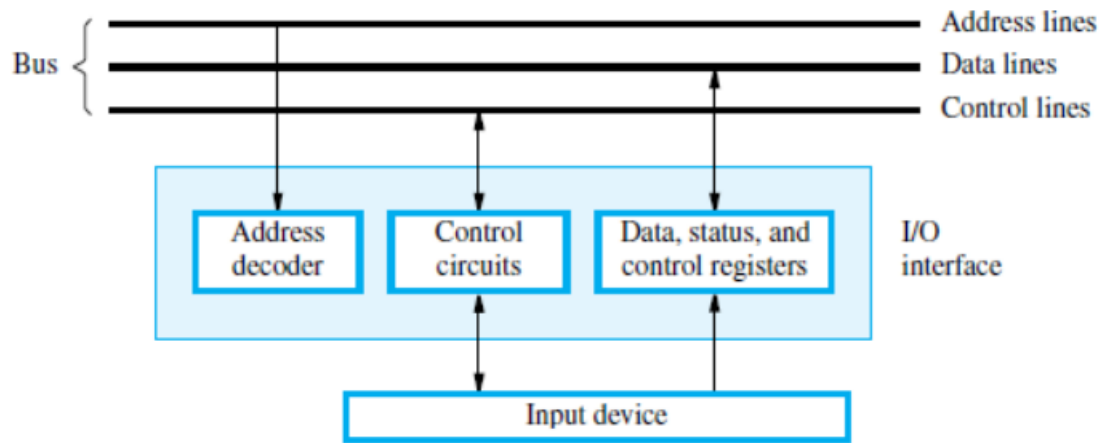


Figure 7.2 I/O interface for an input device.

Mechanisms Used for Interfacing I/O-Devices

- 1) Program Controlled I/O: Processor repeatedly checks status-flag to achieve required synchronization b/w processor & I/O device. Main drawback: The processor wastes time in checking status of device before actual data-transfer takes place.
- 2) Interrupt I/O: I/O-device initiates the action instead of the processor. I/O-device sends an INTR signal over bus whenever it is ready for a data-transfer operation. Like this, required synchronization is done between processor & I/O device.
- 3) Direct Memory Access (DMA): Device-interface transfer data directly to/from the memory w/o continuous involvement by the processor. DMA is a technique used for high speed I/O-device.

4.2 Interrupts

There are many situations where other tasks can be performed while waiting for an I/O device to become ready. A hardware signal called an Interrupt will alert the processor when an I/O device becomes ready. Interrupt-signal is sent on the interrupt-request line. The processor can be performing its own task without the need to continuously check the I/O-device. The routine executed in response to an interrupt-request is called ISR. The processor must inform the device that its request has been recognized by sending INTA signal. (INTR Interrupt Request, INTA Interrupt Acknowledge, ISR Interrupt Service Routine)

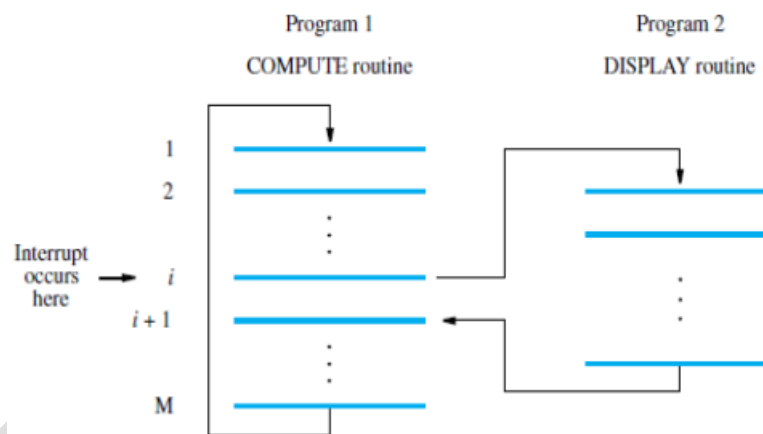
For example, consider COMPUTE and PRINT routines. Assume that an interrupt request arrives during execution of instruction *i* in Fig below.

The processor first completes the execution of instruction *i*. Then, processor loads the PC with the address of the first instruction of the ISR (interrupt service routine). After the execution of ISR, the processor must come back to instruction *i*+1. Therefore, when an interrupt occurs, the current content of PC is put in temporary storage location. A return at the end of ISR reloads

the PC from that temporary storage location. This causes the execution to resume at instruction_i+1.

When processor is handling interrupts, it must inform device that its request has been recognized. This may be accomplished by special control signal on bus. An interrupt-acknowledge signal is used. The task of saving and restoring the information can be done automatically by the processor or by program instruction. Saving and restoring registers involves memory transfers: Increases the total execution time. Increases the delay between the time an interrupt request is received, and the start of execution of the interrupt-service routine. This delay is called interrupt latency.

To reduce the interrupt latency, most processors save only the minimal amount of information. This minimal amount of information includes Program Counter and processor status registers. Any additional information that must be saved, must be saved explicitly by the program instructions at the beginning of the interrupt service routine. Interrupt Latency is a delay between time an interrupt-request is received and start of the execution of the ISR.



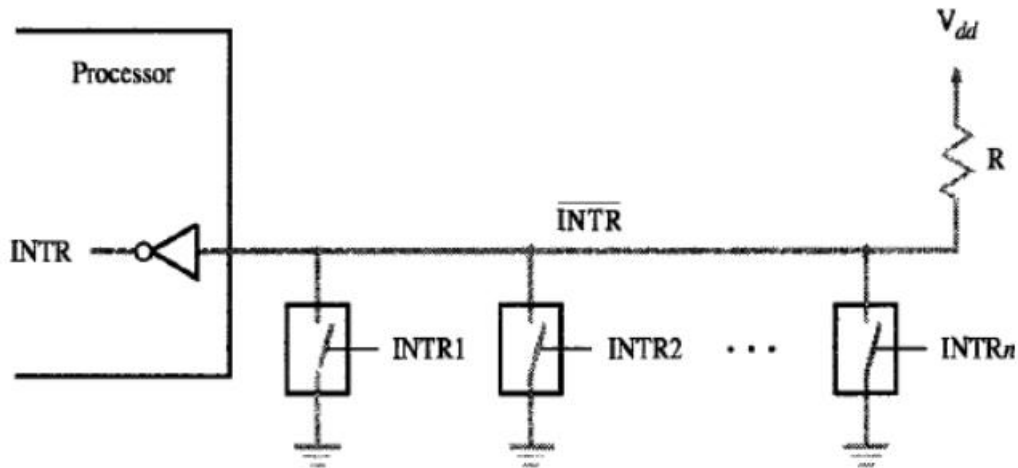
Transfer of control through the use of interrupts

Interrupt Hardware

Most computers have several I/O devices that can request an interrupt. A single interrupt-request (IR) line may be used to serve n devices. All devices are connected to IR line via switches to ground. To request an interrupt, a device closes its associated switch. Thus, if all IR signals are inactive, the voltage on the IR line will be equal to V_{dd}. When a device requests an interrupt, the voltage on the line drops to 0. This causes the INTR received by the processor to go to 1. The value of INTR is the logical OR of the requests from individual devices.

$$\text{INTR} = \text{INTR}_1 + \text{INTR}_2 + \dots + \text{INTR}_n$$

A special gate known as open-collector or open-drain are used to drive the INTR line. The Output of the open collector control is equal to a switch to the ground that is open when gates input is in "0" state and closed when the gates input is in "1" state. Resistor R is called a Pull-up Resistor because it pulls the line voltage up to the high-voltage state when the switches are open.



An equivalent circuit for an open-drain bus used to implement a common interrupt-request line.

4.3 Enabling and Disabling Interrupts

All computers fundamentally should be able to enable and disable interruptions as desired. Processors generally provide the ability to enable and disable such interruptions as desired. The problem of infinite loop occurs due to successive interruptions of active INTR signals. There are 3 mechanisms to solve problem of infinite loop:

- 1) Processor should ignore the interrupts until execution of first instruction of the ISR by the processor hardware. By using an Interrupt-disable instruction as the first instruction in the interrupt-service routine, the programmer can ensure that no further interruptions will occur until an Interrupt-enable instruction is executed. Typically, the Interrupt-enable instruction will be the last instruction in the interrupt-service routine before the Return from-interrupt instruction.
- 2) Processor should automatically disable interrupts before starting the execution of the ISR. The second option, which is suitable for a simple processor with only one interrupt request line, is to have the processor automatically disable interrupts before starting the execution of the interrupt-service routine. After saving the contents of the PC and the processor status register (PS) on the stack, the processor performs the equivalent of executing an Interrupt-disable instruction. It is often the case that one bit in the PS register, called Interrupt-enable, indicates whether interrupts are enabled.
- 3) In the third option, the processor has a special interrupt-request line for which the interrupt handling circuit responds only to the leading edge of the signal. Such a line is said to be edge triggered.

Sequence of events involved in handling an interrupt-request:

- 1) The device raises an interrupt-request.
- 2) The processor interrupts the program currently being executed.
- 3) Interrupts are disabled by changing the control bits in the processor status register (PS).
- 4) The device is informed that its request has been recognized. In response, the device deactivates the interrupt-request signal.
- 5) The action requested by the interrupt is performed by the interrupt-service routine.

6) Interrupts are enabled, and execution of the interrupted program is resumed.

4.4 Handling Multiple Devices

While handling multiple devices, the issues concerned are:

- 1) How can the processor recognize the device requesting an interrupt?
- 2) How can the processor obtain the starting address of the appropriate ISR?
- 3) Should a device be allowed to interrupt the processor while another interrupt is being serviced?
- 4) How should 2 or more simultaneous interrupt-requests be handled?

When a request is received over the common interrupt-request line, additional information is needed to identify the device that activated the line. The information needed to determine whether a device is requesting an interrupt is available in its status register. When a device raises an interrupt request, it sets to 1 one of the bits in its status register, which we will call the IRQ bit. The two methods to handle multiple devices are: Polling and Vectored Interrupts

Polling

Information needed to determine whether device is requesting interrupt is available in status-register. Following condition-codes are used:

DIRQ Interrupt-request for display.

KIRQ Interrupt-request for keyboard.

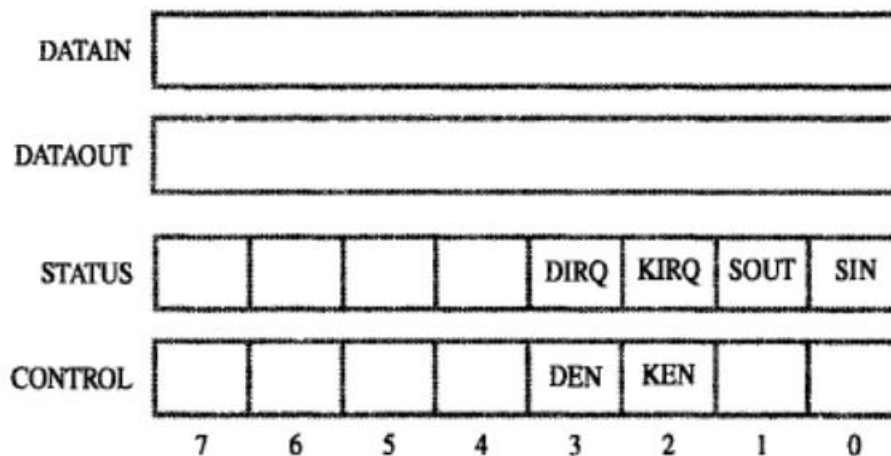
KEN keyboard enable.

DEN Display Enable.

SIN, SOUT status flags.

For an input device, SIN status flag is used. SIN = 1 when a character is entered at the keyboard. SIN = 0 when the character is read by processor. IRQ=1 when a device raises an interrupt-requests. Simplest way to identify interrupting-device is to have ISR poll all devices connected to bus. The first device encountered with its IRQ bit set is serviced. After servicing first device, next requests may be serviced. Advantage: Simple & easy to implement.

Disadvantage: More time spent polling IRQ bits of all devices.



Registers in keyboard and display interfaces

Vectored Interrupts

A device requesting an interrupt identifies itself by sending a special code to processor over bus. Then, the processor starts executing the ISR. The special-code indicates starting-address of ISR. The special-code length ranges from 4 to 8 bits. The location pointed to by the interrupting-device is used to store the starting address to ISR. The starting address to ISR is called the interrupt vector. Processor loads interrupt-vector into PC & executes appropriate ISR. When processor is ready to receive interrupt-vector code, it activates INTA line. Then, I/O-device responds by sending its interrupt-vector code & turning off the INTR signal. The interrupt vector also includes a new value for the Processor Status Register.

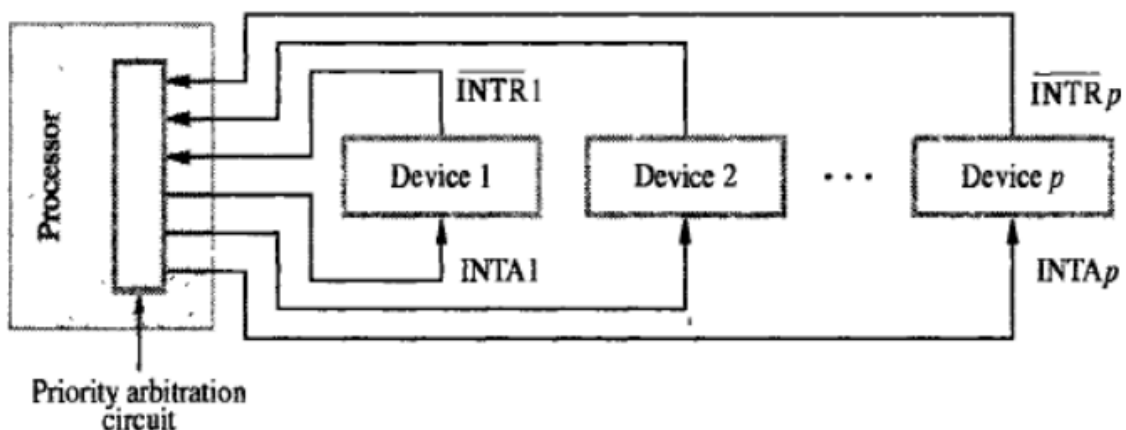
Interrupt Nesting

Interrupts should be disabled during the execution of an interrupt-service routine, to ensure that a request from one device will not cause more than one interruption. The same arrangement is often used when several devices are involved, in which case execution of a given interrupt-service routine, once started, always continues to completion before the processor accepts an interrupt request from a second device.

Interrupt-service routines are typically short, and the delay they may cause is acceptable for most simple devices. A multiple-priority scheme is implemented by using separate INTR & INTA lines for each device. Each INTR line is assigned a different priority-level. Priority-level of processor is the priority of program that is currently being executed. Processor accepts interrupts only from devices that have higher priority than its own. At the time of execution of ISR for some device, priority of processor is raised to that of the device. Thus, interrupts from devices at the same level of priority or lower are disabled.

Privileged Instruction: Processor's priority is encoded in a few bits of PS word. (PS Processor-Status). Encoded-bits can be changed by Privileged Instructions that write into PS. Privileged-instructions can be executed only while processor is running in Supervisor Mode. Processor is in supervisor-mode only when executing operating-system routines.

Privileged Exception: User program cannot accidentally or intentionally change the priority of the processor & disrupt the system-operation. An attempt to execute a privileged-instruction while in user-mode leads to a Privileged Exception.



Implementation of interrupt priority using individual interrupt-request and acknowledge lines

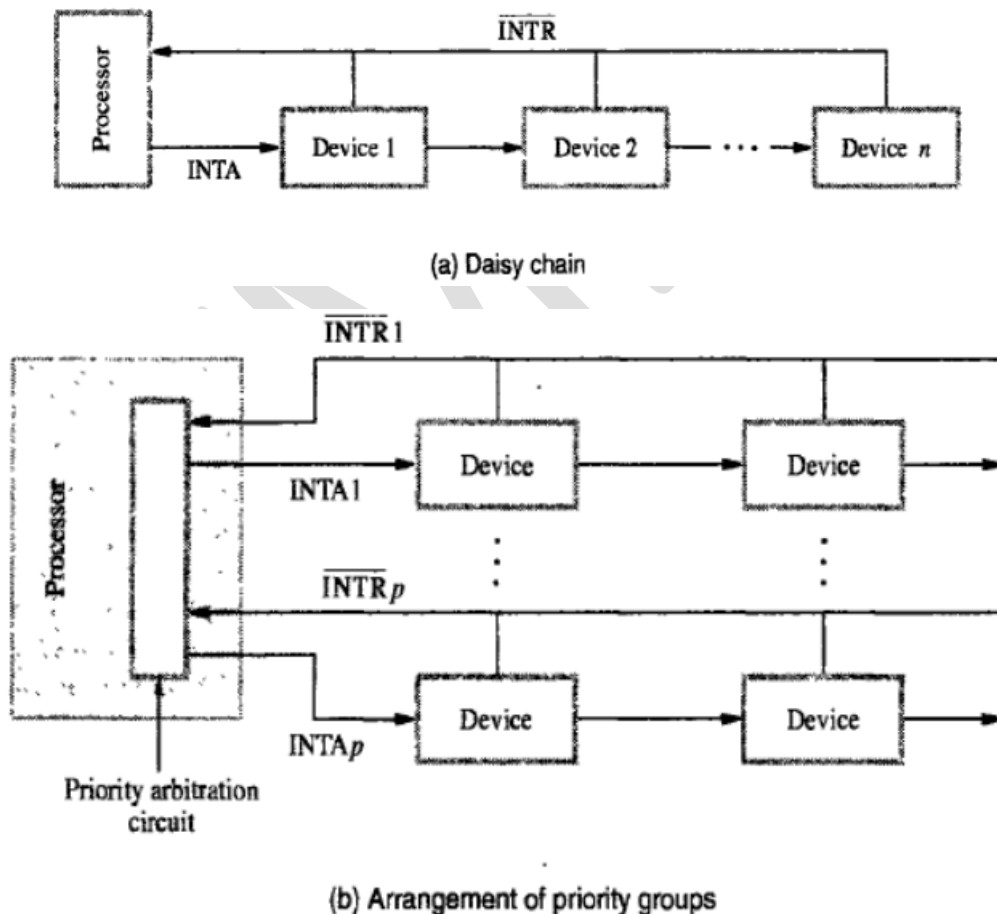
Simultaneous Requests

The processor must have some mechanisms to decide which request to service when simultaneous requests arrive. A widely used scheme is to connect the devices to form a daisy chain, as shown in fig. a. INTR line is common to all devices. The interrupt-acknowledge line, INTA, is connected in a daisy-chain fashion, such that the INTA signal propagates serially through the devices. INTA signal propagates serially through devices.

When several devices raise an interrupt-request, INTR line is activated. Processor responds by setting INTA line to 1. This signal is received by device 1. Device-1 passes signal on to device 2 only if it does not require any service. If device-1 has a pending-request for interrupt, the device-1 blocks INTA signal & proceeds to put its identifying-code on data-lines. Device that is electrically closest to processor has highest priority.

Advantage: It requires fewer wires than the individual connections.

Arrangement of Priority Groups: Here, the devices are organized in groups & each group is connected at a different priority level. Within a group, devices are connected in a daisy chain.

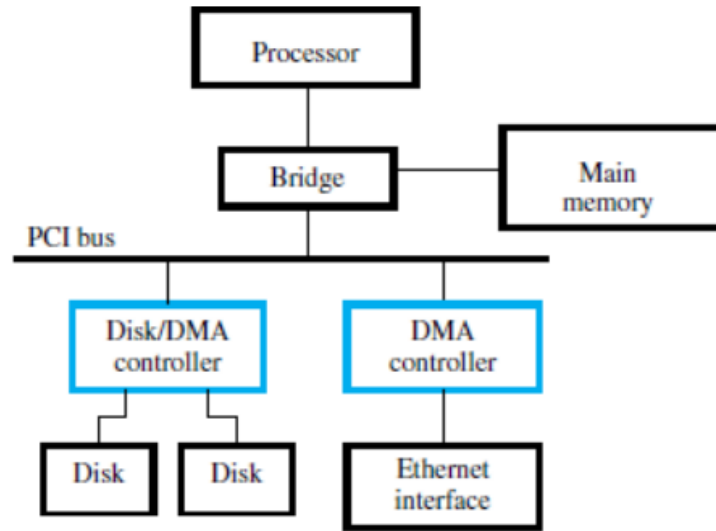


Interrupt priority schemes

4.5 Direct Memory Access

The transfer of a block of data directly b/w an external device & main-memory w/o continuous involvement by processor is called DMA. DMA controller is a control circuit that performs DMA transfers. it is a part of the I/O device interface, performs the functions that would

normally be carried out by processor. While a DMA transfer is taking place, the processor can be used to execute another program.



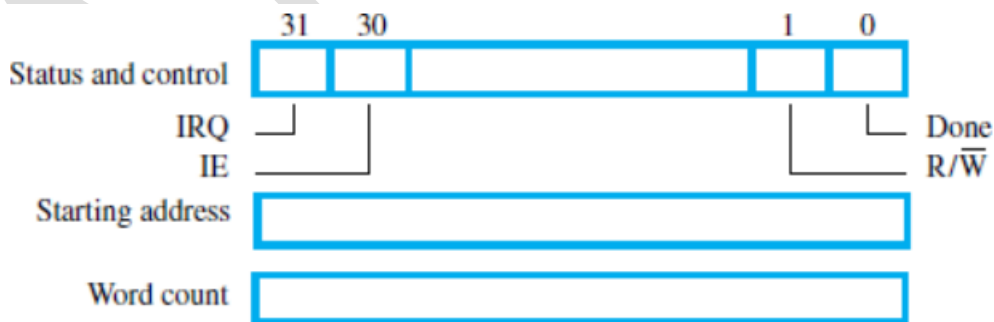
Use of DMA controllers in a computer system

DMA interface has three registers: 1) First register is used for storing starting address.

2) Second register is used for storing word-count.

3) Third register contains status- & control-flags.

The R/W bit determines direction of transfer. If R/W=1, controller performs a read-operation (i.e., it transfers data from memory to I/O), Otherwise, controller performs a write-operation (i.e., it transfers data from I/O to memory). If Done=1, the controller has completed transferring a block of data and is ready to receive another command. (IE Interrupt Enable). If IE=1, controller raises an interrupt after it has completed transferring a block of data. If IRQ=1, controller requests an interrupt. Requests by DMA devices for using the bus are always given higher priority than processor requests.



Typical registers in a DMA controller

There are 2 ways in which the DMA operation can be carried out:

1) Processor originates most memory-access cycles. DMA controller is said to "steal" memory cycles from processor. Hence, this technique is usually called Cycle Stealing.

2) DMA controller is given exclusive access to main memory to transfer a block of data without any interruption. This is known as Block Mode (or burst mode).

Bus Arbitration

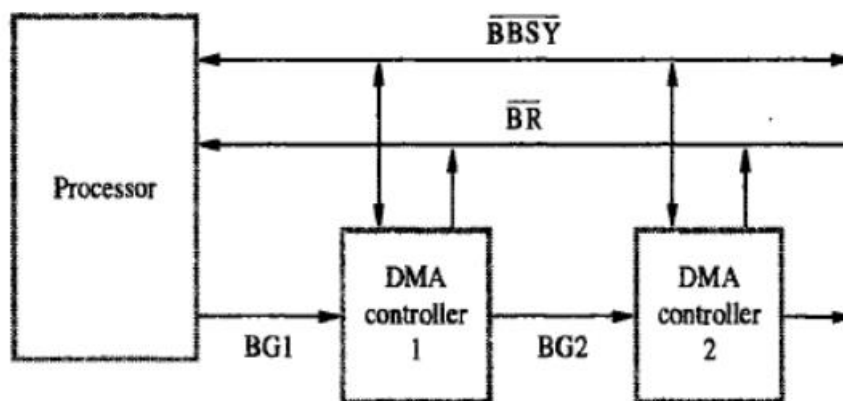
The device that is allowed to initiate data-transfers on bus at any given time is called bus-master. There can be only one bus-master at any given time. Bus Arbitration is the process by which → next device to become the bus-master is selected & bus-mastership is transferred to that device.

The two approaches are: 1) Centralized Arbitration: A single bus-arbiter performs the required arbitration. 2) Distributed Arbitration: All devices participate in selection of next bus-master.

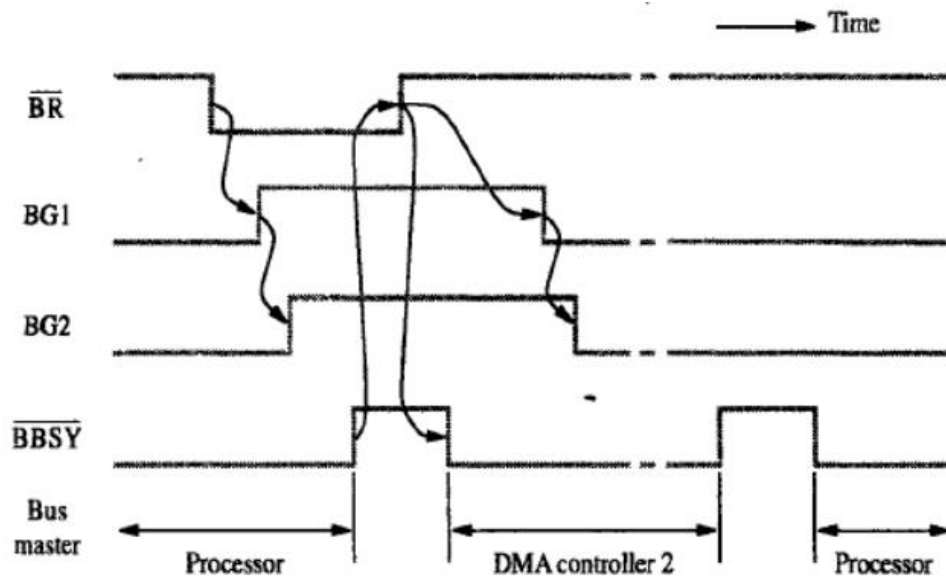
A conflict may arise if both the processor and a DMA controller or two DMA controllers try to use the bus at the same time to access the main memory. To resolve this, an arbitration procedure is implemented on the bus to coordinate the activities of all devices requesting memory transfers. The bus arbiter may be the processor, or a separate unit connected to the bus.

Centralized Arbitration

A single bus-arbiter performs the required arbitration. Normally, processor is the bus-master. Processor may grant bus-mastership to one of the DMA controllers. A DMA controller indicates that it needs to become bus-master by activating BR line. The signal on the BR line is the logical OR of bus-requests from all devices connected to it. Then, processor activates BG1 signal indicating to DMA controllers to use bus when it becomes free. BG1 signal is connected to all DMA controllers using a daisy-chain arrangement. If DMA controller-1 is requesting the bus, Then, DMA controller-1 blocks propagation of grant-signal to other devices. Otherwise, DMA controller-1 passes the grant downstream by asserting BG2. Current bus-master indicates to all devices that it is using bus by activating BBSY line. The bus-arbiter is used to coordinate the activities of all devices requesting memory transfers. Arbiter ensures that only 1 request is granted at any given time according to a priority scheme. (BR Bus-Request, BG Bus-Grant, BBSY Bus Busy). The timing diagram shows the sequence of events for the devices connected to the processor. DMAcontroller-2 requests and acquires bus-mastership and later releases the bus. After DMA controller-2 releases the bus, the processor resumes bus-mastership.



A simple arrangement for bus arbitration using a daisy chain

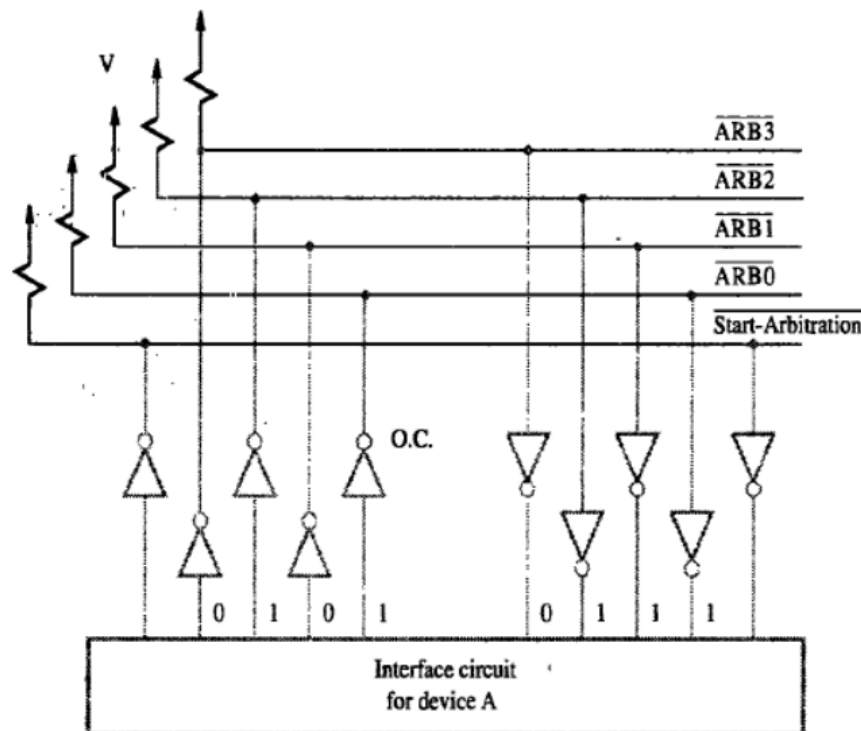


Sequence of signals during transfer of bus mastership for the devices in previous figure.

Distributed Arbitration

All devices participate in the selection of next bus-master. Each device on bus is assigned a 4-bit identification number (ID). When 1 or more devices request bus, they assert Start-Arbitration signal & place their 4-bit ID numbers on four open-collector lines ARB 0 through ARB 3. A winner is selected as a result of interaction among signals transmitted over these lines. Net-outcome is that the code on 4 lines represents request that has the highest ID number.

Advantage: This approach offers higher reliability since operation of bus is not dependent on any single device.



Distributed arbitration scheme

For E.g.: Assume 2 devices A & B have their ID 5 (0101), 6 (0110) and their code is 0111. Each device compares the pattern on the arbitration line to its own ID starting from MSB. If the device detects a difference at any bit position, it disables the drivers at that bit position. Driver is disabled by placing "0" at the input of the driver.

In e.g. "A" detects a difference in line ARB1, hence it disables the drivers on lines ARB1 & ARB0. This causes pattern on arbitration-line to change to 0110. This means that "B" has won contention.

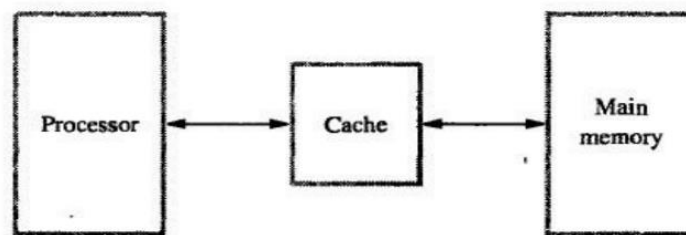
4.6 Cache Memories

Processor is much faster than the main memory. As a result, the processor has to spend much of its time waiting while instructions and data are being fetched from the main memory. This serves as a major obstacle towards achieving good performance. Speed of the main memory cannot be increased beyond a certain point. So, we use Cache memories. Cache memory is an architectural arrangement which makes the main memory appear faster to the processor than it really is. Cache memory is based on the property of computer programs known as "locality of reference".

Analysis of programs indicates that many instructions in localized areas of a program are executed repeatedly during some period of time, while the others are accessed relatively less frequently. These instructions may be the ones in a loop, nested loop or few procedures calling each other repeatedly. This is called "locality of reference". Its types are:

Temporal locality of reference: Recently executed instruction is likely to be executed again very soon.

Spatial locality of reference: Instructions with addresses close to a recently instruction are likely to be executed soon.



Use of a cache memory

A simple arrangement of cache memory is as shown above.

- Processor issues a Read request, a block of words is transferred from the main memory to the cache, one word at a time.
- Subsequent references to the data in this block of words are found in the cache.
- At any given time, only some blocks in the main memory are held in the cache. Which blocks in the main memory are in the cache is determined by a "mapping function".
- When the cache is full, and a block of words needs to be transferred from the main memory, some block of words in the cache must be replaced. This is determined by a "replacement algorithm".

Cache hit: Existence of a cache is transparent to the processor. The processor issues Read and Write requests in the same manner. If the data is in the cache it is called a Read or Write hit.

- **Read hit:** The data is obtained from the cache.
- **Write hit:** Cache has a replica of the contents of the main memory. Contents of the cache and the main memory may be updated simultaneously. This is the write-through protocol.

Update the contents of the cache, and mark it as updated by setting a bit known as the dirty bit or modified bit. The contents of the main memory are updated when this block is replaced. This is write-back or copy-back protocol.

Cache miss: If the data is not present in the cache, then a Read miss or Write miss occurs.

- **Read miss:** Block of words containing this requested word is transferred from the memory. After the block is transferred, the desired word is forwarded to the processor. The desired word may also be forwarded to the processor as soon as it is transferred without waiting for the entire block to be transferred. This is called load-through or early-restart.
- **Write-miss:** Write-through protocol is used, then the contents of the main memory are updated directly. If write-back protocol is used, the block containing the addressed word is first brought into the cache. The desired word is overwritten with new information.

Cache Coherence Problem

A bit called as “valid bit” is provided for each block. If the block contains valid data, then the bit is set to 1, else it is 0. Valid bits are set to 0, when the power is just turned on.

When a block is loaded into the cache for the first time, the valid bit is set to 1. Data transfers between main memory and disk occur directly bypassing the cache. When the data on a disk changes, the main memory block is also updated. However, if the data is also resident in the cache, then the valid bit is set to 0.

The copies of the data in the cache, and the main memory are different. This is called the cache coherence problem

Mapping functions: Mapping functions determine how memory blocks are placed in the cache.

A simple processor example:

- Cache consisting of 128 blocks of 16 words each.
- Total size of cache is 2048 (2K) words.
- Main memory is addressable by a 16-bit address.
- Main memory has 64K words.
- Main memory has 4K blocks of 16 words each.

Three mapping functions can be used.

1. Direct mapping
2. Associative mapping
3. Set-associative mapping.

MODULE-5

Chapter 1: Basic Processing Unit

5.1 Some Fundamental Concepts

The processing unit which executes machine instructions and coordinates the activities of other units of computer is called the Instruction Set Processor (ISP) or processor or Central Processing Unit (CPU). The primary function of a processor is to execute the instructions stored in memory. Instructions are fetched from successive memory locations and executed in processor, until a branch instruction occurs.

To execute an instruction, processor has to perform following 3 steps:

1. Fetch contents of memory-location pointed to by PC. Content of this location is an instruction to be executed. The instructions are loaded into IR, Symbolically, this operation is written as: $IR \leftarrow [[PC]]$
2. Increment PC by 4. $PC \leftarrow [PC] + 4$
3. Carry out the actions specified by instruction (in the IR). The steps 1 and 2 are referred to as Fetch Phase. Step 3 is referred to as Execution Phase.

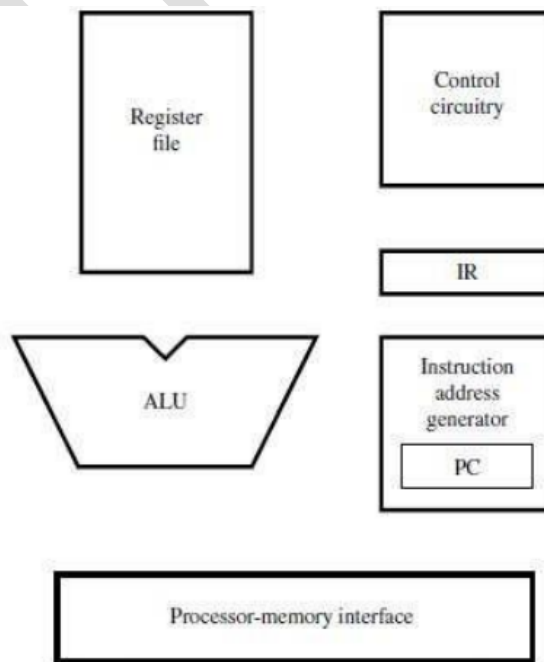
The operation specified by an instruction can be carried out by performing one or more of the following actions: 1) Read the contents of a given memory-location and load them into a register.

2) Read data from registers or memory location.

3) Perform an arithmetic or logic operation and place the result into a register.

4) Store data from a register into a given memory-location.

The hardware-components needed to perform these actions are shown in Figure.



Main hardware components of a processor

Single Bus Organization

Here the processor contains only a single bus for the movement of data, address and instructions. ALU and all the registers are interconnected via a Single Common Bus. Data & address lines of the external memory-bus is connected to the internal processor-bus via MDR & MAR respectively. (MDR - Memory Data Register, MAR - Memory Address Register).

MDR has 2 inputs and 2 outputs. Data may be loaded into MDR either from memory-bus (external) or from processor-bus (internal).

MAR's input is connected to internal bus; MAR's output is connected to external bus. (address sent from processor to memory only)

Instruction Decoder & Control Unit is responsible for Decoding the instruction and issuing the control-signals to all the units inside the processor, implementing the actions specified by the instruction (loaded in the IR).

Processor Registers - Register R0 through R(n-1) are also called as General-Purpose Register. The programmer can access these registers for general-purpose use.

Temporary Registers – There are 3 temporary registers in the processor. Registers - Y, Z & Temp are used for temporary storage during program-execution. The programmer cannot access these 3 registers.

In ALU,

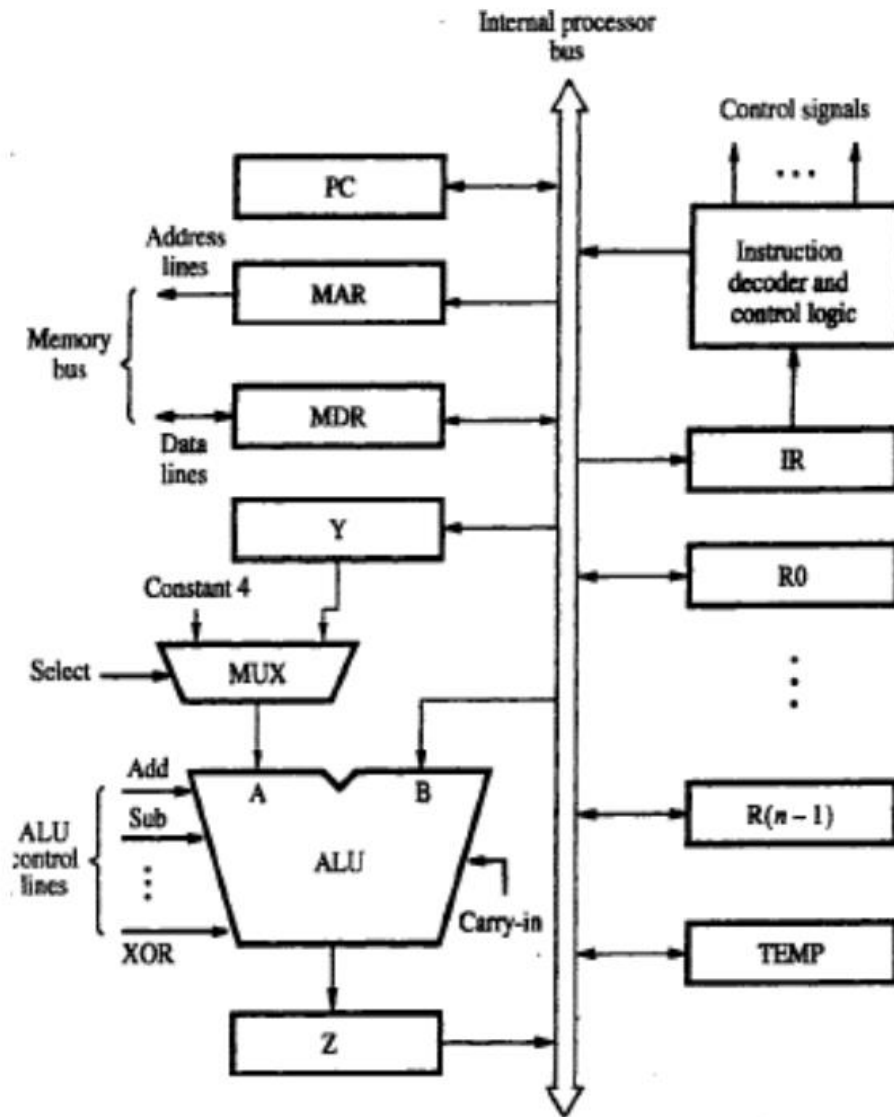
- 1) "A" input gets the operand from the output of the multiplexer (MUX).
- 2) "B" input gets the operand directly from the processor-bus. There are 2 options provided for "A" input of the ALU. MUX is used to select one of the 2 inputs. MUX selects either output of Y or constant-value 4 (which is used to increment PC content).

An instruction is executed by performing one or more of the following operations:

- 1) Transfer a word of data from one register to another or to the ALU.
- 2) Perform arithmetic or a logic operation and store the result in a register.
- 3) Fetch the contents of a given memory-location and load them into a register.
- 4) Store a word of data from a register into a given memory-location.

Disadvantage: Only one data-word can be transferred over the bus in a clock cycle.

Solution: Provide multiple internal paths. Multiple paths allow several data transfers to take place in parallel.



Single bus organization of the datapath inside a processor

5.2 Register Transfers

Instruction execution involves a sequence of steps in which data are transferred from one register to another. For each register, two control-signals are used: $R_{i\text{in}}$ & $R_{i\text{out}}$. These are called Gating Signals. $R_{i\text{in}}=1$ data on bus is loaded into R_i . $R_{i\text{out}}=1$ content of R_i is placed on bus. $R_{i\text{out}}=0$, bus can be used for transferring data from other registers.

For example, Move R_1, R_2 ; This transfers the contents of register R_1 to register R_2 . This can be accomplished as follows:

- 1) Enable the output of registers R_1 by setting $R_{1\text{out}}$ to 1. This place the contents of R_1 on processor-bus.
- 2) Enable the input of register R_2 by setting $R_{2\text{in}}$ to 1.

This loads data from processor-bus into register R_2 . All operations and data transfers within the processor take place within time periods defined by the processor-clock. The control-signals that govern a particular transfer are asserted at the start of the clock cycle.

Input & Output Gating for one Register Bit: A 2-input multiplexer is used to select the data applied to the input of an edge triggered D flip-flop. $R_{in}=1$ mux selects data on bus. This data will be loaded into flip-flop at rising-edge of clock. $R_{in}=0$ mux feeds back the value currently stored in flip-flop (Figure 7.3). Q output of flip-flop is connected to bus via a tri-state gate. $R_{out}=0$ gate's output is in the high-impedance state. $R_{out}=1$ the gate drives the bus to 0 or 1, depending on the value of Q.

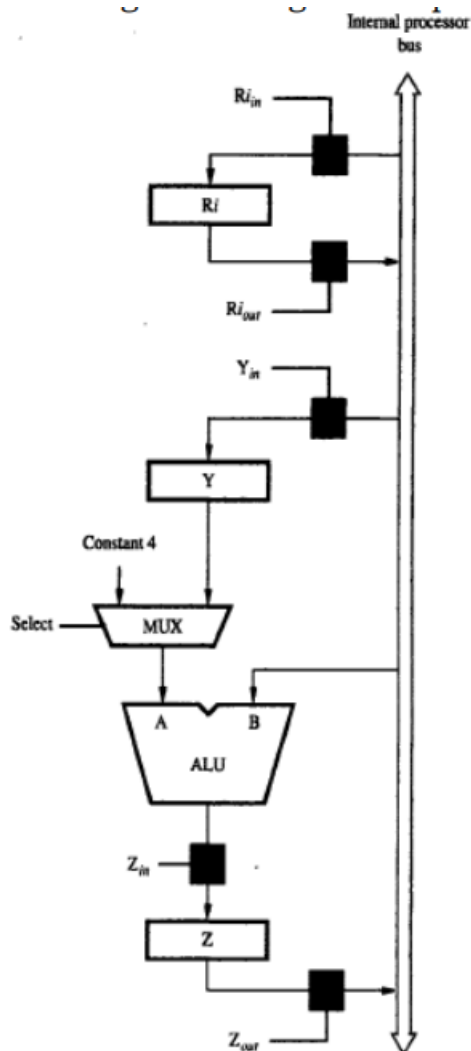


Figure 7.2 Input and output gating for the registers in Figure 7.1.

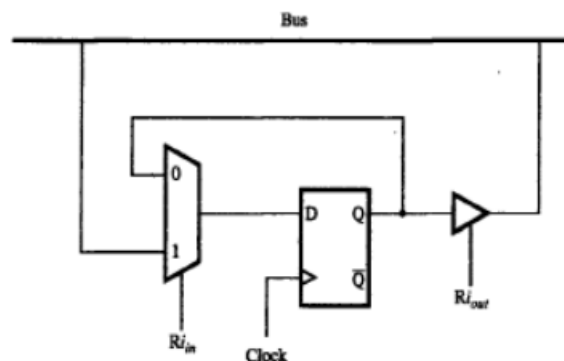


Figure 7.3 Input and output gating for one register bit.

5.3 Performing ALU operations

The ALU performs arithmetic operations on the 2 operands applied to its A and B inputs. One of the operands is output of MUX; And, the other operand is obtained directly from processor-bus. The result (produced by the ALU) is stored temporarily in register Z.

The sequence of operations for $[R3] [R1] + [R2]$ is as follows: 1) $R1_{out}$, Y_{in} 2) $R2_{out}$, $SelectY$, Add , Z_{in} 3) Z_{out} , $R3_{in}$

Instruction execution proceeds as follows:

Step 1 --> Contents from register R1 are loaded into register Y.

Step2 --> Contents from Y and from register R2 are applied to the A and B inputs of ALU; Addition is performed & Result is stored in the Z register.

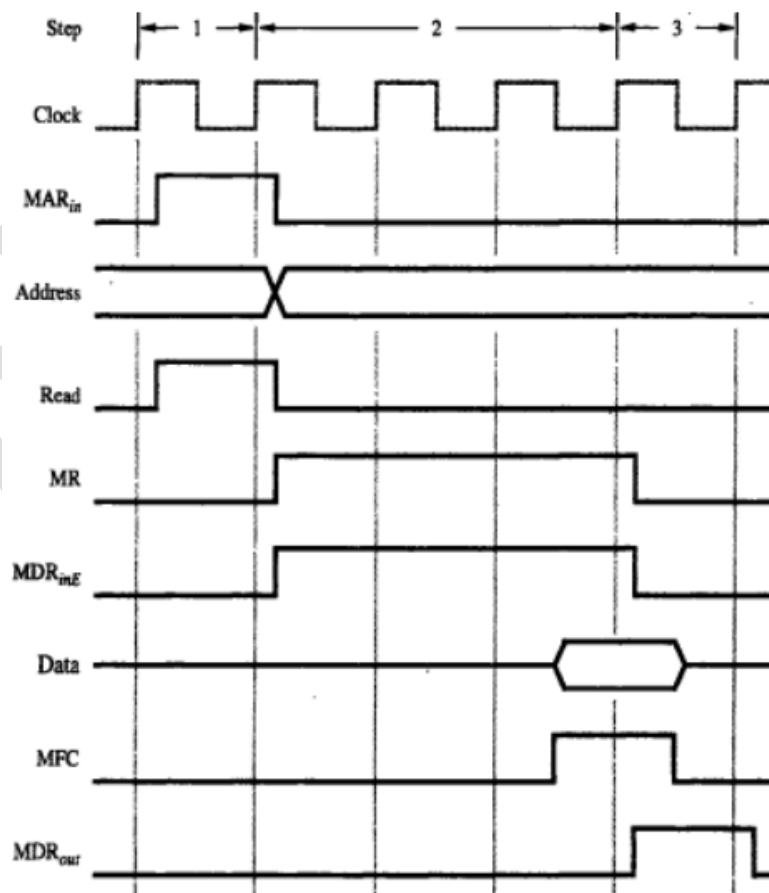
Step 3 --> The contents of Z register is stored in the R3 register. The signals are activated for the duration of the clock cycle corresponding to that step. All other signals are inactive.

5.4 Fetching a word from Memory

To fetch instruction/data from memory, processor transfers required address to MAR. At the same time, processor issues Read signal on control-lines of memory-bus. When requested data are received from memory, they are stored in MDR. From MDR, they are transferred to other registers. The response time of each memory access varies (based on cache miss, memory mapped I/O).

To accommodate this, MFC is used. (MFC Memory Function Completed). MFC is a signal sent from addressed device to the processor. MFC informs the processor that the requested operation has been completed by addressed device. Consider the instruction Move (R1),R2. The sequence of steps is:

- 1) R1out, MARin, Read; desired address is loaded into MAR & Read command is issued.
- 2) MDRinE, WMFC; load MDR from memory-bus & Wait for MFC response from memory.
- 3) MDRout, R2in; load R2 from MDR. where WMFC=control-signal that causes processor's control. circuitry to wait for arrival of MFC signal.



Timing of a memory read operation

5.5 Storing a word in memory

Consider the instruction Move R2, (R1). This requires the following sequence:

- 1) R1_{out}, MAR_{in}; desired address is loaded into MAR.
- 2) R2_{out}, MDR_{in}, Write ;data to be written are loaded into MDR & Write command is issued.
- 3) MDR_{out}E, WMFC ;load data into memory-location pointed by R1 from MDR.

5.6 Execution of a Complete Instruction

Consider the instruction Add (R3), R1 which adds the contents of a memory-location pointed by R3 to register R1. Executing this instruction requires the following actions:

- 1) Fetch the instruction.
- 2) Fetch the first operand.
- 3) Perform the addition &
- 4) Load the result into R1.

Step	Action
1	PC _{out} , MAR _{in} , Read, Select4, Add, Z _{in}
2	Z _{out} , PC _{in} , Y _{in} , WMFC
3	MDR _{out} , IR _{in}
4	R3 _{out} , MAR _{in} , Read
5	R1 _{out} , Y _{in} , WMFC
6	MDR _{out} , SelectY, Add, Z _{in}
7	Z _{out} , R1 _{in} , End

Figure 7.6 Control sequence for execution of the instruction Add (R3),R1

Instruction execution proceeds as follows:

Step1--> The instruction-fetch operation is initiated by loading contents of PC into MAR & sending a Read request to memory. The Select signal is set to Select4, which causes the Mux to select constant 4. This value is added to operand at input B (PC's content), and the result is stored in Z.

Step2--> Updated value in Z is moved to PC. This completes the PC increment operation and PC will now point to next instruction.

Step3--> Fetched instruction is moved into MDR and then to IR. The step 1 through 3 constitutes the Fetch Phase. At the beginning of step 4, the instruction decoder interprets the contents of the IR. This enables the control circuitry to activate the control-signals for steps 4 through 7. The step 4 through 7 constitutes the Execution Phase.

Step4--> Contents of R3 are loaded into MAR & a memory read signal is issued.

Step5--> Contents of R1 are transferred to Y to prepare for addition.

Step6--> When Read operation is completed, memory-operand is available in MDR, and the addition is performed.

Step7--> Sum is stored in Z, then transferred to R1. The End signal causes a new instruction fetch cycle to begin by returning to step1.

Branching Instructions

Instruction execution proceeds as follows:

Step 1-3--> The processing starts & the fetch phase ends in step3.

Step 4--> The offset-value is extracted from IR by instruction-decoding circuit. Since the updated value of PC is already available in register Y, the offset X is gated onto the bus, and an addition operation is performed.

Step 5--> the result, which is the branch-address, is loaded into the PC. The branch instruction loads the branch target address in PC so that PC will fetch the next instruction from the branch target address. The branch target address is usually obtained by adding the offset in the contents of PC. The offset X is usually the difference between the branch target-address and the address immediately following the branch instruction.

In case of conditional branch, we have to check the status of the condition-codes before loading a new value into the PC. e.g.: Offset-field-of-IRout, Add, Zin, If N=0 then End If N=0, processor returns to step 1 immediately after step 4. If N=1, step 5 is performed to load a new value into PC.