MODULE-1

Introduction to Digital Design

Syllabus

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. Text book 1: 1.9, 2.4, 2.5, 2.8, 3.1, 3.2, 3.3, 3.5, 3.6, 3.9

Text book 1: **M. Morris Mano** & Michael D. Ciletti, **Digital Design With an Introduction to Verilog Design**, 5e, Pearson Education.

BINARY LOGIC

Definition of Binary Logic

- **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 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**.

Example, x.y = z or xy = z is read "x AND y is equal to z."

If z = 1 if and only if x = 1 and y = 1; otherwise z = 0. The result of the operation x .y is z.

x, y, and z are binary variables and can be equal either to 1 or 0

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 (or by an **overbar**).

For example, x'= z (or x = z) is read "not x is equal to z,"**.**

If x = 1, then z = 0, and 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.

Truth table

- A **truth table** is a table of all possible combinations of the variables, showing the relation between the values that the variables may take and the result of the operation.
- The truth tables for the operations AND and OR with variables x and y are obtained by listing all possible values that the variables may have when combined in pairs.

Truth Tables of Logical Operations

AND			OR			NOT	
x	y	$x \cdot y$	x	y	$x + y$	x	x'
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	0
1	0	0	1	0	1		
1	1	1	1	1	1		

Gates:

"A digital circuit having one or more input signals but only one output signal is called a gate".

Logic Gates:

"The gates which perform logical operation is called logic gates".

- It take binary input and gives binary outputs.
- The output of the logic gates can be understood using truth table, which contains inputs, outputs of logic circuits.

Logic gates are electronic circuits that operate on one or more input signals to produce an output signal.

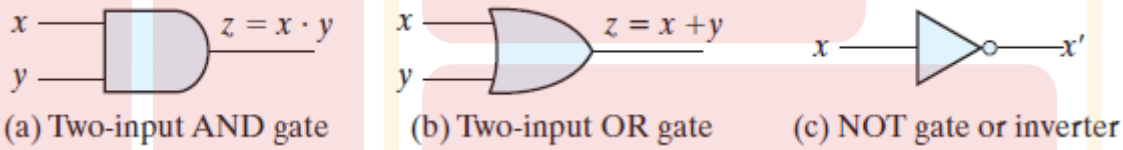


Fig: Symbols for digital logic circuits

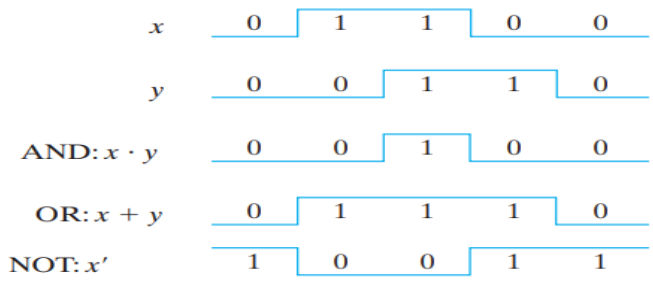
The **timing diagrams** illustrate the idealized response of each gate to the four input signal combinations. The **horizontal axis** of the timing diagram represents the **time**, and the **vertical axis** shows the **signal** as it changes between the two possible voltage levels.

The low level represents logic 0, the high level logic 1.

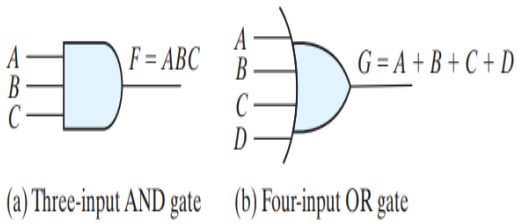
The **AND gate** responds with a logic 1 output signal when both input signals are logic 1.

The **OR gate** responds with a logic 1 output signal if any input signal is logic 1.

The **NOT gate** is commonly referred to as an inverter. The output signal inverts the logic sense of the input signal.



Input-Output Signals for gates Timing Diagram



(a) Three-input AND gate (b) Four-input OR gate

Gates with multiple inputs

BASIC THEOREMS AND PROPERTIES OF BOOLEAN ALGEBRA:

Duality:

- The important property of Boolean algebra is called the *duality principle* and states that every algebraic expression deducible from the postulates of Boolean algebra remains valid if the operators and identity elements are interchanged.
- In a two-valued Boolean algebra, the identity elements and the elements of the set *B* are the same: 1 and 0. The duality principle has many applications.
- If the *dual* of an algebraic expression is desired, we simply interchange OR and AND operators and replace 1's by 0's and 0's by 1's.

Basic Theorems

- Table 1 lists out six theorems of Boolean algebra and four of its postulates. The theorems and postulates listed are the most basic relationships in Boolean algebra.
- The theorems, like the postulates, are listed in pairs; each relation is the dual of the one paired with it. The postulates are basic axioms of the algebraic structure and need no proof.
- The theorems must be proven from the postulates.

Table 2.1
Postulates and Theorems of Boolean Algebra

Postulate 2	(a)	$x + 0 = x$	(b)	$x \cdot 1 = x$
Postulate 5	(a)	$x + x' = 1$	(b)	$x \cdot x' = 0$
Theorem 1	(a)	$x + x = x$	(b)	$x \cdot x = x$
Theorem 2	(a)	$x + 1 = 1$	(b)	$x \cdot 0 = 0$
Theorem 3, involution		$(x')' = x$		
Postulate 3, commutative	(a)	$x + y = y + x$	(b)	$xy = yx$
Theorem 4, associative	(a)	$x + (y + z) = (x + y) + z$	(b)	$x(yz) = (xy)z$
Postulate 4, distributive	(a)	$x(y + z) = xy + xz$	(b)	$x + yz = (x + y)(x + z)$
Theorem 5, DeMorgan	(a)	$(x + y)' = x'y'$	(b)	$(xy)' = x' + y'$
Theorem 6, absorption	(a)	$x + xy = x$	(b)	$x(x + y) = x$

THEOREM 1(a): x + x = x.

Statement	Justification
x + x = (x + x) . 1	postulate 2(b)
= (x + x)(x + x')	5(a)
= x + xx'	4(b)
= x + 0	5(b)
= x	2(a)

THEOREM 1(b): x . x = x.

Statement	Justification
x . x = xx + 0	postulate 2(a)
= xx + xx'	5(b)
= x(x + x')	4(a)
= x . 1	5(a)
= x	2(b)

Note that theorem 1(b) is the dual of theorem 1(a) and that each step of the proof in part (b) is the dual of its counterpart in part (a)..

THEOREM 2(a): x + 1 = 1.

Statement	Justification
x + 1 = 1 . (x + 1)	postulate 2(b)
= (x + x')(x + 1)	5(a)
= x + x' . 1	4(b)
= x + x'	2(b)
= 1	5(a)

THEOREM 2(b): x . 0 = 0 by duality.

THEOREM 3: (x')' = x. From postulate 5, we have x + x' = 1 and x . x' = 0, which together define the complement of x. The complement of x' is x and is also (x')'.

Therefore, since the complement is unique, we have (x')' = x. The theorems involving two or three variables may be proven algebraically from the postulates and the theorems that have already been proven.

THEOREM 6(a): x + xy = x.

Statement	Justification
x + xy = x . 1 + xy	postulate 2(b)
= x(1 + y)	4(a)

= x(y + 1) 3(a)

= x . 1 2(a)

= x 2(b)

THEOREM 6(b): x(x + y) = x by duality.

- The theorems of Boolean algebra can be proven by means of truth tables.
- In truth tables, both sides of the relation are checked to see whether they yield identical results for all possible combinations of the variables involved.
- The following truth table verifies the first absorption theorem:

x	y
0	0
0	1
1	0
1	1

xy	x + xy
0	0
0	0
0	1
1	1

The truth table for the first DeMorgan's theorem, (x + y)' = x'y', is as follows:

x	y	x + y	(x + y)'
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

x'	y'	x'y'
1	1	1
1	0	0
0	1	0
0	0	0

The **operator precedence** for evaluating Boolean expressions is

(1) parentheses, (2) NOT, (3) AND, and (4) OR

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.

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. 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 2n, where n is the number of variables in the function.
- Table 1 shown below represents 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 table shows that the function is equal to 1 when x = 1 or when yz = 01 and is equal to 0 otherwise.

Truth Tables for F_1 and F_2

x	y	z	F_1	F_2
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	0	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	0
1	1	1	1	0

Table 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 for F1 is shown in Fig. 2.1. 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.

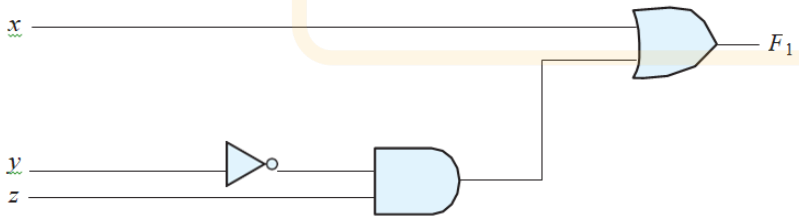


FIGURE 2.1
Gate implementation of $F_1 = x + y'z$

Consider, for example, the following Boolean function:**F2 = x'y'z + x'yz + xy'**

- A schematic of an implementation of this function with logic gates is shown in Fig. 2.2(a).
- Input variables x and y are complemented with inverters to obtain x' and y'. The three terms in the expression are implemented with three AND gates. The OR

gate forms the logical OR of the three terms. **The truth table for F2 is listed in Table 2.2.**

Now consider the possible **simplification** *of the function by* **applying** some of the **identities of Boolean algebra**:

F2 = x'y'z + x'yz + xy' = x'z(y' + y) + xy' = **x'z + xy'**

- The function is reduced to only two terms and can **be implemented with gates as shown in Fig. 2.2(b).** It is obvious that the circuit in (b) is simpler than the one in (a), yet both implement the same function.
- By means of a truth table, it is possible to verify that the two expressions are equivalent. The simplified expression is equal to 1 when xz = 01 or when xy = 10.
- In general, there are many equivalent representations of a logic function. Finding the most economic representation of the logic is an important design task.

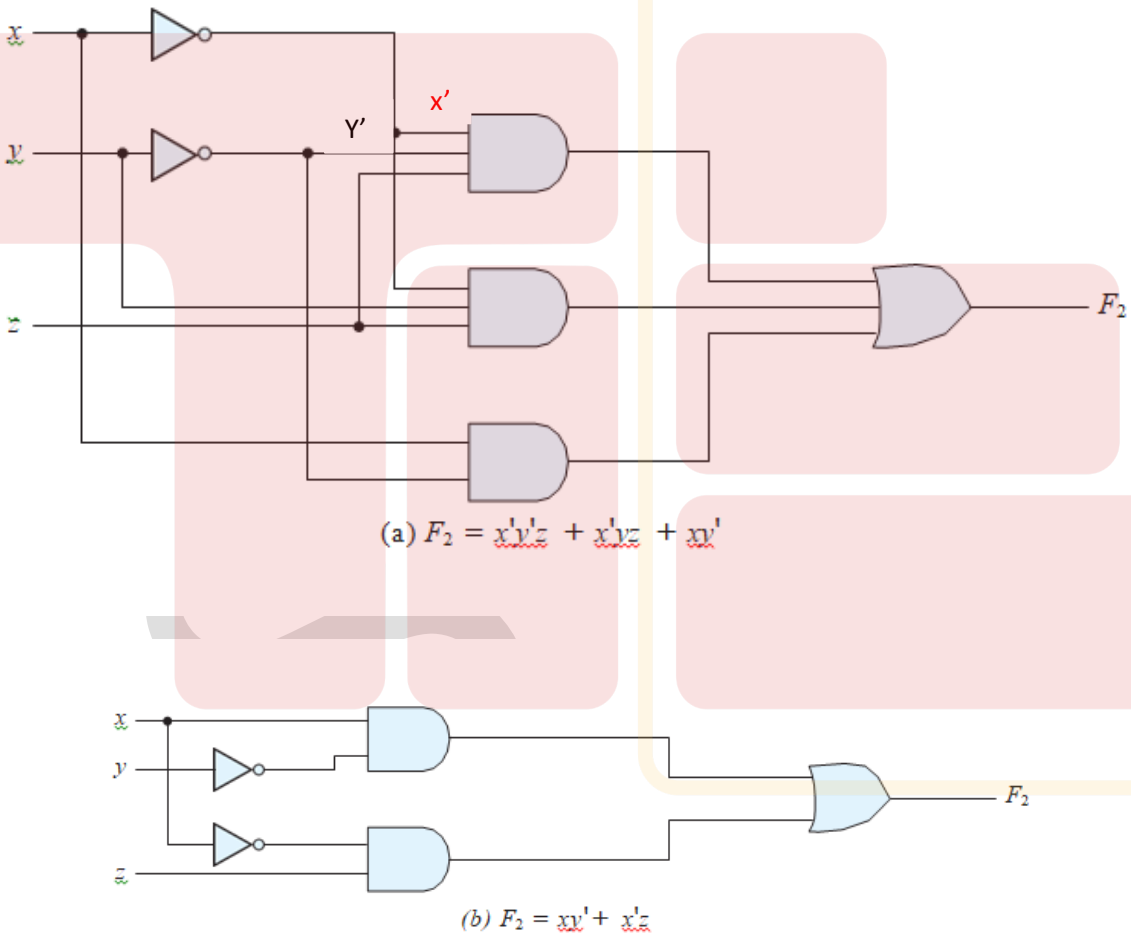


FIGURE 2.2
Implementation of Boolean function F_2 with gates

Complement of a Function

$$(A + B + C)' = (A + x)' \qquad \text{let } B + C = x$$
$$= A'x' \qquad \text{by theorem 5(a) (DeMorgan)}$$
$$= A'(B + C)' \qquad \text{substitute } B + C = x$$
$$= A'(B'C') \qquad \text{by theorem 5(a) (DeMorgan)}$$
$$= A'B'C' \qquad \text{by theorem 4(b) (associative)}$$

$$(A + B + C + D + \cdots + F)' = A'B'C'D' \ldots F'$$
$$(ABCD \ldots F)' = A' + B' + C' + D' + \cdots + F'$$

Find the complement of the functions $F_1 = x'yz' + x'y'z$ and $F_2 = x(y'z' + yz)$. By applying DeMorgan's theorems as many times as necessary, the complements are obtained as follows:

$$F_1 = x'yz' + x'y'z$$
$$F_1' = (x'yz' + x'y'z)'$$
$$= (x'yz')'(x'y'z)'$$
$$= (x + y' + z)(x + y + z')$$

$$F_2' = [x(y'z' + yz)]'$$
$$= x' + (y'z' + yz)' = x' + (y'z')'(yz)'$$
$$= x' + (y + z)(y' + z')$$
$$= x' + yz' + y'z$$

Find the complement of the functions F_1 and F_2 of Example 2.2 by taking their duals and complementing each literal.

1. $F_1 = x'yz' + x'y'z.$

The dual of F_1 is $(x' + y + z')(x' + y' + z)$.

Complement each literal: $(x + y' + z)(x + y + z') = F_1'$.

2. $F_2 = x(y'z' + yz).$

The dual of F_2 is $x + (y' + z')(y + z)$.

Complement each literal: $x' + (y + z)(y' + z') = F_2'$.

DIGITAL LOGIC GATES

- Since Boolean functions are expressed in terms of AND, OR, and NOT operations, it is easier to implement a Boolean function with these type of gates.
- Factors to be weighed in considering the construction of other types of logic gates are (1) the feasibility and economy of producing the gate with physical components, (2) the possibility of extending the gate to more than two inputs, (3) the basic properties of the binary operator, such as commutativity and associativity, and (4) the ability of the gate to implement Boolean functions alone or in conjunction with other gates.
- The graphic symbols and truth tables of the eight gates are shown in Fig. 2.5. Each gate has one or two binary input variables, designated by x and y, and one binary output variable, designated by F.
- The **inverter** circuit inverts the logic sense of a binary variable, producing the NOT, or complement, function. The small circle in the output of the graphic symbol of an inverter (referred to as a bubble) designates the logic complement.
- The **NAND function** is the complement of the AND function, as indicated by a graphic symbol that consists of an AND graphic symbol followed by a small circle.
- The **NOR function** is the complement of the OR function and uses an OR graphic symbol followed by a small circle.
- NAND and NOR gates are used extensively as standard logic gates and are in fact far more popular than the AND and OR gates. This is because NAND and NOR gates are easily constructed with transistor circuits and because digital circuits can be easily implemented with them.
- The **exclusive-OR gate** has a graphic symbol similar to that of the OR gate, except for the additional curved line on the input side. The equivalence, or exclusive-NOR, gate is the complement of the exclusive-OR, as indicated by the small circle on the output side of the graphic symbol.

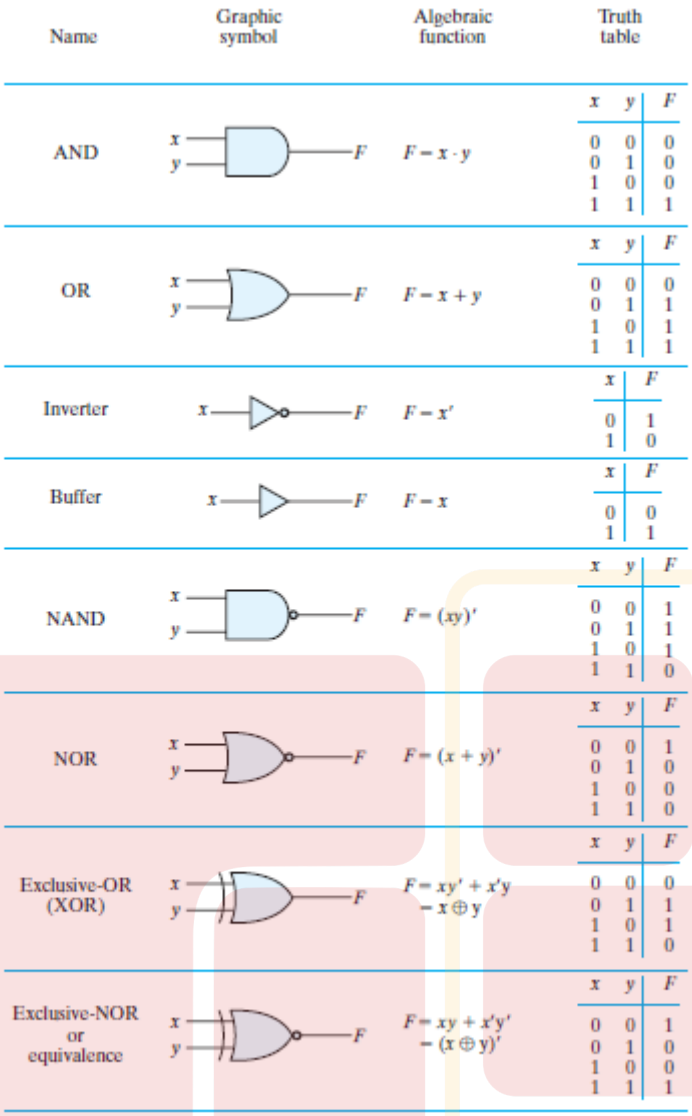


FIGURE 2.5
Digital logic gates

THE MAP METHOD

- **The map method provides a simple, straightforward procedure for minimizing Boolean functions**. This method may be regarded as a **pictorial form of a truth table**. The map method is also known as the **Karnaugh map or K-map**.
- A K-map is a diagram made up of squares, with each square representing **one minterm** of the function that is to be minimized.
- Since any Boolean function can be expressed as a sum of minterms, it follows that a Boolean function is recognized graphically in the map from the area enclosed by those squares whose minterms are included in the function.
- Map represents a visual diagram of all possible ways a function may be expressed in standard form. By recognizing various patterns, the user can derive alternative algebraic expressions for the same function, from which the simplest can be selected.

- **The simplified expressions produced by the map are always in one of the two standard forms: sum of products or product of sums**.
- The simplest algebraic expression is an algebraic expression with a minimum number of terms and with the smallest possible number of literals(variable) in each term. This expression produces a circuit diagram with a minimum number of gates and the minimum number of inputs to each gate.

FOUR-VARIABLE K-MAP

- The map for Boolean functions of four binary variables (w, x, y, z) is shown in Fig. 3.8. In Fig. 3.8(a) are listed the 16 minterms and the squares assigned to each.
- In Fig. 3.8(b), the map is redrawn to show the relationship between the squares and the four variables. The rows and columns are numbered in a Gray code sequence, **with only one digit changing value between two adjacent rows or columns.**
- The minterm corresponding to each square can be obtained from the concatenation of the row number with the column number. For example, the numbers of the third row (11) and the second column (01), when concatenated, give the binary number 1101, the binary equivalent of decimal 13. Thus, the square in the third row and second column represents minterm m13.
- **One square represents one minterm, giving a term with four literals. Two adjacent squares represent a term with three literals.**
- Four adjacent squares represent a term with two literals. Eight adjacent squares represent a term with one literal.

- Sixteen adjacent squares produce a function that is always equal to No other combination of squares can simplify the function.

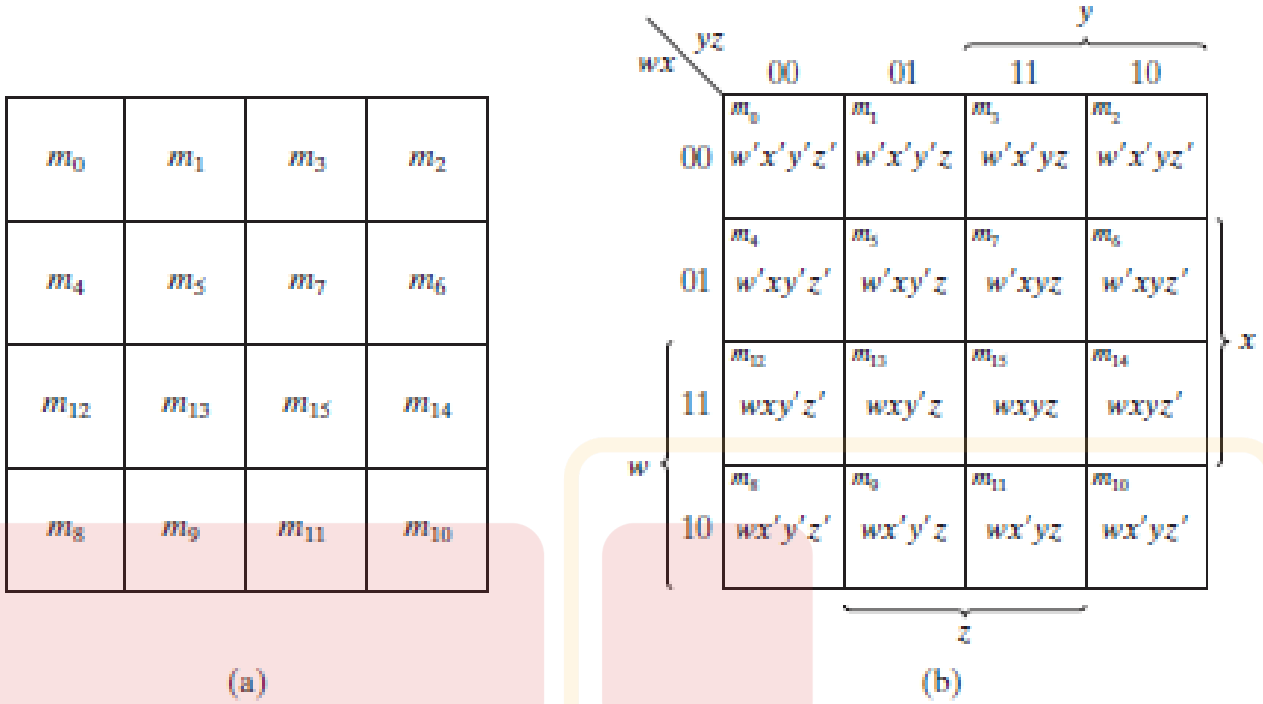


FIGURE 3.8
Four-variable map

1)Simplify the Boolean function

F (w, x, y, z) = Σm **(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)**

Since the function has four variables, a four-variable map must be used. The minterms listed in the sum are marked by 1's in the map of Fig. 3.9.

Eight adjacent squares marked with 1's can be combined to form the one literal term y'. The remaining three 1's on the right cannot be combined to give a simplified term; they must be combined as two or four adjacent squares.

The larger the number of squares combined, the smaller is the number of literals in the term. In this example, the top two 1's on the right are combined with the top two 1's on the left to give the term **w'z'.**

These squares make up the two middle rows and the two end columns, giving the term **xz'.** The simplified function is

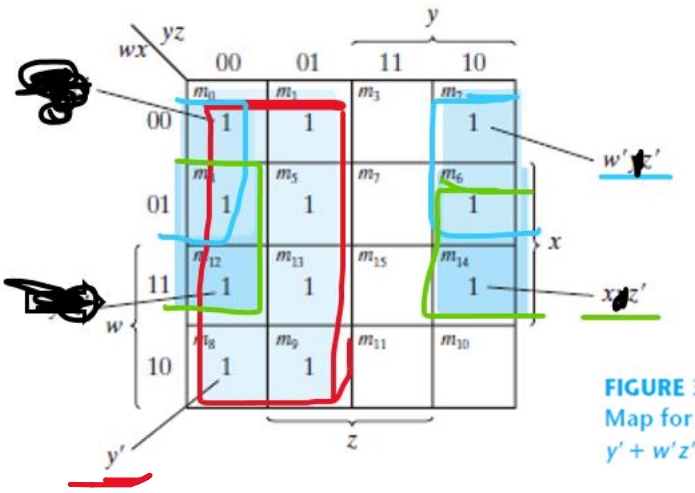
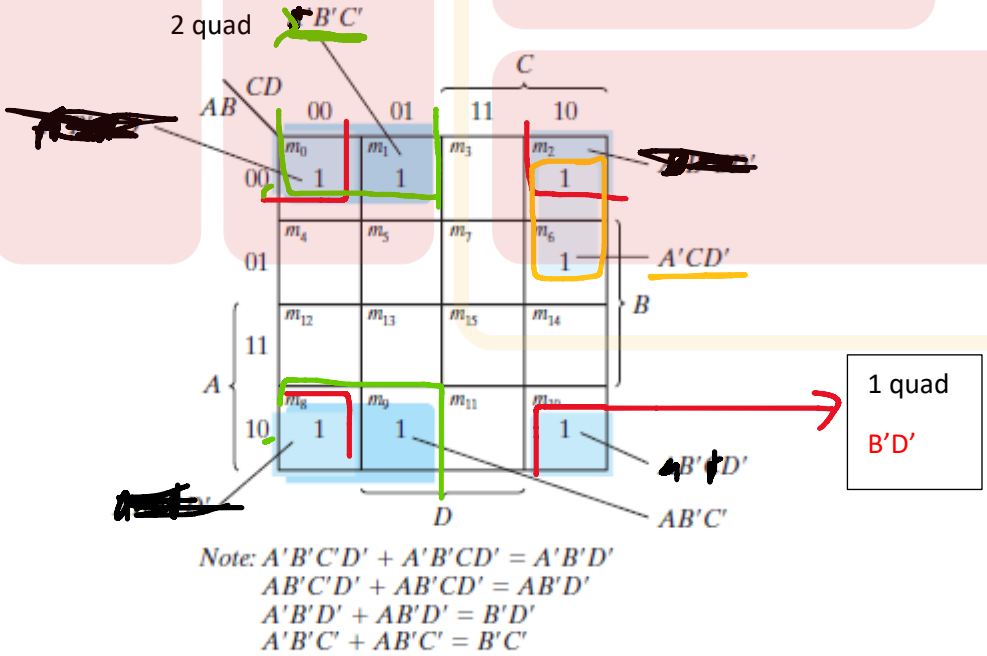


FIGURE 3.9
Map for Example 3.5, $F(w, x, y, z) = \Sigma(0,1, 2, 4, 5, 6, 8, 9, 12, 13, 14) = y' + w'z' + xz'$

2) Simplify the Boolean function

F = A'B'C' + B'CD' + A'BCD' + AB'C'

- F=A’B’C’(D+D’)+B’CD’(A+A’)+A’BCD’+AB’C’(D+D’)
- F=A’B’C’D’+A’B’C’D+AB’CD’+A’B’CD’ +A’BCD’+AB’C’D+AB’C’D’



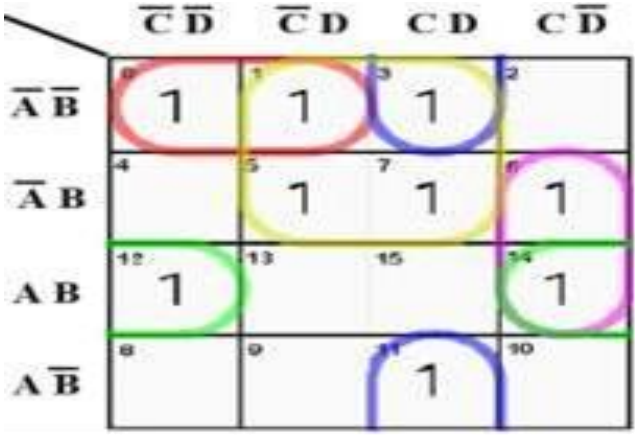
Note: $A'B'C'D' + A'B'CD' = A'B'D'$
$AB'C'D' + AB'CD' = AB'D'$
$A'B'D' + AB'D' = B'D'$
$A'B'C' + AB'C' = B'C'$

FIGURE 3.10
Map for Example 3.6, $A'B'C' + B'CD' + A'BCD' + AB'C' = B'D' + B'C' + A'CD'$

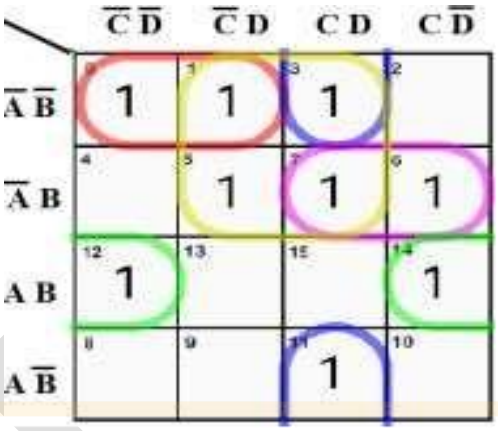
The simplified function is

F = B'D' + B'C' + A'CD'

3) Solve S= F(A,B,C)=Σ *m*(0, 1, 3, 5, 6, 7, 11, 12, 14) using Kmap and implement using basic gates.

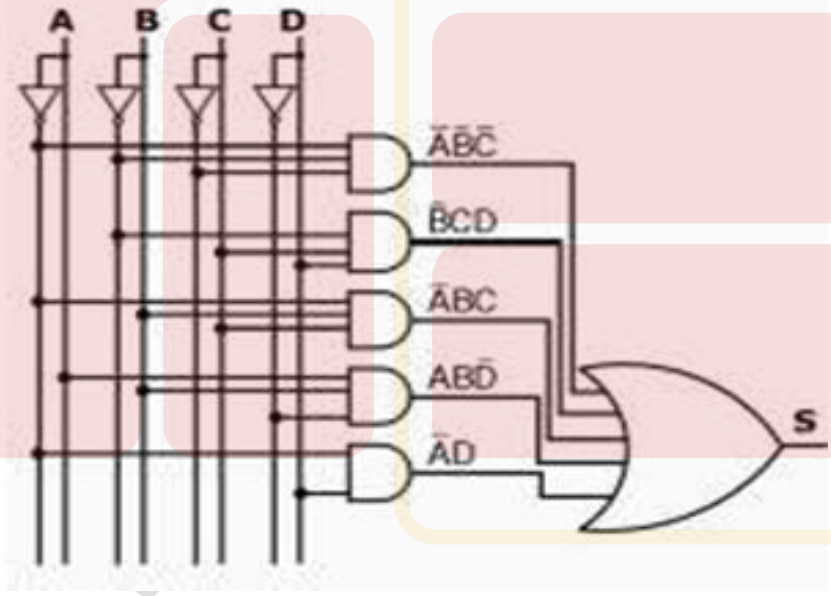


OR

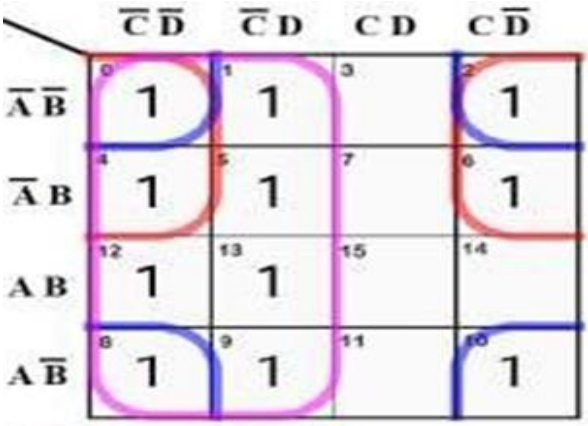


$S = \bar{A}\bar{B}\bar{C} + \bar{B}CD + BC\bar{D} + A\bar{B}\bar{D} + \bar{A}D$ OR $S = \bar{A}\bar{B}\bar{C} + \bar{B}CD + \bar{A}BC + A\bar{B}\bar{D} + \bar{A}D$

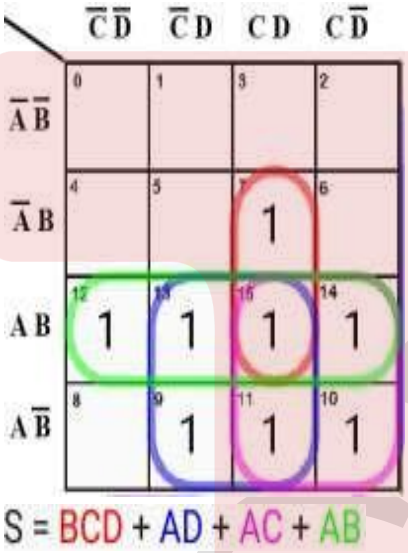
$S = \bar{A}\bar{B}\bar{C} + \bar{B}CD + \bar{A}BC + A\bar{B}\bar{D} + \bar{A}D$



4) Solve S=Σ *M*(0,1, 2, 4, 5,6, 8,9,10,12,13) using Kmap

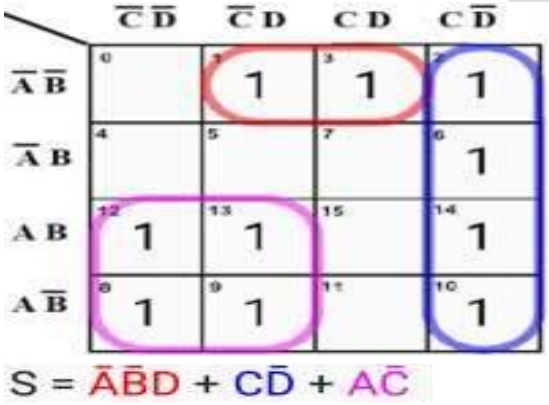


5) Solve S= F(A,B,C,D)=Σm(7,9,10,11,12,13,14,15) using K map to get minimum SOP expression.



Solve S=F(A,B,C,D)=Σm(1,2,3,6,8,9,10,12,13,14) using K map

to get minimum SOP expression.



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).

- **A prime implicant** is a **product term** obtained by combining the **maximum possible number of adjacent squares in the map.**
- The prime implicants of a function can be obtained from the map by combining all possible maximum numbers of squares.
- prime implicants are the building blocks used in the simplification of Boolean functions

Essential Prime Implicant:

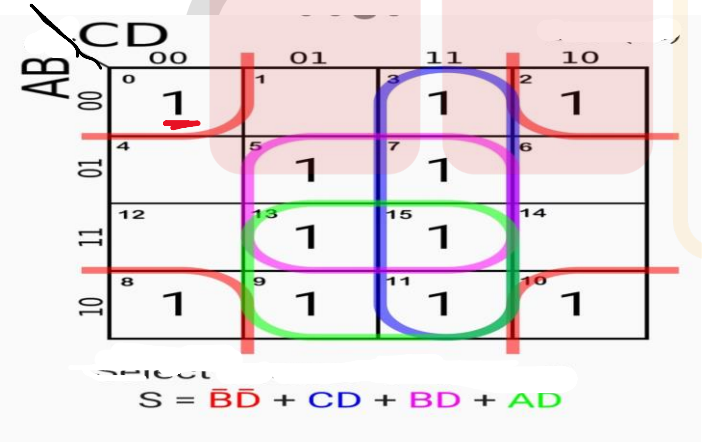
- An essential prime implicant is a prime implicant that covers **at least one minterm that no other prime implicant covers.**

essential prime implicants are a subset of prime implicants that are necessary to cover specific minterms in order to achieve a minimal representation of the Boolean function.

1) Simplify following four-variable Boolean function:

F(A, B, C, D) = Σm (0, 2, 3, 5, 7, 8, 9, 10, 11, 13, 15)

The simplified expression is obtained from the logical sum of the two essential prime implicants and any two prime implicants that cover minterms m3, m9, and m11..



Essential Prime Implicants are BD and B'D'

Prime Implicants are B'D',CD,BD,AD

There are four possible ways that the function can be expressed with four product terms of two literals each:

F = BD + B'D' + CD + AD

= BD + B'D' + CD + AB'

= BD + B'D' + B'C + AD

= BD + B'D' + B'C + AB'

DON'T-CARE CONDITIONS

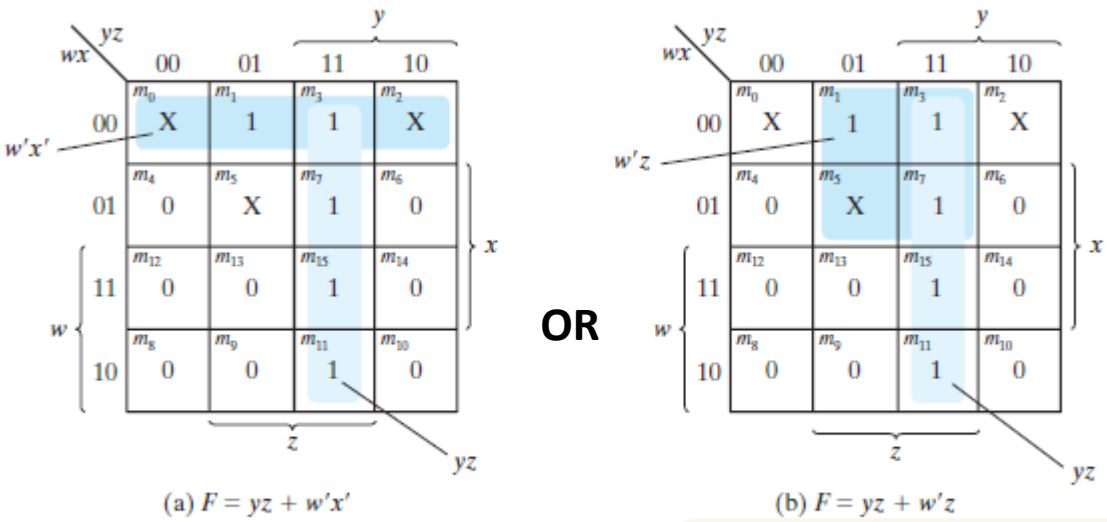
- The logical sum of the minterms associated with a Boolean function specifies the conditions under which the function is equal to 1. The function is equal to 0 for the rest of the minterms. This pair of conditions assumes that all the combinations of the values for the variables of the function are valid.
- In some applications the function is not specified for certain combinations of the variables.
- Functions that have unspecified outputs for some input combinations are called incompletely specified functions.
- In most applications, we simply don't care what value is assumed by the function for the unspecified minterms. For this reason, it is customary to call the unspecified minterms of a function don't-care conditions. These don't-care conditions can be used on a map to provide further simplification of the Boolean expression.
- 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 particular minterm.
- In choosing adjacent squares to simplify the function in a map, the don't-care min- terms may be assumed to be either 0 or 1. When simplifying the function, we can choose to include each don't-care minterm with either the 1's or the 0's, depending on which combination gives the simplest expression.

Simplify the Boolean function

F (w, x, y, z) = (1, 3, 7, 11, 15)which has the don't-care conditions

d (w, x, y, z) = (0, 2, 5)

The minterms of F are the variable combinations that make the function equal to 1. The minterms of d are the don't-care minterms that may be assigned either 0 or 1. The map simplification is shown in Fig. 3.15. The minterms of F are marked by 1's, those of d are marked by X's, and the remaining squares are filled with 0's. To get the simplified expres- sion in sum-of-products form, we must include all five 1's in the map, but we may or may not include any of the X's, depending on the way the function is simplified. The term yz covers the four minterms in the third column.The remaining minterm, m1, can be combined



(a) $F = yz + w'x'$ (b) $F = yz + w'z$

FIGURE 3.15
Example with don't-care conditions

with minterm m3 to give the three-literal term w'x'z. However, by including one or two adjacent X's we can combine four adjacent squares to give a two-literal term. In Fig. 3.15(a), don't-care minterms 0 and 2 are included with the 1's, resulting in the simplified function

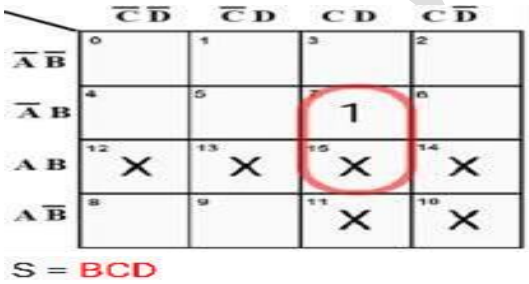
F = yz + w'x'

In Fig. 3.15(b), don't-care minterm 5 is included with the 1's, and the simplified function is now

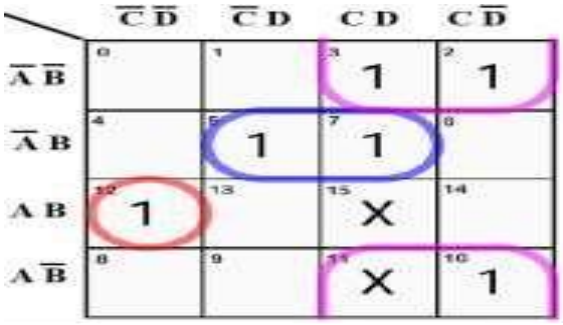
F = yz + w'z

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

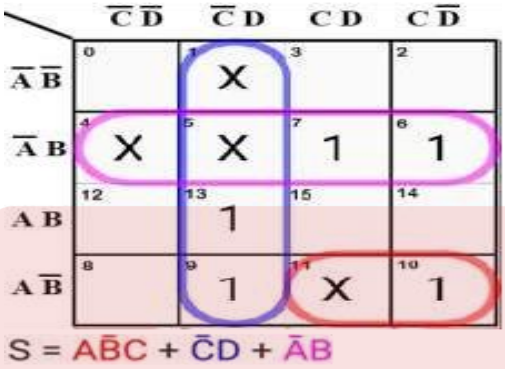
Solve S=F(A,B,C,D)=Σm(7)+d(10,11,12,13,14,15) using K map to get minimum SOP expression



Solve S =F(A,B,C,D)=Σm(2,3,5,7,10,12)+d(11,15) using K map to get minimum SOP expression



Solve S=F(A,B,C,D)=Σm(6,7,9,10,13)+d(1,4,5,11) using K map to get minimum SOP expression.



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.

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. 3.16.
- The complement operation is obtained from a one-input NAND gate that behaves exactly like an inverter. The AND operation requires two NAND gates. The first produces the NAND operation and the second inverts the logical sense of the signal. The OR operation is achieved through a NAND gate with additional inverters in each input.
- 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.
- Two equivalent graphic symbols for the NAND gate are shown in Fig. 3.17. The AND-invert symbol has been defined previously and consists

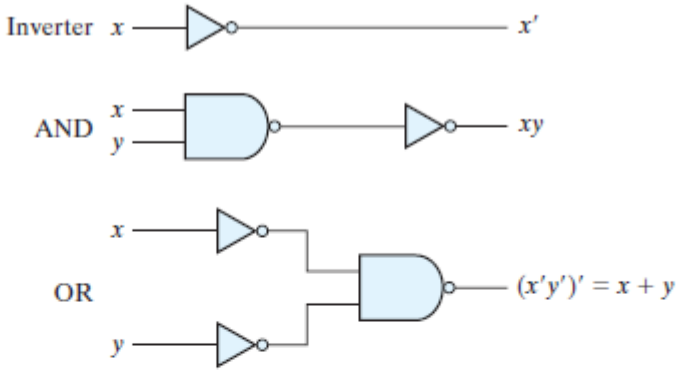
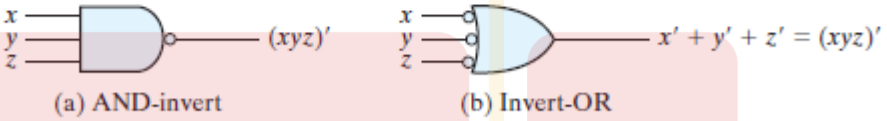


FIGURE 3.16
Logic operations with NAND gates



(a) AND-invert (b) Invert-OR

FIGURE 3.17
Two graphic symbols for a three-input NAND gate

of an AND graphic symbol followed by a small circle negation indicator referred to as a bubble.

- It is possible to represent a NAND gate by an OR graphic symbol that is preceded by a bubble in each input. The invert-OR symbol for the NAND gate follows DeMorgan's theorem and the convention that the negation indicator (bubble) denotes complementation. The two graphic symbols' representations are useful in the analysis and design of NAND circuits. When both symbols are mixed in the same diagram, the circuit is said to be in mixed notation.

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. 3.18. All three diagrams are equivalent and implement the function

  F = AB + CD

- The function is implemented in Fig. 3.18(a) with AND and OR gates. In Fig. 3.18(b), the **AND gates are replaced by NAND gates and the OR gate is replaced by a NAND gate with an OR-invert graphic symbol.** Remember that a bubble denotes complementation and two bubbles along the same line represent double complementation, so both can be removed.

- Removing the bubbles on the gates of (b) produces the circuit of (a). Therefore, the two diagrams implement the same function and are equivalent.

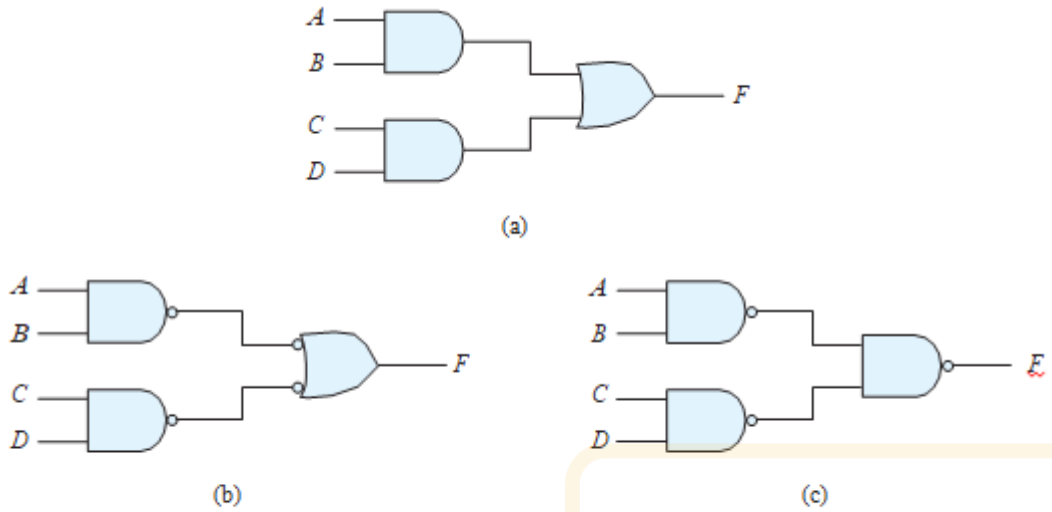


FIGURE 3.18
Three ways to implement $F = AB + CD$

- In Fig. 3.18(c), the output NAND gate is redrawn with the AND-invert graphic symbol. In drawing NAND logic diagrams, the circuit shown in either Fig. 3.18(b) or (c) is acceptable.
- The one in Fig. 3.18(b) is in mixed notation and represents a more direct relationship to the Boolean expression it implements. The NAND implementation in Fig. 3.18(c) can be verified algebraically.
- The function it implements can easily be converted to sum-of- products form by DeMorgan's theorem:

F = ((AB)'(CD)')' = AB + CD

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. 3.19(a), from which the simplified function is obtained:

  F = xy' + x'y + z

- The two-level NAND implementation is shown in Fig. 3.19(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. 3.19(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'

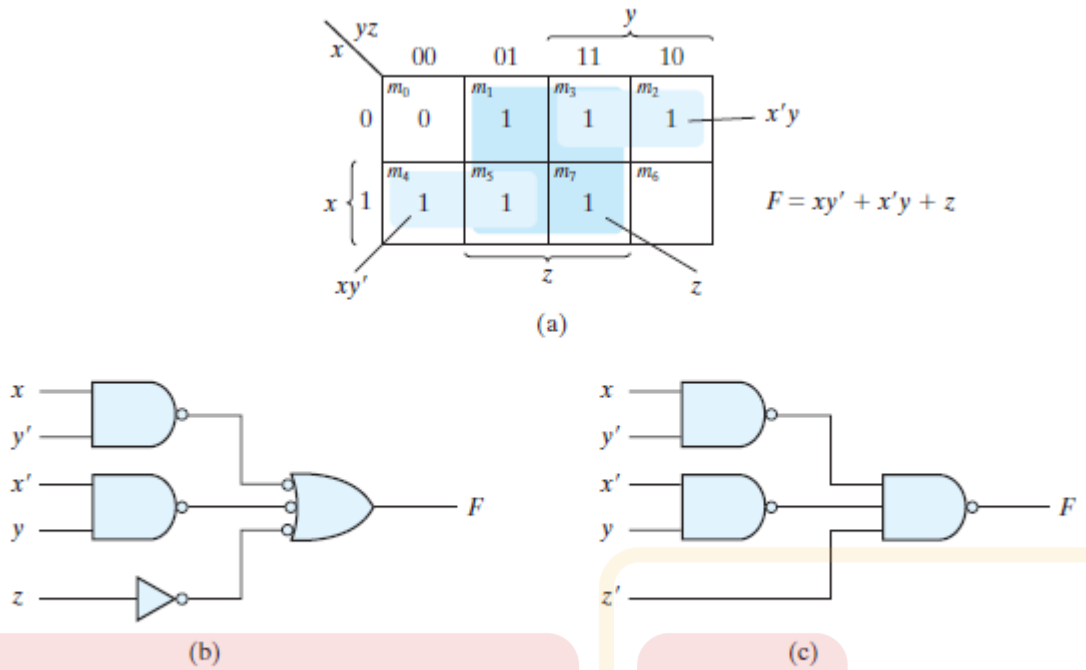


FIGURE 3.19
Solution to Example 3.9

- The procedure described in the previous example indicates that a Boolean function can be implemented with two levels of NAND gates. The **procedure** for **obtaining the logic diagram from a Boolean function is as follows:**

1. Simplify the function and express it in sum-of-products form.

2. **Draw a NAND gate** for each product term of the expression that has **at least two literals.** The inputs to each NAND gate are the literals of the term. This procedure produces a group of **first-level gates**.

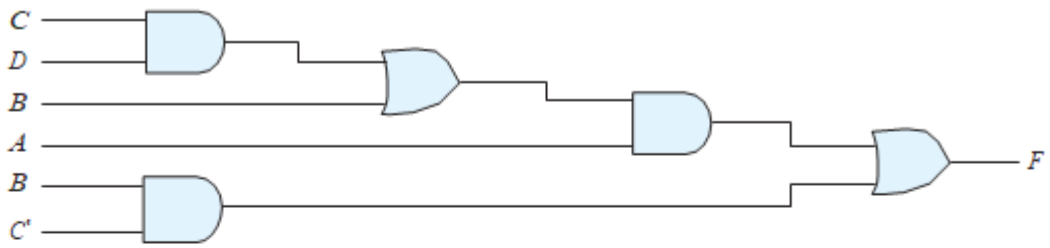
3. Draw a single gate using the **AND-invert or the invert-OR** graphic symbol in the **second level**, with inputs coming from outputs of first-level gates.

4. A term with a single literal requires an inverter in the first level. However, if the single literal is complemented, it can be connected directly to an input of the second-level NAND gate.

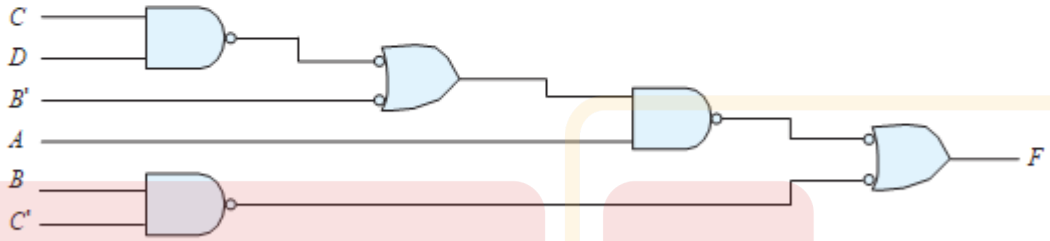
Multilevel NAND Circuits

- The standard form of expressing Boolean functions results in a two-level implementation. There are occasions, however, when the design of digital systems results in gating structures with three or more levels.
- The most common procedure in the design of multilevel circuits is to express the Boolean function in terms of AND, OR, and complement operations. The function can then be implemented with AND and OR gates. After that, if necessary, it can be converted into an all-NAND circuit.

Consider, for example, the Boolean function F = A (CD + B) + BC'



(a) AND–OR gates



(b) NAND gates

FIGURE 3.20
Implementing $F = A(CD + B) + BC'$

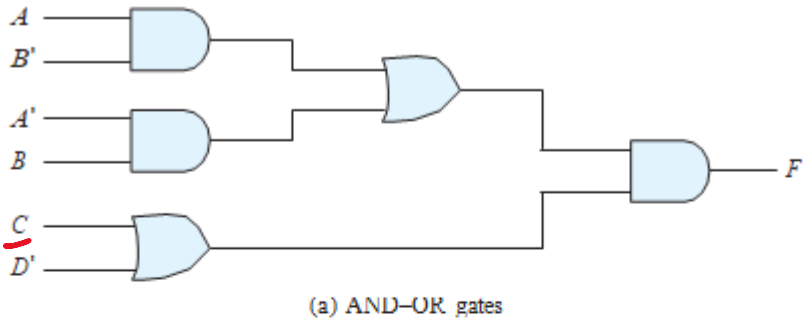
- The AND–OR implementation is shown in Fig. 3.20(a). There are four levels of gating in the circuit. The first level has two AND gates. The second level has an OR gate followed by an AND gate in the third level and an OR gate in the fourth level.
- A logic diagram with a pattern of alternating levels of AND and OR gates can easily be converted into a NAND circuit with the use of mixed notation, shown in Fig. 3.20(b).
- The procedure is to change every AND gate to an AND-invert graphic symbol and every OR gate to an invert-OR graphic symbol. The NAND circuit performs the same logic as the AND–OR diagram as long as there are two bubbles along the same line.
- The bubble associated with input B causes an extra comple- mentation, which must be compensated for by changing the input literal to B'.

The general procedure for converting a **multilevel AND–OR diagram** into an all-NAND diagram using mixed notation is as follows:

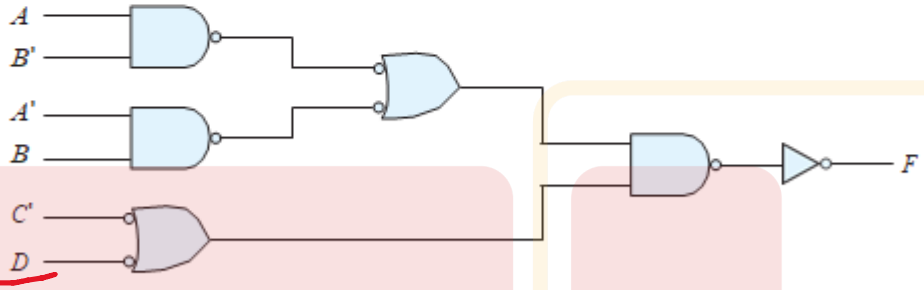
1. **Convert all AND gates** to **NAND** gates with AND-invert graphic symbols.
2. **Convert all OR gates** to NAND gates with **invert-OR** graphic symbols.
3. Check all the bubbles in the diagram. For every bubble that is not compensated by another small circle along the same line, insert an inverter (a one-input NAND gate) or complement the input literal.

As another example, consider the multilevel Boolean function

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



(a) AND–OR gates



(b) NAND gates

FIGURE 3.21
Implementing $F = (AB' + A'B)(C + D')$

- The AND–OR implementation of this function is shown in Fig. 3.21(a) with three levels of gating. The conversion to NAND with mixed notation is presented in Fig. 3.21(b) of the diagram.
- The two additional bubbles associated with inputs C and D' cause these two literals to be complemented to C' and D.
- The bubble in the output NAND gate complements the output value, so we need to insert an inverter gate at the output in order to complement the signal again and get the original value back.

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. 3.22.
- The complement operation is obtained from a one- input NOR gate that behaves exactly like an inverter.The OR operation requires two NOR gates, and the AND operation is obtained with a NOR gate that has inverters in each input.
- The two graphic symbols for the mixed notation are shown in Fig. 3.23. 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.

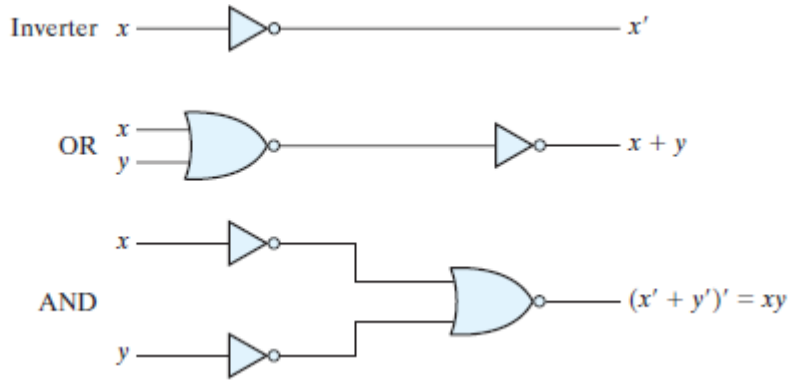
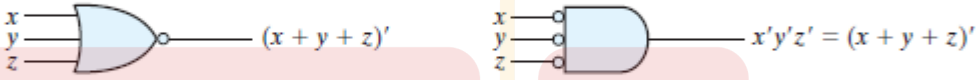


FIGURE 3.22
Logic operations with NOR gates



(a) OR-invert (b) Invert-AND

FIGURE 3.23
Two graphic symbols for the NOR gate

- A two-level implementation with NOR gates requires that the function be simplified **into product-of-sums form.**
- Remember that the simplified product-of-sums expression is obtained from the map by combining the 0's and complementing. A product-of-sums expression is implemented with a first level of OR gates that produce the sum terms followed by a second-level AND gate to produce the product.
- The transformation from the OR–AND diagram to a NOR diagram is achieved by **changing the OR gates to NOR gates** with OR-invert graphic symbols and the **AND gate to** a NOR gate with **an invert-AND** graphic symbol.
- A single literal term going into the second-level gate must be complemented.

Figure 3.24 shows the NOR implementation of a function expressed as a product of sums: F = (A + B)(C + D)E

The OR–AND pattern can easily be detected by the removal of the bubbles along the same line. Variable E is complemented to compensate for the third bubble at the input of the second-level gate.

The procedure for converting a multilevel AND–OR diagram to an all-NOR diagram is similar to the one presented for NAND gates. **For the NOR case**, **we must convert each OR gate to an OR-invert symbol and each AND gate to an invert-AND symbol.** Any bubble that is not compensated by another bubble along the same line needs an inverter, or the complementation of the input literal.

NOR implementation for the following function **F = (A + B)(C + D)E in Fig3.24**

NOR implementation for the following Boolean function is

F = (AB' + A'B)(C + D') in Fig:3.25

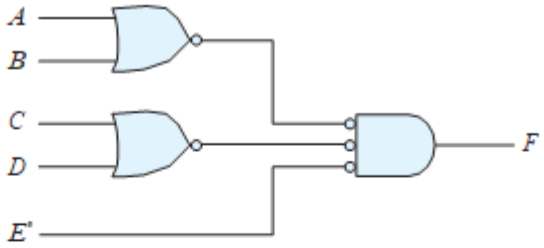


FIGURE 3.24
Implementing $F = (A + B)(C + D)E$

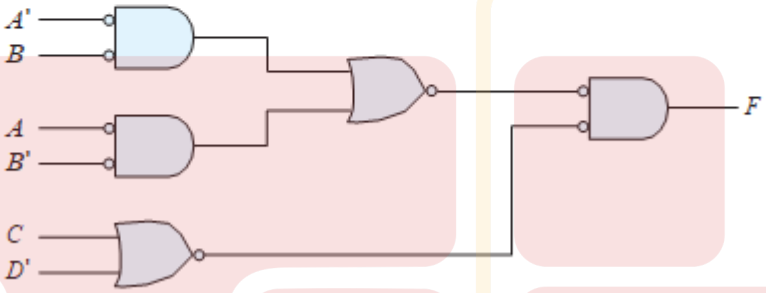


FIGURE 3.25
Implementing $F = (AB' + A'B)(C + D')$ with NOR gates

The equivalent AND–OR diagram can be recognized from the NOR diagram by removing all the bubbles. To compensate for the bubbles in four inputs, it is necessary to complement the corresponding input literals.

Hardware Description Language

- 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.

- 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.
- 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.**
- Verilog is an easier language than VHDL to describe, learn, and use, we have chosen it for this book.
- 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 * /.**
- **Verilog is case sensitive**, which means that uppercase and lowercase letters are distinguishable (e.g., not is not the same as NOT).
- 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*.
- There 3 types of modelling:
- **Dataflow modeling** describes a system in terms of how data flows through the system.
- **Behavioral modeling** describes a system's behavior or function in an algorithmic fashion.
- **Structural modeling** describes a system in terms of its structure and interconnections between components.

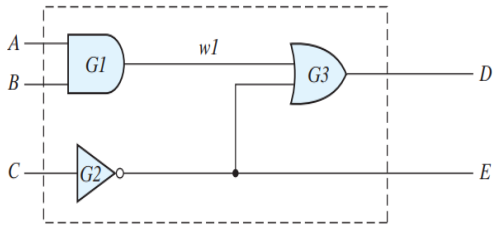
Write a Verilog program for OR gate using i)dataflow modelling ii)Behavioral modelling and iii)structural modelling.

```
1  module dataflow(a, b, y);
2   input a, b;
3   output y;
4   assign y = a | b;
5   endmodule
6
```

```
module beh (a, b, y);
 input a, b;
output y;
 reg y;
 always @ (a or b)
 begin
 if ((a = = 0) && (b = = 0)) y = 0;
else
y = 1;
end
endmodule
```

```
1  module structural(a, b, y);
2   input a, b;
3   output y;
4   or g1(y,a,b);
5   endmodule
6
```

2)write a Verilog code for the following circuit.



```
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
```

HDL describes a circuit that is specified with the following two Boolean expressions:

$$E = A + BC + B'D$$
$$F = B'C + BC'D'$$

```
module beh(E, F, A, B, C, D);
output E, F;
input A, B, C, D;
assign E = A || (B && C) || ((!B) && D);
assign F = ((!B) && C) || (B && (!C) && (!D));
endmodule
```

Module-2

Combinational Logic

Syllabus:

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.

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.
- 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.

combinational circuit

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.

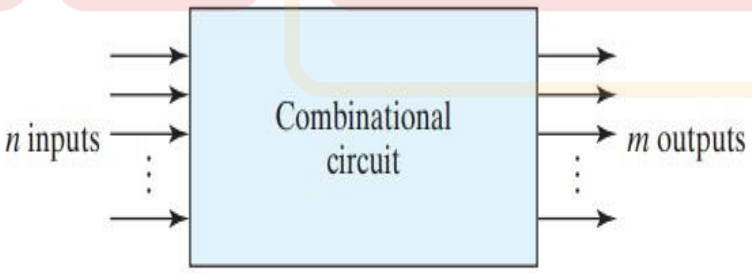


FIGURE 4.1
Block diagram of combinational circuit

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.

- If the **registers** are included with the combinational gates, then the total circuit must be considered to 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.

Design procedure

The procedure to design combinational circiut 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).

Example for design Procedure

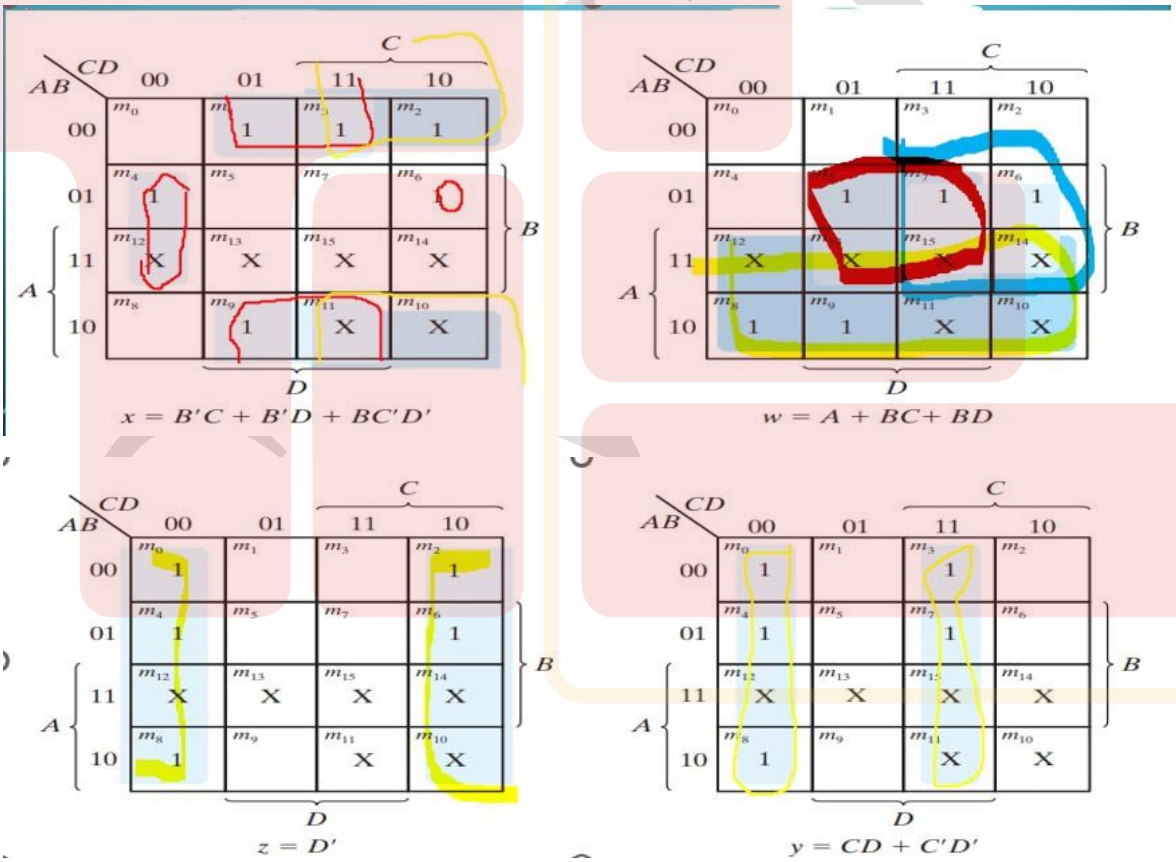
Code Conversion (Convert BCD to Excess-3 Code)

- A code converter is a circuit that makes the two systems compatible even though each uses a different binary code.
- Since each code uses four bits to represent a decimal digit, there must be four input variables and four output variables. We designate the four input binary variables by the symbols A, B, C, and D, and the four output variables by w, x, y , and z .
- **ADD 3** to BCD to get Excess -3 Code

Table 4.2
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

Note that four binary variables may have 16 bit combinations, but only 10 are listed in the truth table. The six bit combinations not listed for the input variables are don't-care combinations.



implemented with three or more levels of gates:

$$z = D'$$
$$y = CD + C'D' = CD + (C + D)'$$
$$x = B'C + B'D + BC'D' = B'(C + D) + BC'D'$$
$$= B'(C + D) + B(C + D)'$$
$$w = A + BC + BD = A + B(C + D)$$

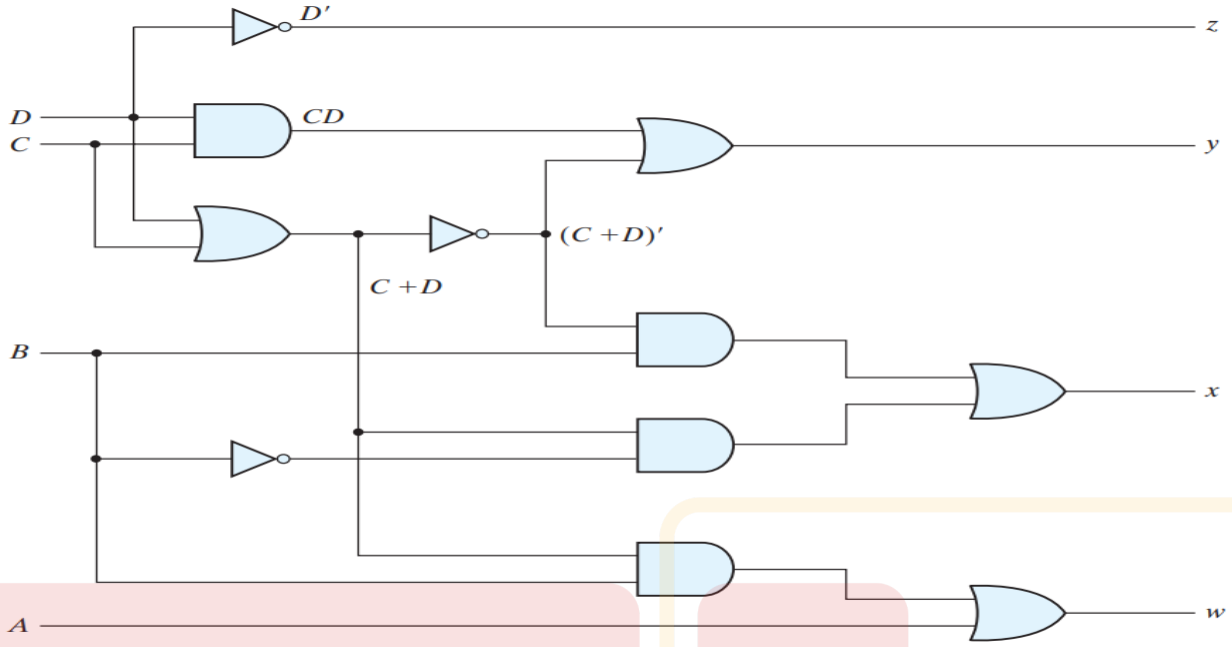


FIGURE 4.4
Logic diagram for BCD-to-excess-3 code converter

Binary Adder- Subtractor

- A combinational circuit that performs the **addition of two bits** is called a **half adder .**
- The **addition of three bits** (two significant bits and a previous carry) is a **full adder.**
- **A binary adder–subtractor** is a combinational circuit that performs the **arithmetic operations of addition and subtraction with binary numbers**.
- The half adder design is carried out first, from which we develop the full adder.
- Connecting n full adders in cascade produces a binary adder for two n -bit numbers.

Half Adder

- Half Adder circuit needs two binary inputs and two binary outputs.
- output variables produce the sum and carry. We assign symbols x and y to the two inputs and S (for sum) and C (for carry) to the outputs.
- The C output is 1 only when both inputs are 1. The S output represents the least significant bit of the sum.
- The truth table for the half adder is listed in Table 4.3 .

Table 4.3
Half Adder

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

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. 4.5(a) . It can be also implemented with an exclusive-OR and an AND gate as shown in Fig. 4.5(b)

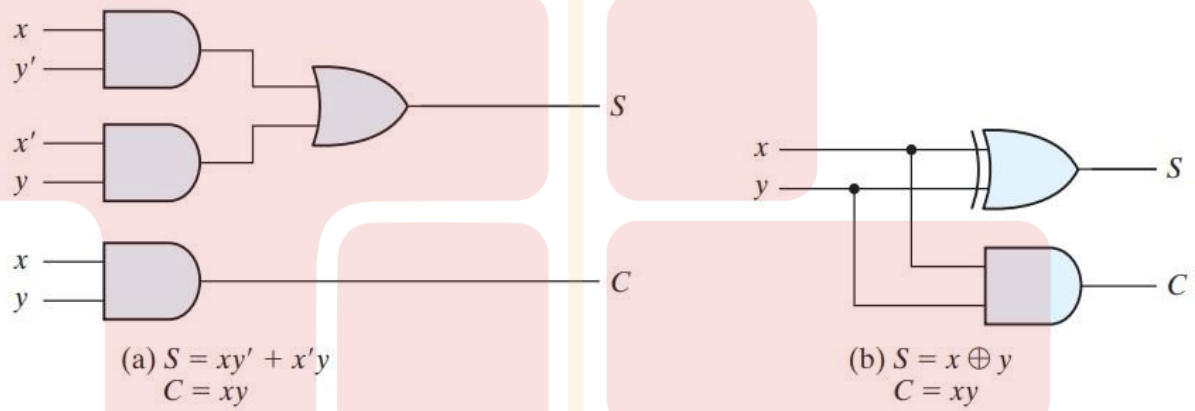


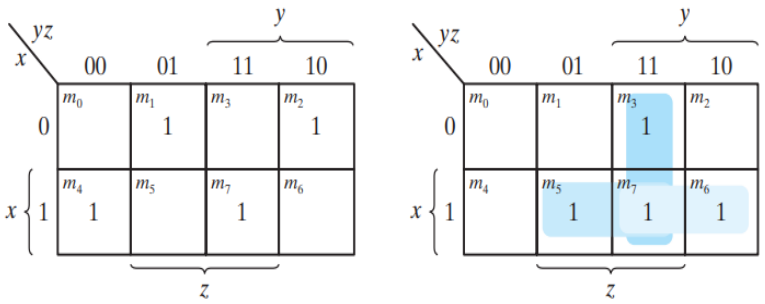
FIGURE 4.5
Implementation of half adder

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. The two outputs are designated by the symbols S for sum and C for carry.

Table 4.4
Full Adder

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



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

FIGURE 4.6
K-Maps for full adder

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

- The logic diagram for the full adder implemented in sum-of-products form is shown in Fig. 4.7

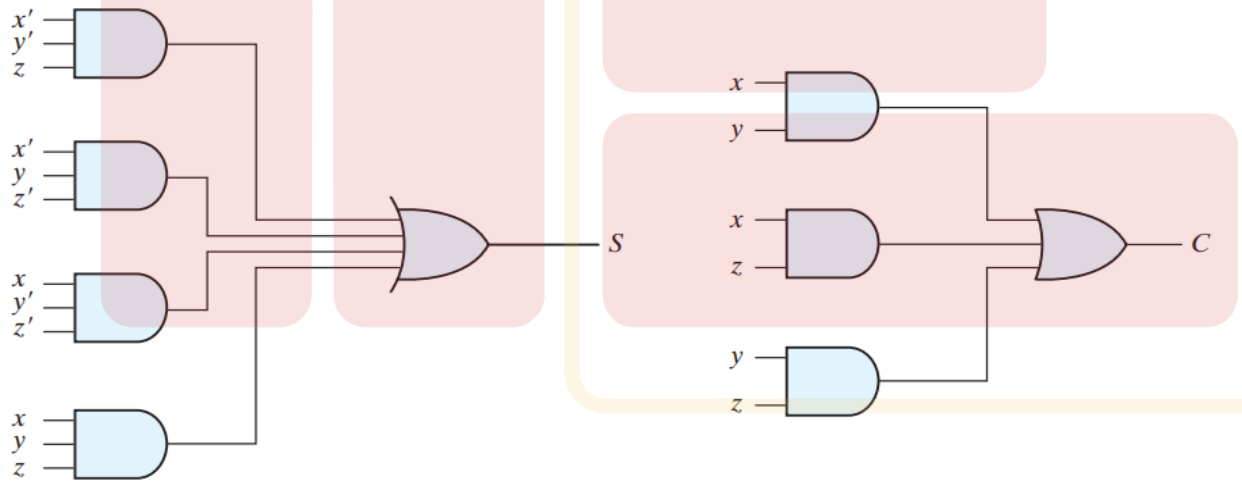


FIGURE 4.7
Implementation of full adder in sum-of-products form

Implementation of Full adder using 2 half adder :

We know that

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

=z'(xy'+x'y)+z(xy+x'y')

=z'(xy'+x'y)+z(x'y+xy')'

=z'(x ⊕ y)+z(x ⊕ y)'

=z'A+zA'

=z ⊕ A

S=z ⊕ x ⊕y

C=xy+xz+yz

=xy+xz(y+y')+yz(x+x')

=xy+xyz+xy'z+xyz+x'yz

=xy+xyz+z(xy'+x'y)

=xy(1+z)+z(x^y)

C=xy+z(x⊕y)

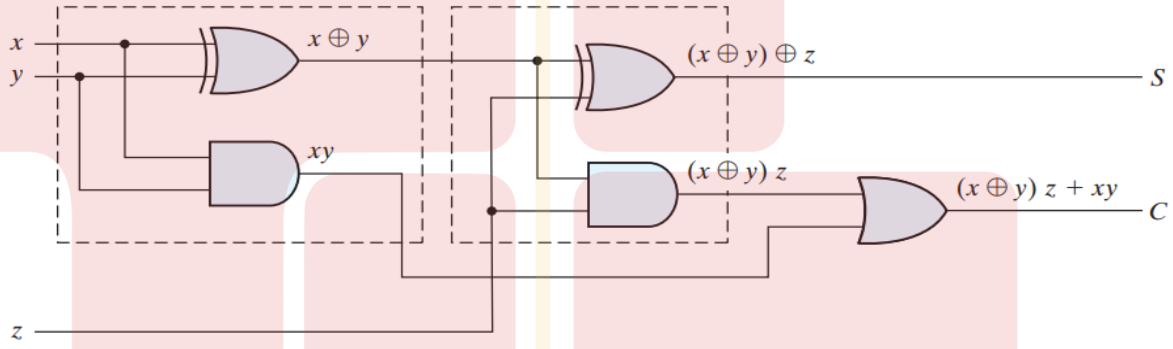


FIGURE 4.8
Implementation of full adder with two half adders and an OR gate

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.**

- n-bit numbers requires a chain of n full adders or a chain of one-half adder and n-1 full adders.
- Eg:4bit numbers requires a chain of 4 fulladders or one HA and 3FAs.
- 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.

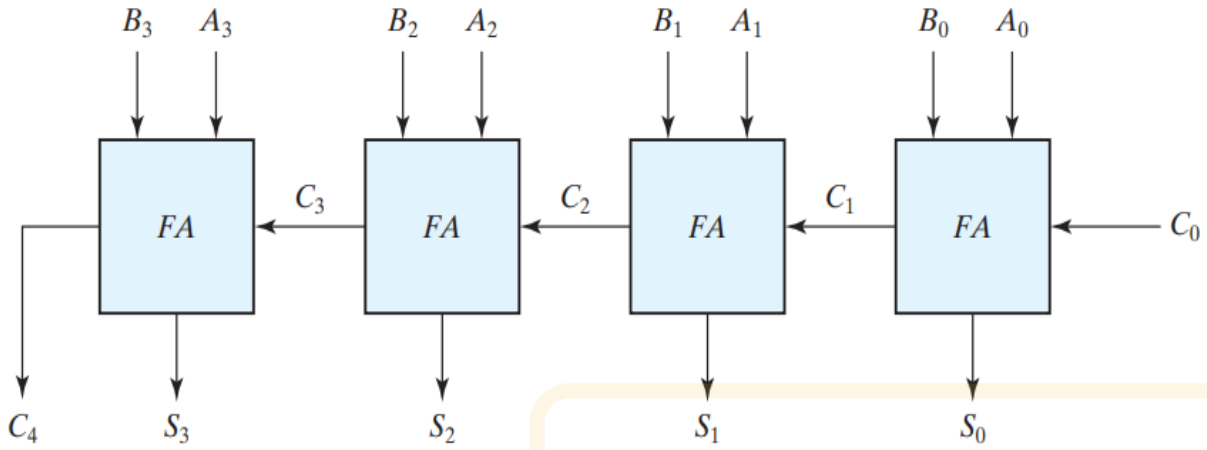


FIGURE 4.9
Four-bit adder

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 **input carry C0** in the least significant position must be **0**.

The value of Ci+1 in a given significant position is the output carry of the full adder.

The carry propagation time is an important attribute of the adder because it limits the speed with which two numbers are added.

1. write Verilog code for 4 bit parallel adder using full adder as component.

```
module fourbit_full_adder(a, b,sum,cout);
input [3:0] a;
input [3:0] b;
output [3:0] sum;
output cout;
wire c1, c2, c3;
full_adder fa0(a[0], b[0], 0, sum[0], c1);
full_adder fa1(a[1], b[1], c1, sum[1], c2);
full_adder fa2(a[2], b[2], c2, sum[2], c3);
full_adder fa3(a[3], b[3], c3, sum[3], cout);
endmodule

module full_adder (a, b, cin, sum, cout);
input a, b, cin;
output sum, cout;
assign sum = a^b^cin;
assign cout = (a&b)|(b&cin)|(cin&a);
endmodule
```

2. **Write Verilog code for 4 bit adder .**

HDL (Dataflow: Four-Bit Adder)

```
module binary_adder (
  output [3: 0]     Sum,
  output            C_out,
  input [3: 0]      A, B,
  input             C_in
);
  assign {C_out, Sum} = A + B + C_in;
endmodule
```

There are several techniques for reducing the carry propagation time in a parallel adder. The most widely used technique employs the principle of **carry lookahead logic .**

Carry Propagation

- Carry Propagation The addition of two binary numbers in parallel implies that all the bits of the augend and addend are available for computation at the same time.
- Consider the circuit of the full adder shown in Fig. 4.10 . If we define two new binary variables.

- Gi is called a carry generate , and it produces a carry of 1 when both Ai and Bi are 1, regardless of the input carry Ci .
- Pi is called a carry propagate , because it determines whether a carry into stage i will propagate into stage i + 1

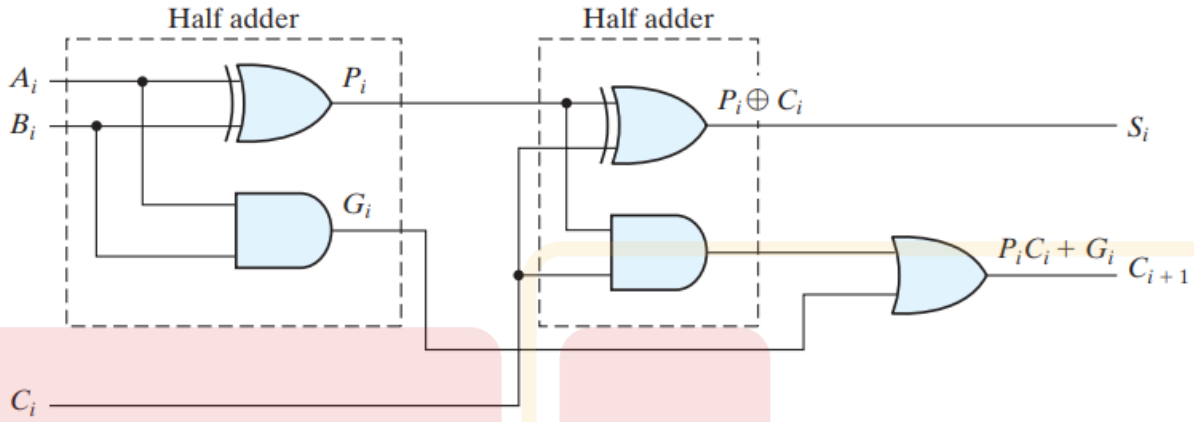


FIGURE 4.10
Full adder with *P* and *G* shown

- $P_i = A_i \oplus B_i \qquad S_i = P_i \oplus C_i$
- $G_i = A_i B_i \qquad C_{i+1} = G_i + P_i C_i$

Binary ADDER-Subtractor

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. 4.13 . The mode input M controls the operation. **When M = 0, the circuit is an adder,** and when **M = 1, the circuit becomes a subtractor.** Each exclusive-OR gate receives input M and one of the inputs of B.

When M = 0, we have B ⊕0 = B. The full adders receive the value of B , the input carry is 0, and the circuit performs A plus B . When M = 1, we have B ⊕ 1 = B' and C0 = 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 . (The exclusive-OR with output V is for detecting an overflow.)

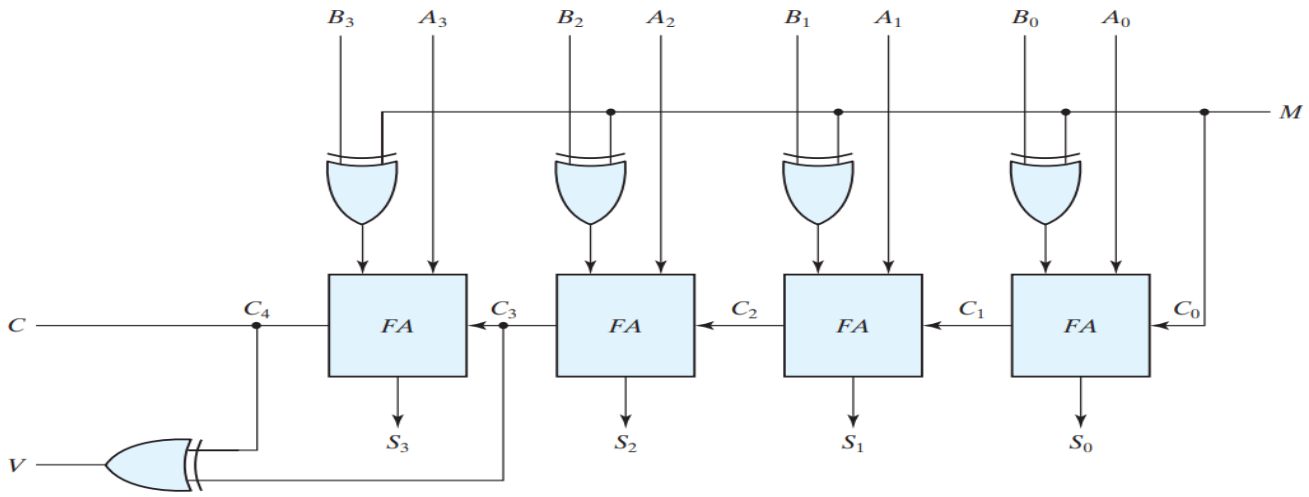
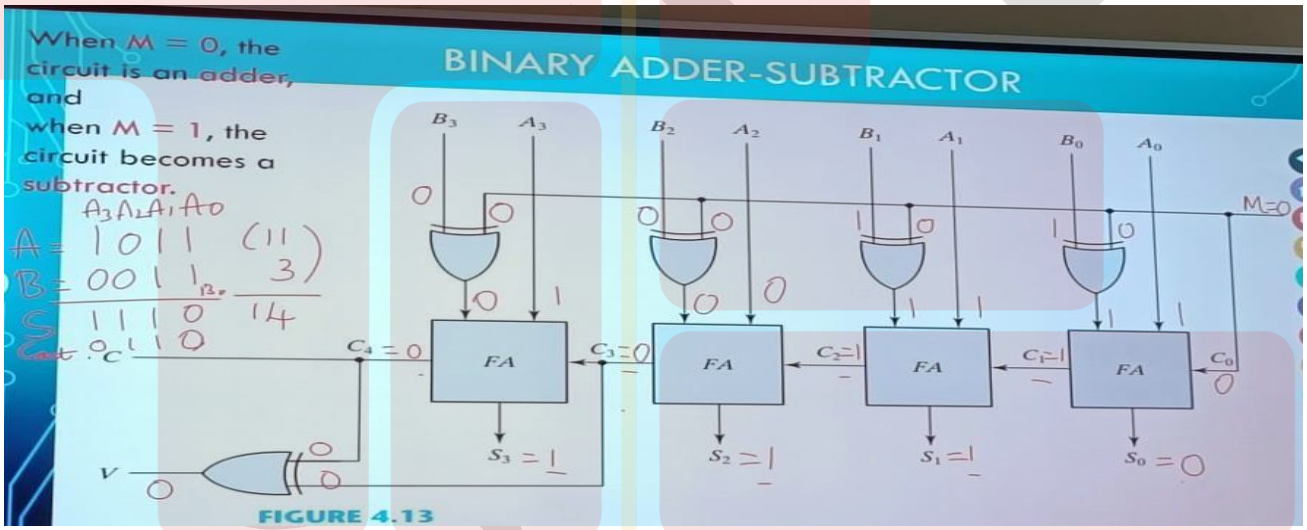
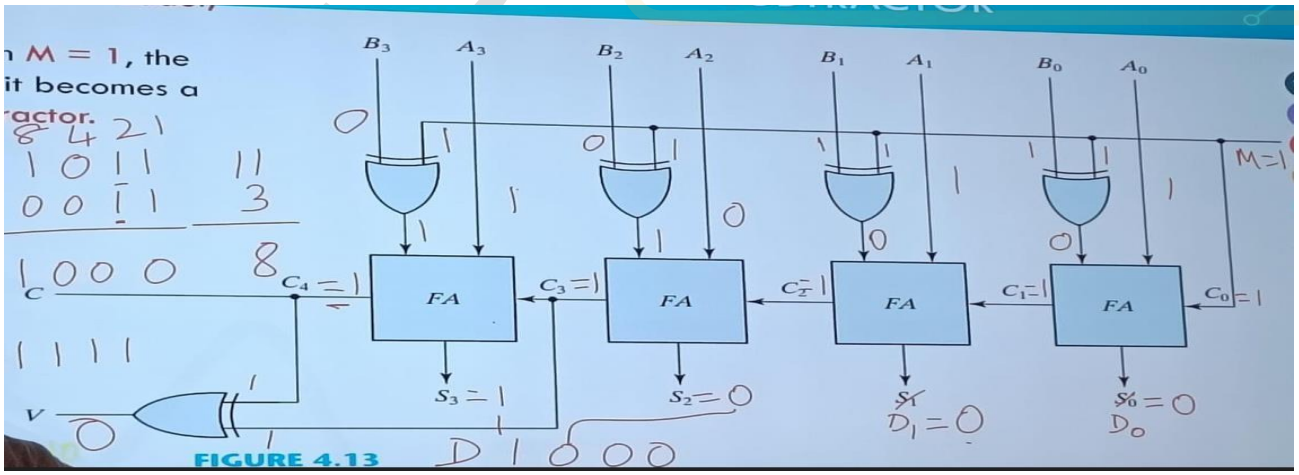


FIGURE 4.13
Four-bit adder–subtractor (with overflow detection)

Binary Addition Example:



Binary Subtraction Example:

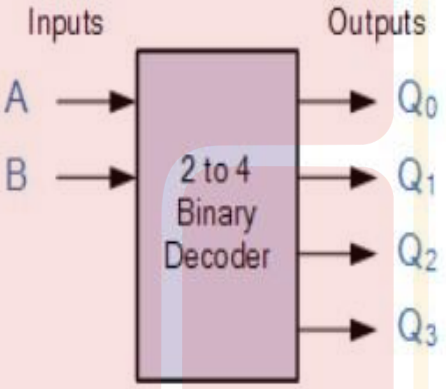


DECODERS

- A Decoder is a combinational circuit that converts binary information from n input lines to a maximum of 2^n unique output lines.
- The decoders presented here are called n -to- m -line decoders, where m … 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.

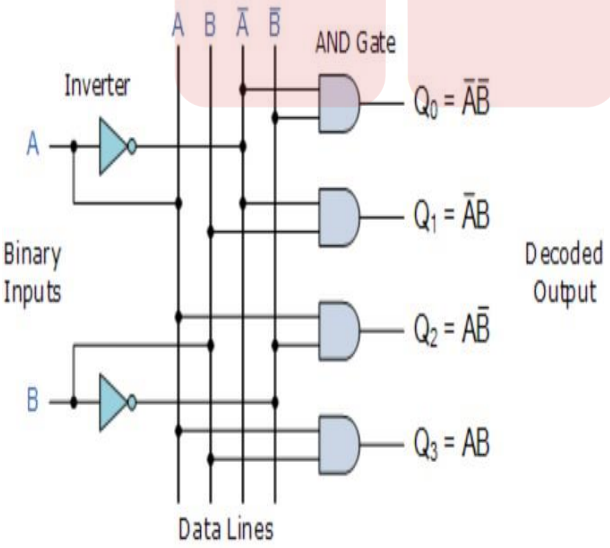
2:4 decoder (1 of 4 decoder)

A 2 to 4 decoder is a combinational logic circuit that takes two input lines, typically labeled A and B, and generates four output lines, usually labeled Q0, Q1, Q2, and Q3. The decoder analyzes the input combination and activates the corresponding output line



Truth Table

A	B	Q_0	Q_1	Q_2	Q_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1



3:8 Decoder

- A 3 to 8 decoder has three inputs (x,y,z) and eight outputs (D0 to D7).
- Based on the 3 inputs one of the eight outputs is selected.
- The truth table for 3 to 8 decoder is shown in the below table.
- From the truth table, it is seen that only one of eight outputs (D0 to D7) is selected based on three select inputs.
- From the truth table, the logic expressions for outputs can be written as follows:

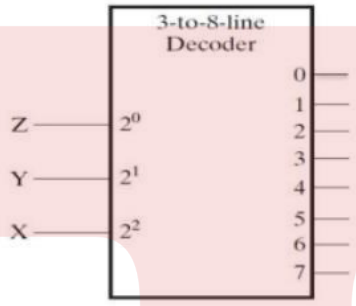


Table 4.6
Truth Table of a Three-to-Eight-Line Decoder

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

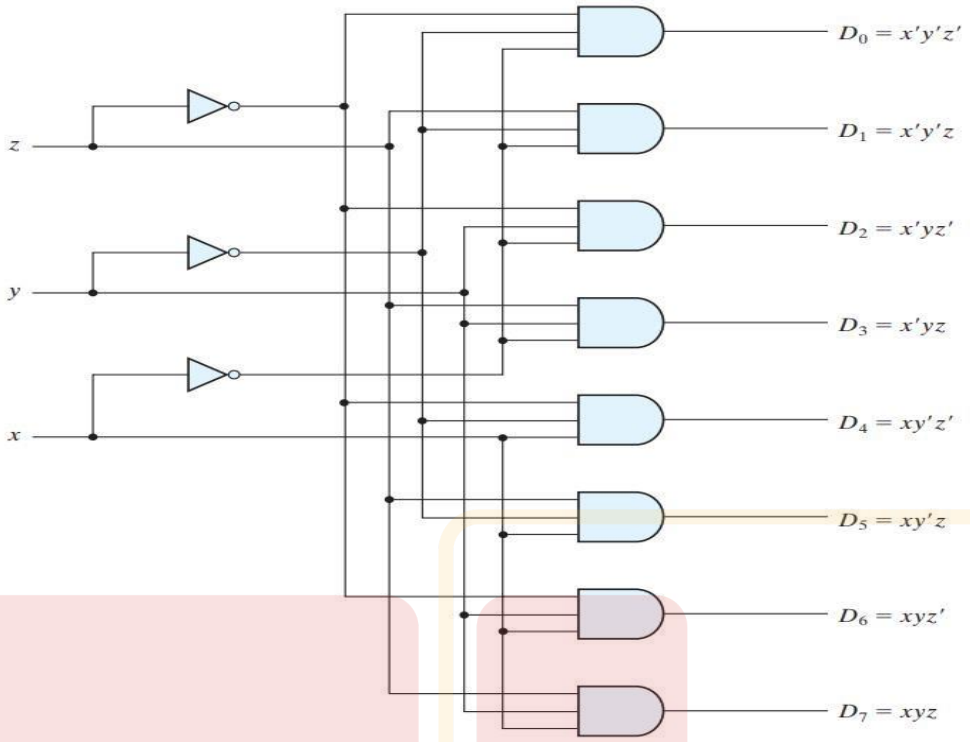


FIGURE 4.18
Three-to-eight-line decoder

Decoders with enable inputs can be connected together to form a larger decoder circuit.

Implement 4:16 decoder using 2 3:8 decoder.

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.**

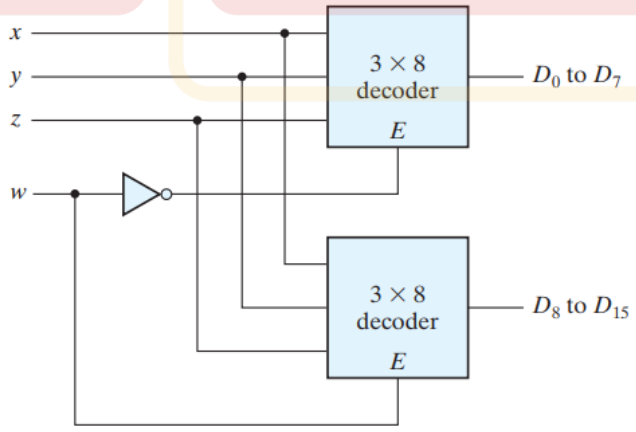


FIGURE 4.20
4 × 16 decoder constructed with two 3 × 8 decoders

Combinational Logic Implementation

Implement the following boolean function using 3:8 decoder

S(x, y, z) = ∑(1, 2, 4, 7)

C(x, y, z) = ∑(3, 5, 6, 7)

Since there are three inputs and a total of eight minterms, we need a three-to-eight-line decoder.

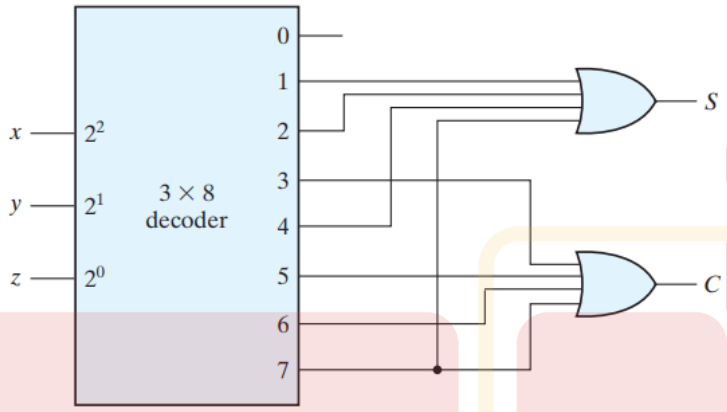
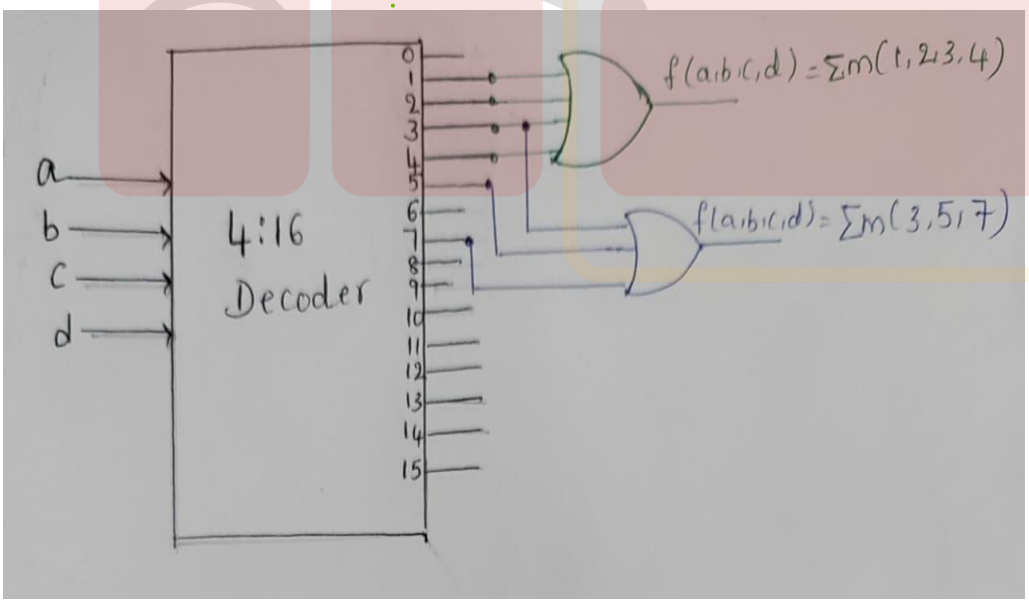


FIGURE 4.21
Implementation of a full adder with a decoder

The decoder generates the eight minterms for x , y , and z . The OR gate for output S forms the logical sum of minterms 1, 2, 4, and 7. The OR gate for output C forms the logical sum of minterms 3, 5, 6, and 7.

Exemplify(Implement) the following function using 3:8 decoder

i) f(a,b, c,d)=∑m (L,2, 3,4)
ii) f(a,b, c, d)=∑m (3,5,7)

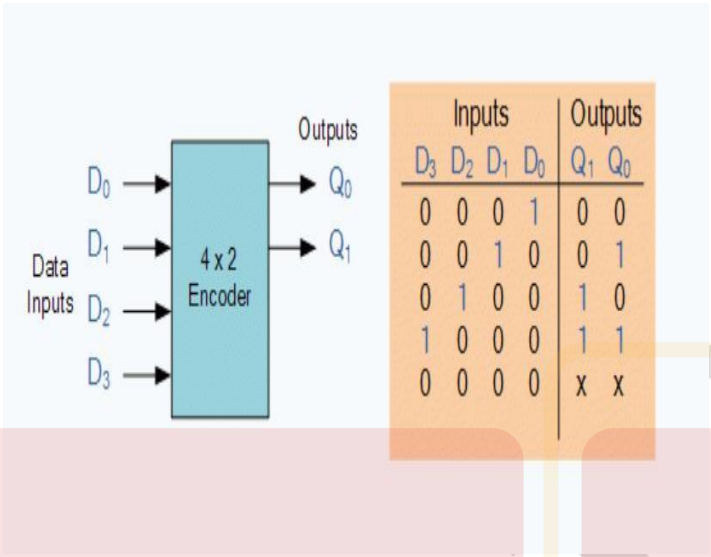


Encoder

- 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.
- 4:2 Encoder(n=2)

4:2 Encoder



Inputs				Outputs	
D_3	D_2	D_1	D_0	Q_1	Q_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1
0	0	0	0	x	x

8:3 Encoder

- an encoder is the octal-to-binary encoder whose truth table is given in Table 4.7
- 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.

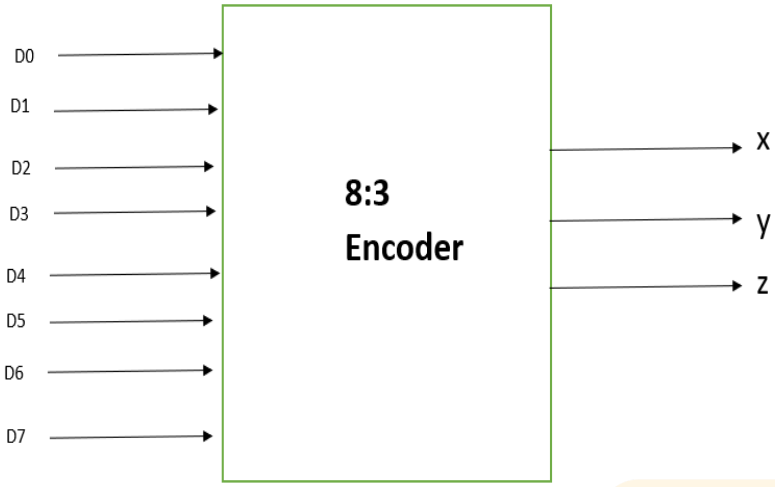


Table 4.7
Truth Table of an Octal-to-Binary Encoder

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

$$z = D_1 + D_3 + D_5 + D_7$$
$$y = D_2 + D_3 + D_6 + D_7$$
$$x = D_4 + D_5 + D_6 + D_7$$

- The encoder defined in Table 4.7 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 D3 and D6 are 1 simultaneously, the output of the encoder will be 111 because all three outputs are equal to 1.
- To resolve this ambiguity, encoder circuits must establish an input priority to ensure that only one input is encoded.
- 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. If we establish a higher priority for inputs with higher subscript numbers, and if both D3 and D6 are 1 at the same time, the output will be 110 because D6 has higher priority than D3. 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 D0 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

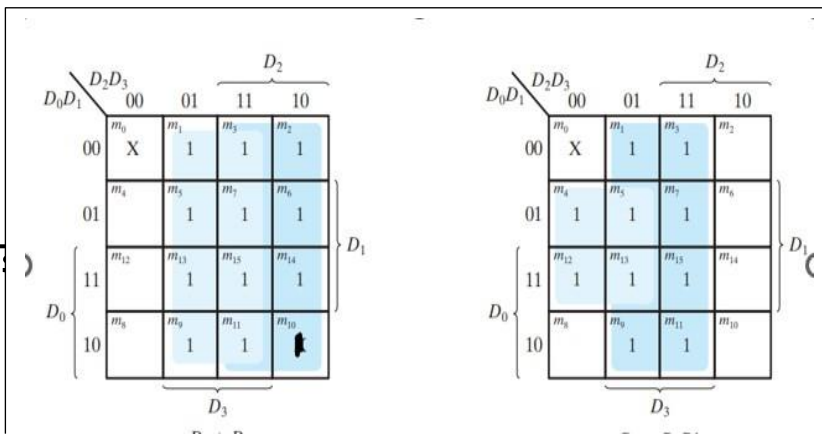
- A priority encoder is an encoder circuit that includes the priority function.
- The operation of the priority encoder is such that if two or more inputs are equal to 1 at the same time, the input having the highest priority will take precedence.
- The truth table of a four-input priority encoder is given in Table 4.8

Table 4.8
Truth Table of a Priority Encoder

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

- 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.
- Input D3 has the highest priority, so, regardless of the values of the other inputs, when this input is 1, the output for xy is 11 (binary 3).
- D2 has the next priority level. The output is 10 if D2 = 1, provided that D3 = 0, regardless of the values of the other two lower priority inputs. The output for D1 is generated only if higher priority inputs are 0.

D0	D1	D2	D3	X	Y	V
0	0	0	0	X	X	0
0	0	0	1	1	1	1
0	0	1	0	1	0	1



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

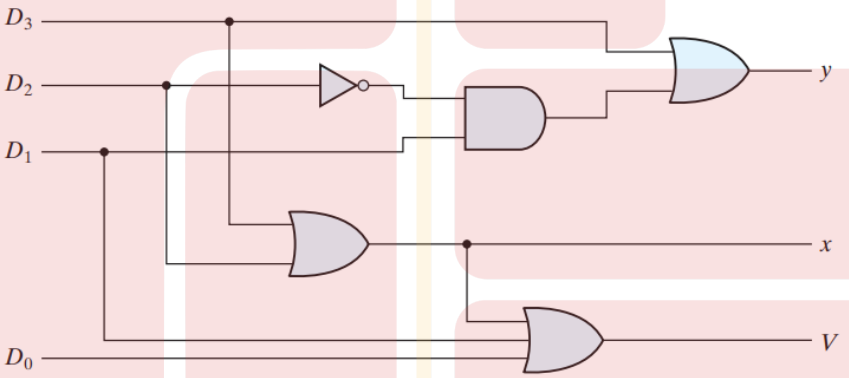
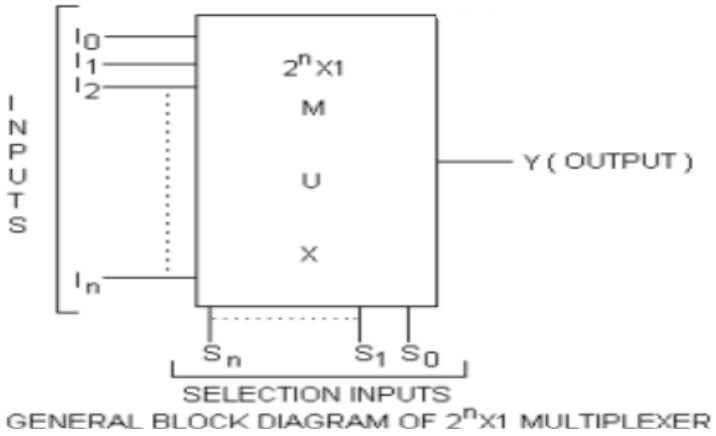


FIGURE 4.23
Four-input priority encoder

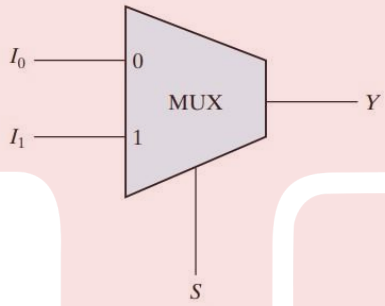
Multiplexer

- 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.



GENERAL BLOCK DIAGRAM OF 2^nX1 MULTIPLEXER

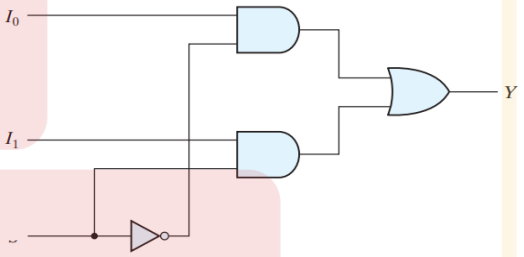
Design 2:1 Multiplexer



(b) Block diagram

S	Y
0	
1	

Truth Table



(a) Logic diagram

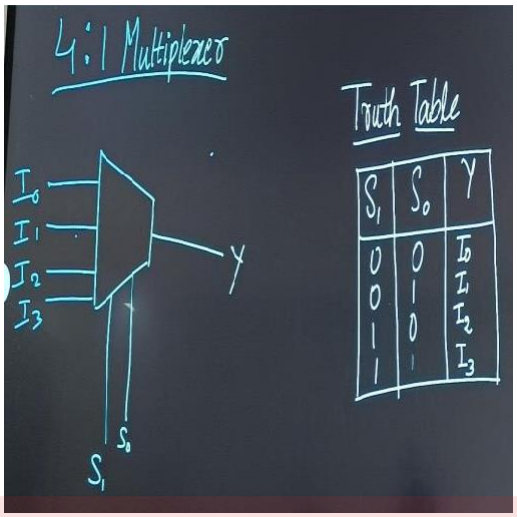
FIGURE 4.24
Two-to-one-line multiplexer

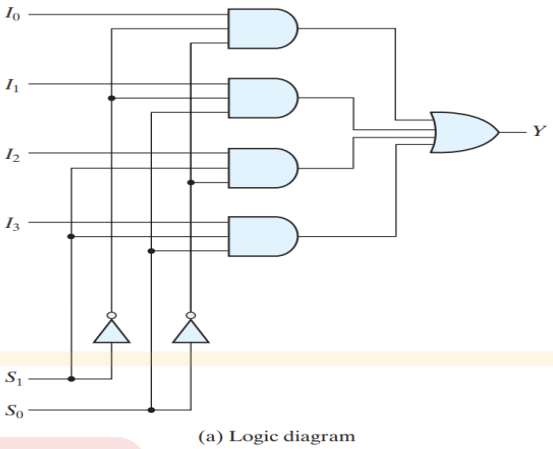
$$y = s'I_0 + sI_1$$

Boolean Expression

A 2-to-1 multiplexer consists of two inputs I0 and I1, one select input S and one output Y. Depending on the select signal, the output is connected to either of the inputs. Since there are two input signals, only two ways are possible to connect the inputs to the outputs, so one select is needed to do these operations.

4:1 Multiplexer



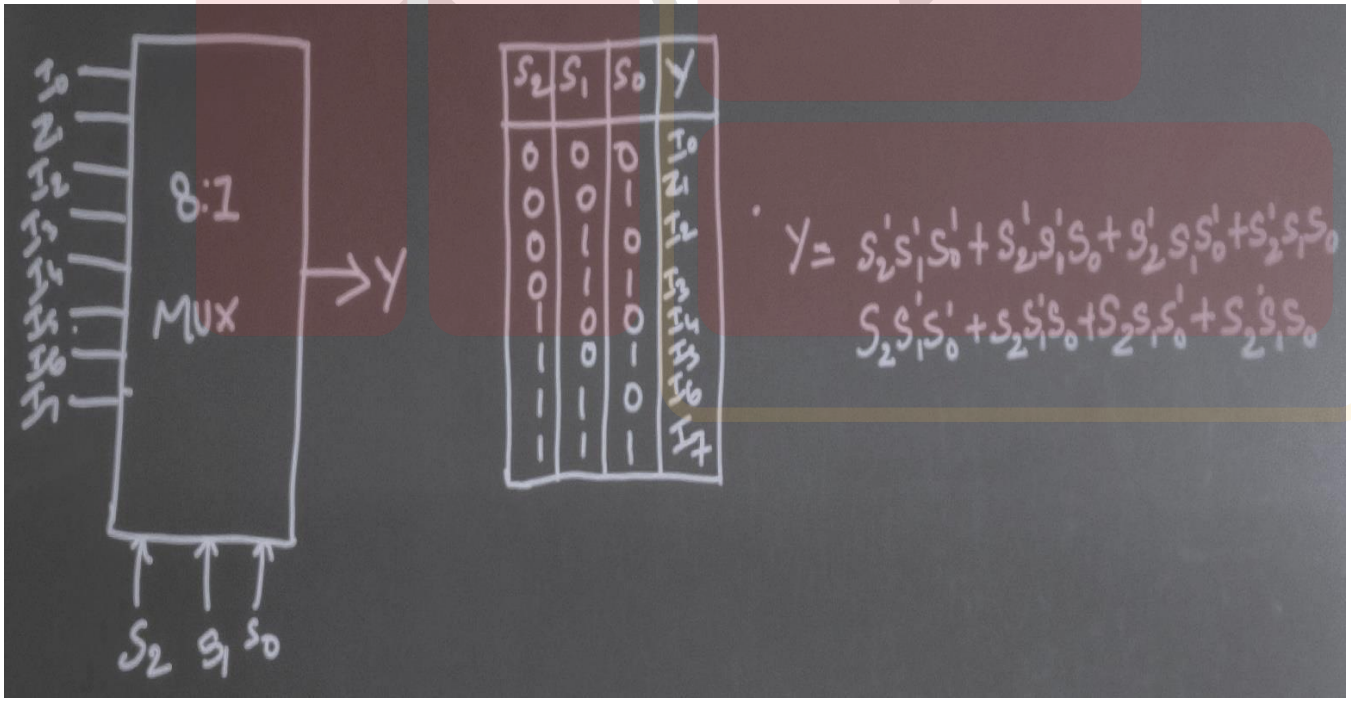


(a) Logic diagram

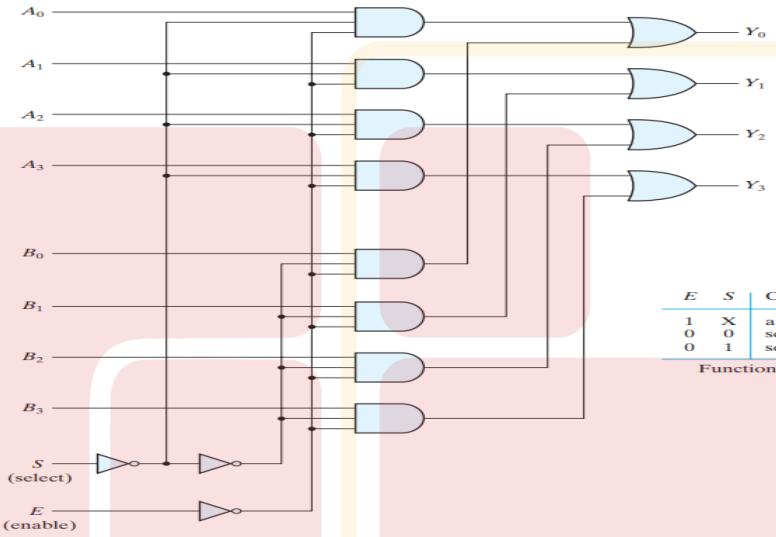
Above figures represents block diagram ,truth table and implementation using basic gates of 4:1 multiplexer.

4x1 Multiplexer has four data inputs I0, I1, I2 & I3, two selection lines S0 & S1 and one output Y. One of these 4 inputs will be connected to the output based on the combination of inputs present at these two selection lines.

8:1 multiplexer



- **Multiplexer circuits can be combined with common selection inputs to provide multiple-bit selection logic**. As an illustration, a **quadruple 2-to-1-line multiplexer** is shown in Fig. 4.26 . The circuit has four multiplexers, each capable of selecting one of two input lines. Output Y0 can be selected to come from either input A0 or input B0. Similarly, output Y1 may have the value of A1 or B1, and so on. Input selection line S selects one of the lines in each of the four multiplexers. The enable input E must be active (i.e., asserted) for normal operation.
- As shown in the function table, the unit is enabled when E = 0. Then, if S = 0, the four A inputs have a path to the four outputs. If, by contrast, S = 1, the four B inputs are applied to the outputs. The outputs have all 0's when E = 1, regardless of the value of S .

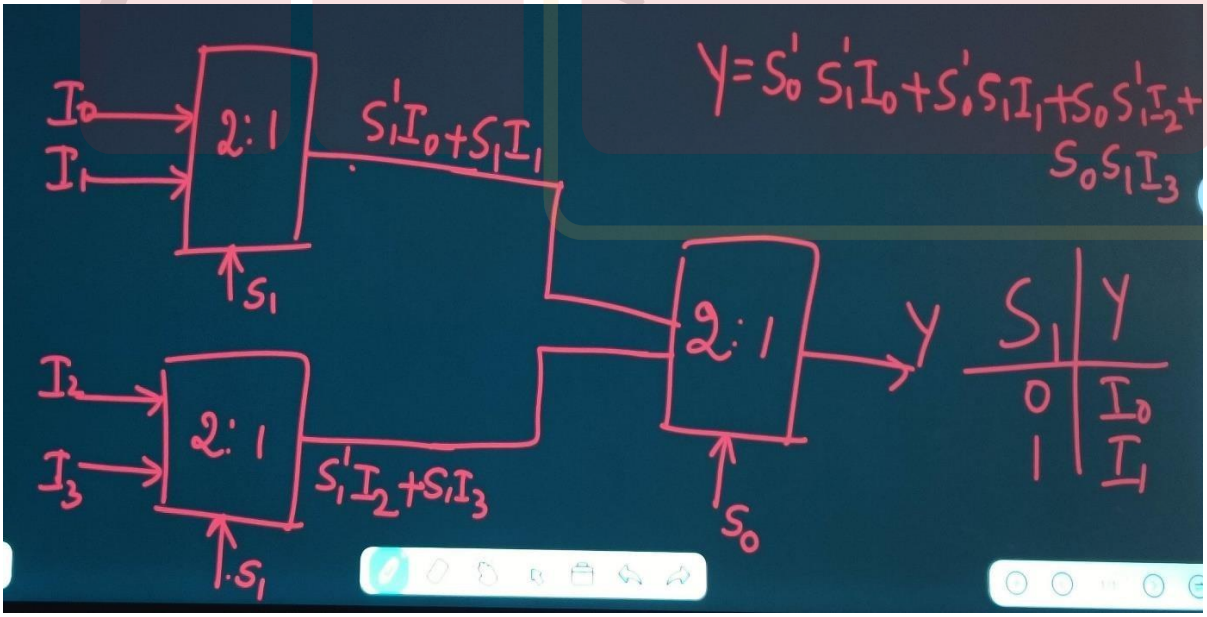


E	S	Output Y
1	X	all 0's
0	0	select A
0	1	select B

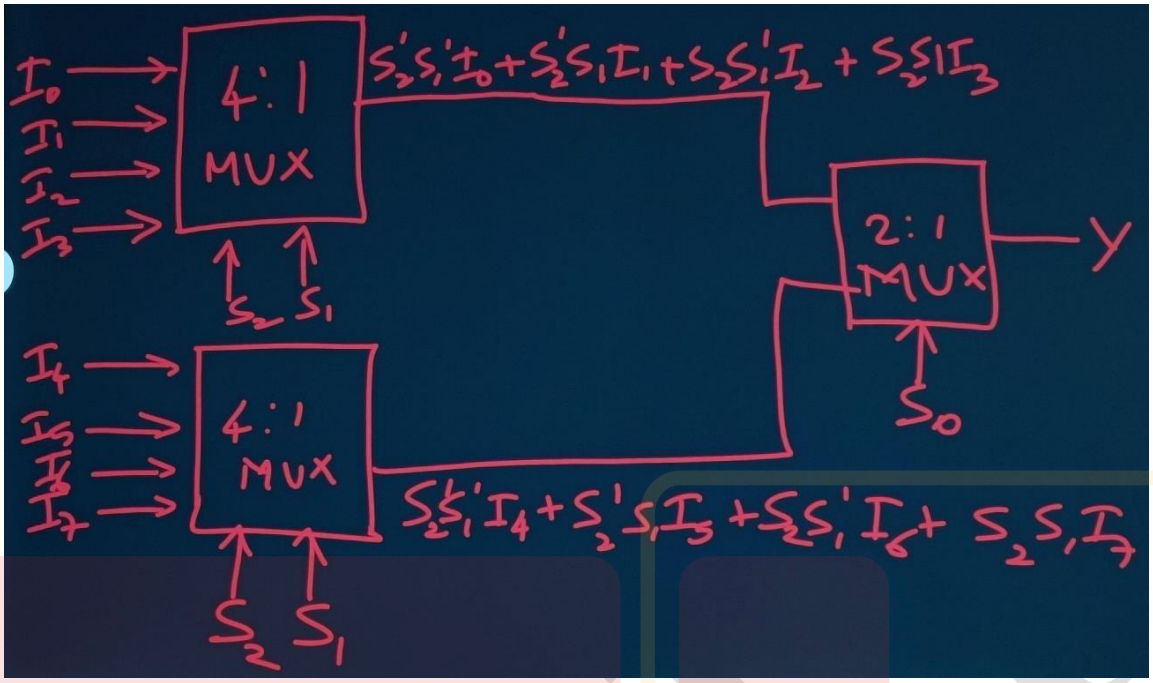
Function table

- **FIGURE 4.26**
  Quadruple two-to-one-line multiplexer

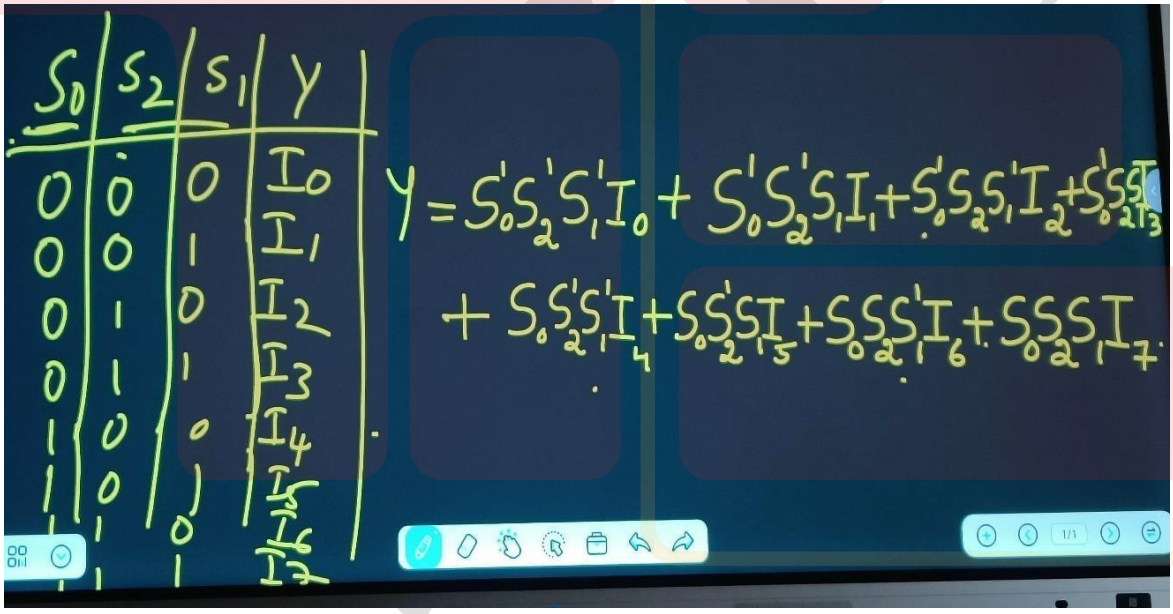
Design 4:1 MUX using only 2:1 MUX



Implement 8:1 Mux using 4:1mux and 2:1mux



8:1 MUX Truth Table



Implement using multiplexer F (x, y, z) = (1, 2, 6, 7)

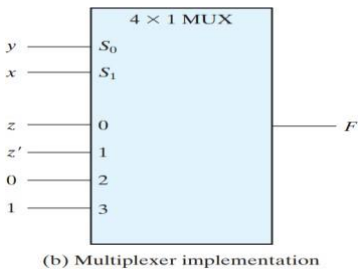
n=3

n- 1=2 select lines

2^{n-1} =4 data inputs

x	y	z	F	
0	0	0	0	$F = z$
0	0	1	1	
0	1	0	1	$F = z'$
0	1	1	0	
1	0	0	0	$F = 0$
1	0	1	0	
1	1	0	1	$F = 1$
1	1	1	1	

(a) Truth table



(b) Multiplexer implementation

Implement using multiplexer F (A, B, C, D) = (1, 3, 4, 11, 12, 13, 14, 15)

A	B	C	D	F	
0	0	0	0	0	$F = D$
0	0	0	1	1	
0	0	1	0	0	$F = D$
0	0	1	1	1	
0	1	0	0	1	$F = D'$
0	1	0	1	0	
0	1	1	0	0	$F = 0$
0	1	1	1	0	
1	0	0	0	0	$F = 0$
1	0	0	1	0	
1	0	1	0	0	$F = D$
1	0	1	1	1	
1	1	0	0	1	$F = 1$
1	1	0	1	1	
1	1	1	0	1	$F = 1$
1	1	1	1	1	

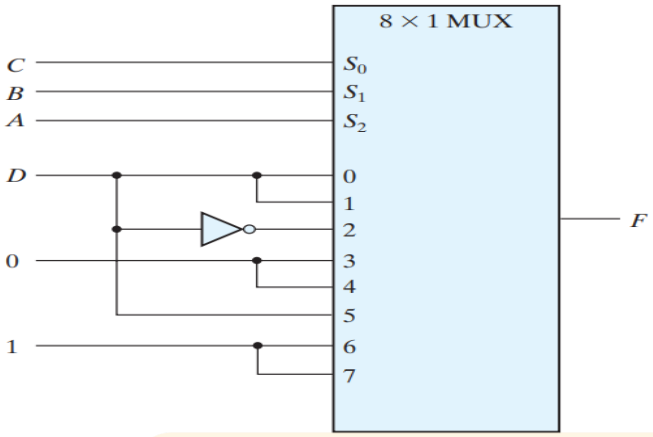
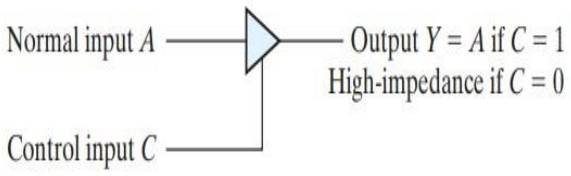


FIGURE 4.28
Implementing a four-input function with a multiplexer

Three-State Gates

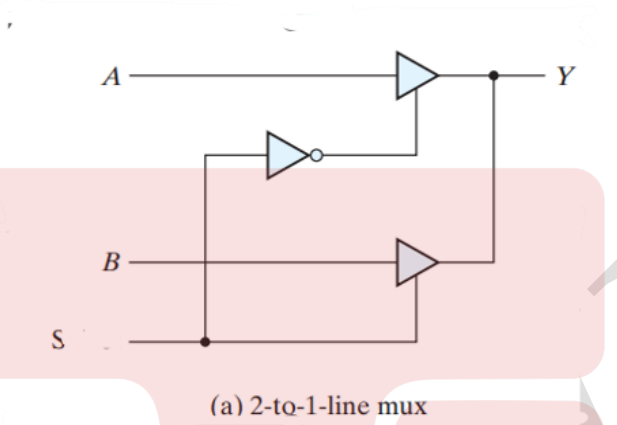
- A multiplexer can be constructed with three-state gates—digital circuits that exhibit three states.
- Two of the states are signals equivalent to logic 1 and logic 0 as in a conventional gate.
- The third state is a high-impedance state in which
- (1) the logic behaves like an open circuit, which means that the output appears to be disconnected,
- (2) the circuit has no logic significance, and
- (3) the circuit connected to the output of the three-state gate is not affected by the inputs to the gate. Three-state gates may perform any conventional logic, such as AND or NAND. However, the one most commonly used is the buffer gate.
- The graphic symbol for a three-state buffer gate is



The buffer has a normal input, an output, and a control input that determines the state of the output. When the control input is equal to 1, the output is enabled and the gate behaves like a conventional buffer, with the output equal to the normal input. When the control input is 0, the output is disabled and the gate goes to a high-impedance state, regardless of the value in the normal input. The high-impedance state of a three-state gate provides a special

feature not available in other gates. Because of this feature, a large number of three-state gate outputs can be connected with wires to form a common line without endangering loading effects.

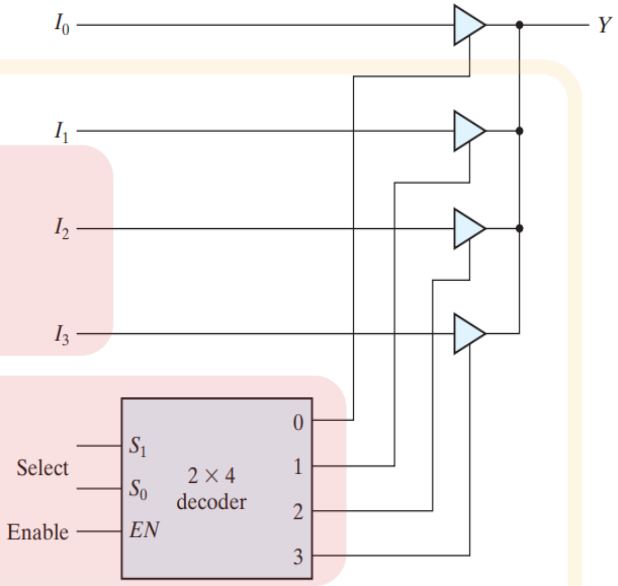
- The construction of multiplexers with three-state buffers is demonstrated in Fig. 4.30 . Figure 4.30(a) shows the construction of a two-to-one-line multiplexer with 2 three-state buffers and an inverter. The two outputs are connected together to form a single output line. (Note that this type of connection cannot be made with gates that do not have three-state outputs.) When the select input is 0, the upper buffer is enabled by its control input and the lower buffer is disabled. Output Y is then equal to input A . When the select input is 1, the lower buffer is enabled and Y is equal to B .



(a) 2-to-1-line mux

S	Y
0	A
1	B

Y=AS'+BS



(b) 4-to-1-line mux

HDL models of combinational circuits

- The logic of a module can be described in any one (or a combination) of the following modeling styles:

- **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.** Gate-level modeling using instantiations of predefined and user-defined primitive gates

- **Dataflow modeling** is used mostly for **describing the Boolean equations of combinational logic,** Dataflow modeling using continuous assignment statements with the keyword assign.

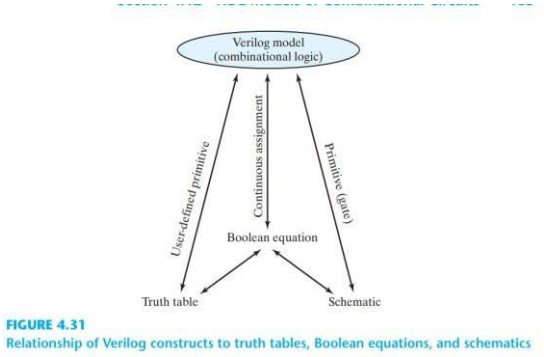


FIGURE 4.31
Relationship of Verilog constructs to truth tables, Boolean equations, and schematics

Table 4.10
Some Verilog HDL Operators

Symbol	Operation	Symbol	Operation
+	binary addition		
–	binary subtraction		
&	bitwise AND	&&	logical AND
|	bitwise OR	||	logical OR
^	bitwise XOR		
~	bitwise NOT	!	logical NOT
= =	equality		
>	greater than		
<	less than		
{}	concatenation		
?:	conditional		

Write a Verilog Program For Binary Adder(4bit)

HDL (Dataflow: Four-Bit Adder)

```
module binary_adder (
  output [3: 0]        Sum,
  output               C_out,
  input [3: 0]         A, B,
  input                C_in
);
  assign {C_out, Sum} = A + B + C_in;
endmodule
```

Write a Verilog code for 2:1 mux(multiplexer)

Using **cond itional operator**

condition ? true-expression : false-expression;

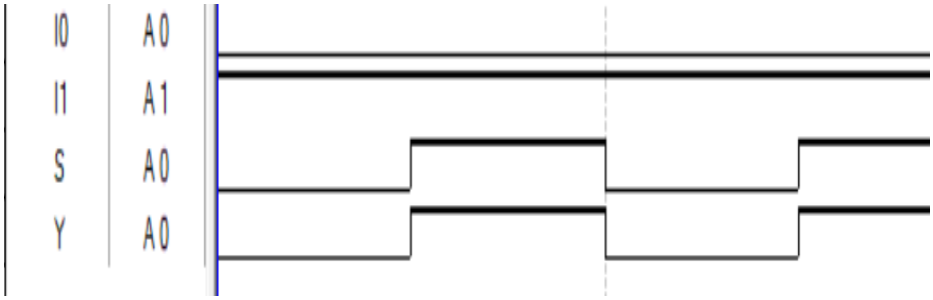
s	y
0	I_0
1	I_1

$$y = s'I_0 + sI_1$$

```
module mux2_1( I0,I1,S,Y);
input S ;
input I0,I1 ;
output Y ;
assign Y=S?I1:I0;
endmodule
```

Using Data flow Model

```
module mux2_1(S,I,Y);
input S;
input [1:0]I;
output Y;
assign Y=(~S&I[0]|S&I[1]);
endmodule
```



Behavioral modelling for 2:1 Mux

Using Case Statement

```
module mux2_1( I0,I1,S,Y);
input I0,I1 ;
input S ;
output Y ;
reg Y;
always @ (S or I0 or I1)
begin
case (S)
0: Y=I0  ;
1: Y=I1  ;
endcase
end
endmodule
```

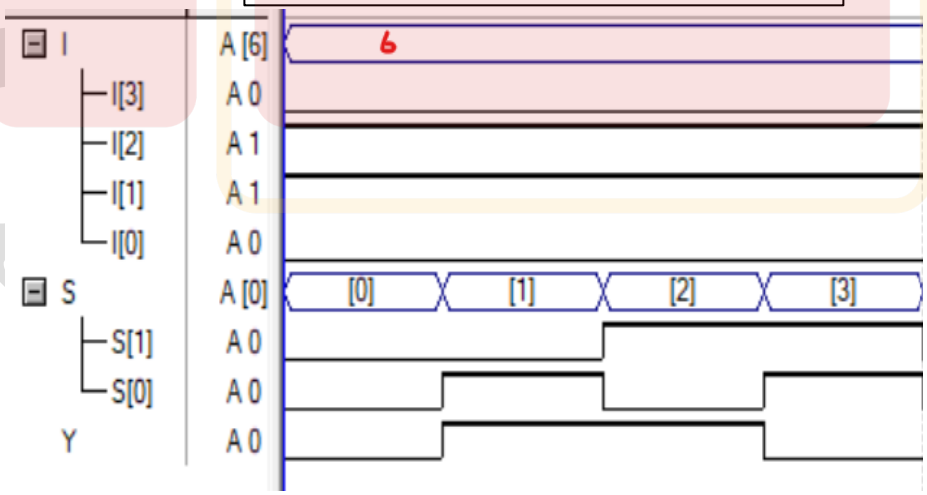
using If else statement

```
module mux2_1( I0,I1,S,Y);
input I0,I1 ;
input S ;
output Y ;
reg Y;
always @ (S ,I0 , I1)
begin
  if(S==0)
     Y=I0  ;
else Y=I1  ;
end
endmodule
```

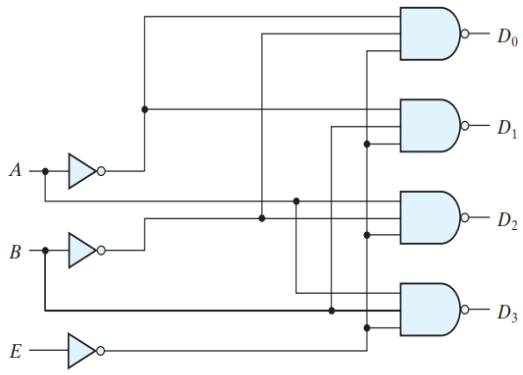
Write Verilog program for 4:1mux using CASE STATEMENT

```
module mux4_1(I,S,Y);
input [1:0] S;
input [3:0]I;
 output Y;
 reg   Y;
always @ (I,S)
begin
 case (S)
 0:Y= I[0];
 1:Y= I[1];
 2:Y= I[2];
 3:Y= I[3];
 endcase
 end
endmodule
```

Timing Diagram



Write a Verilog code for below figure



(a) Logic diagram

E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	1

HDL Example 4.1 (Two-to-Four-Line Decoder)

```
// Gate-level description of two-to-four-line decoder
// Refer to Fig. 4.19 with symbol E replaced by enable, for clarity.

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
```

Sequential Logic

- Sequential logic refers to a type of digital logic circuit that uses memory elements to store information.
- 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.
- a sequential circuit is specified by a time sequence of inputs, outputs, and internal states.

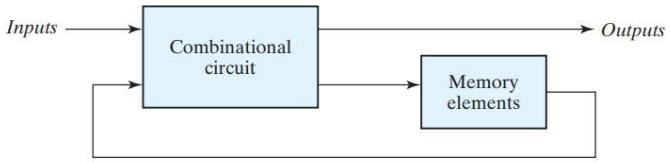


FIGURE 5.1
Block diagram of sequential circuit

Differentiate between combinational logic and sequential logic

	Combinational Logic Circuits	Sequential Logic Circuits
Definition	At any instant of time, the output is only dependent on the current state of the inputs.	At any instant of time, the output is determined by inputs and previous outputs.
Time dependency	Time is not an important parameter.	Time is an important parameter. For timing and synchronizing of different circuit elements, a clock signal is necessary.
Memory	The output is solely dependent on inputs only. No need for memory.	Memory is required to store the previous state of the system.
Design	Easy to design and implement with the help of basic logic gates.	The design of these systems requires basic logic gates and flip flops.
Feedback	There is no feedback.	There is at least one memory element in the feedback path.
Hardware & cost	They are easier to implement but costly, due to hardware. Their implementation requires more hardware.	They are difficult to implement but less costly than sequential circuits.
Speed	They are faster since all inputs are applied at the same time.	They are slower, because of the secondary inputs. So, there is a delay in between inputs. And the output is gated by a clock signal.

- 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.

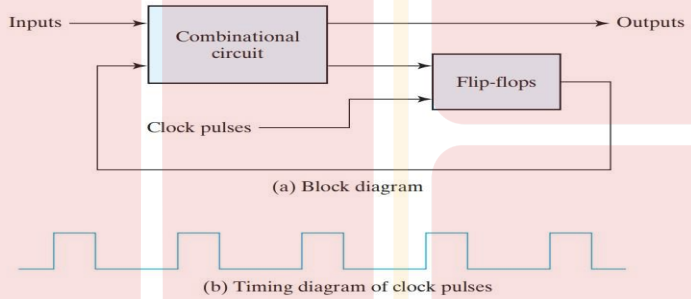


FIGURE 5.2
Synchronous clocked sequential circuit

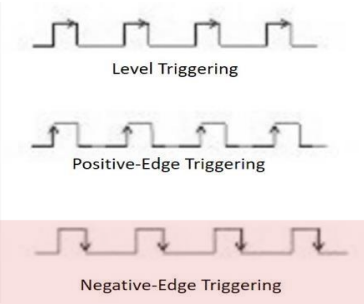
Storage Elements:

1)Latches:

- Latches are digital circuits that serve as basic building blocks in the construction of sequential logic circuits.
- They are **bistable**, meaning they **have two stable states** and can be used to store binary information. Latches are often used for temporary storage of data within a digital system.
- There are several types of latches, with the most common being the

1)SR latch (Set-Reset latch), 2)D latch (Data latch),3) JK latch.

- 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.

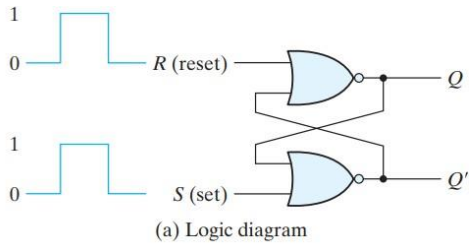


SR Latch (Set-Reset Latch):

- The SR latch has two inputs, S (Set) and R (Reset).It has two outputs, Q and ~Q (complement of Q).
- When S is asserted, Q is set to 1, and when R is asserted, Q is reset to 0.The SR latch is sensitive to the input conditions, and having both S and R asserted simultaneously can lead to unpredictable behavior.

S	R	Q	Q'
0	0	NO CHANGE (Previous output)	
0	1	0	1
1	0	1	0
1	1	FORBIDDEN	

SR Latch with nor gates



(a) Logic diagram

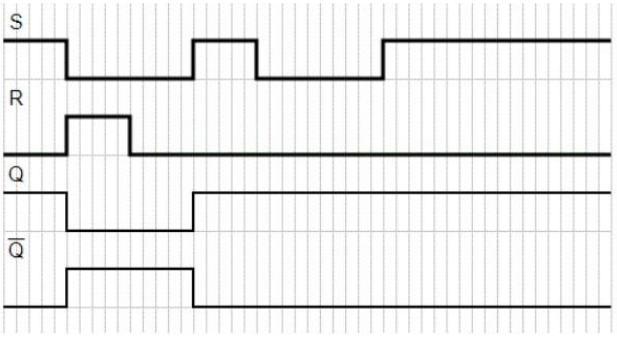
S	R	Q	Q'	
1	0	1	0	
0	0	1	0	(after $S = 1, R = 0$)
0	1	0	1	
0	0	0	1	(after $S = 0, R = 1$)
1	1	0	0	(forbidden)

(b) Function table

FIGURE 5.3
SR latch with NOR gates

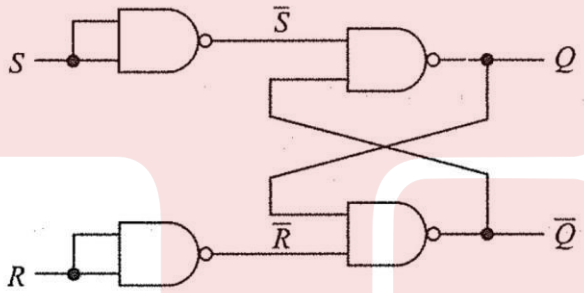
where S and R stand for set and reset. It can be constructed from a pair of cross-coupled NOR logic gates. The stored bit is present on the output marked Q.

While the S and R inputs are both low, feedback maintains the Q and $\overline{Q}$ outputs in a constant state, with Q the complement of Q. If S (Set) is pulsed high while R (Reset) is held low, then the Q output is forced high, and stays high when S returns to low; similarly, if R is pulsed high while S is held low, then the Q output is forced low, and stays low when R returns to low.



Timing Diagram of SR latch

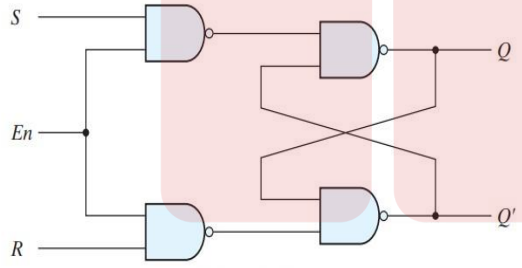
SR latch with NAND gates



S	R	Q	Q′	
1	0	1	0	
0	0	1	0	(after $S = 1, R = 0$)
0	1	0	1	
0	0	0	1	(after $S = 0, R = 1$)
1	1	0	0	(forbidden)

(b) Function table

SR latch with control input



(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

FIGURE 5.5
SR latch with control input

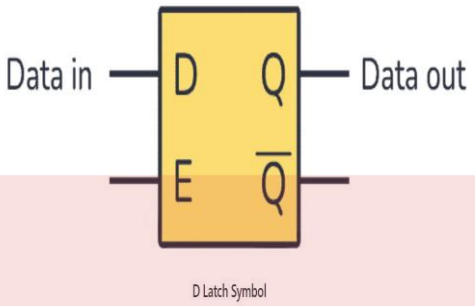
It consists of the basic SR latch and two additional NAND gates. The control input En acts as an enable signal for the other two inputs. The outputs of the NAND gates stay at the logic-1 level as long as the enable signal remains at 0. This is the quiescent condition for the SR latch. When the enable 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.

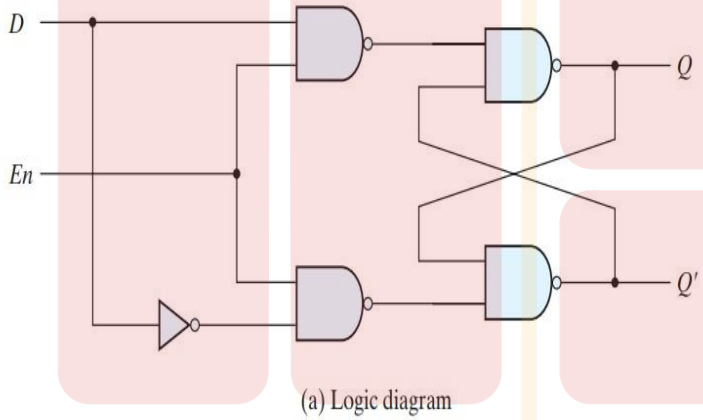
D latch(transparent latch)

A D latch can store a bit value, either 1 or 0. When its Enable pin is HIGH, the value on the D pin will be stored on the Q output.

The D Latch is a logic circuit most frequently used for storing data in digital systems. It is based on the S-R latch, but it doesn't have an "undefined" or "invalid" state problem.



E	D	Q	Description
0	X	Q	Memory (no change)
1	0	0	Reset Q to 0
1	1	1	Set Q to 1



(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

FIGURE 5.6
D 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. 5.6 . 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.

The graphic symbols for the various latches are shown in Fig. 5.7 . A latch is designated by a rectangular block with inputs on the left and outputs on the right. One output designates the normal output, and the other (with the bubble designation) designates the complement output

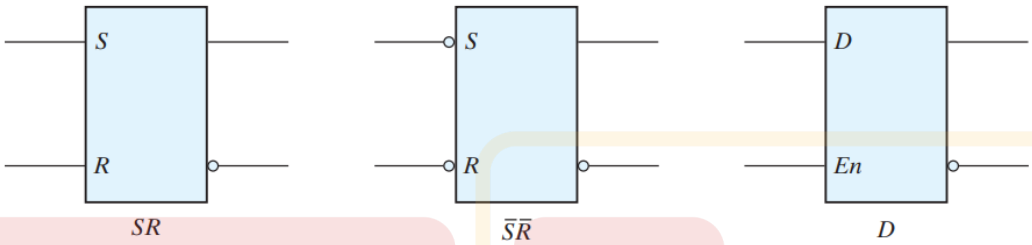


FIGURE 5.7
Graphic symbols for latches

STORAGE ELEMENTS : FLIP - FLOPS

- Flip-flops are fundamental building blocks in digital electronics and sequential logic circuits.
- They are bistable multivibrators, like latches, but they are edge-triggered and use a clock signal to control the timing of state changes.
- Flip-flops are widely used for storing binary information in electronic systems.

Edge triggered DFF

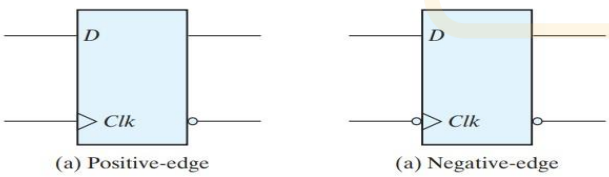


FIGURE 5.11
Graphic symbol for edge-triggered *D* flip-flop

Table of truth:

clk	D	Q	$\overline{Q}$
0	0	Q	$\overline{Q}$
0	1	Q	$\overline{Q}$
1	0	0	1
1	1	1	0

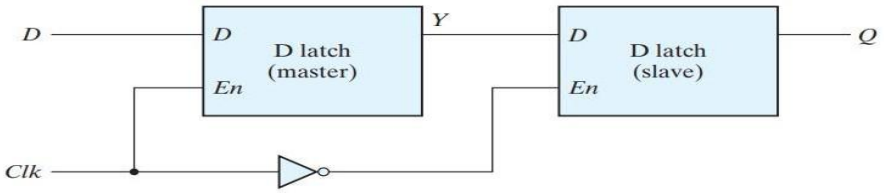


FIGURE 5.9
Master–slave *D* flip-flop

The construction of a D flip-flop with two D latches and an inverter is shown in Fig. 5.9 . The first latch is called the master and the second the slave. The circuit samples the D input and changes its output Q only at the negative edge of the synchronizing or controlling clock (designated as Clk). When the clock is 0, the output of the inverter is 1. The slave latch is enabled, and its output Q is equal to the master output Y . The master latch is disabled because Clk = 0. When the input pulse changes to the logic-1 level, the data from the external D input are transferred to the master. The slave, however, is disabled as long as the clock remains at the 1 level, because its enable input is equal to 0. Any change in the input changes the master output at Y, but cannot affect the slave output. When the clock pulse returns to 0, the master is disabled and is isolated from the D input. At the same time, the slave is enabled and the value of Y is transferred to the output of the flip-flop at Q . Thus, a change in the output of the flip-flop can be triggered only by and during the transition of the clock from 1 to 0.

Comparison between Latch and Flipflop

LATCH	FLIP – FLOP
Latches do not require clock signal.	Flip – flops have clock signals
A latch is an asynchronous device.	A flip – flop is a synchronous device.
Latches are transparent devices i.e. when they are enabled, the output changes immediately if the input changes.	A transition from low to high or high to low of the clock signal will cause the flip – flop to either change its output or retain it depending on the input signal.
A latch is a Level Sensitive device (Level Triggering is involved).	A flip – flop is an edge sensitive device (Edge Triggering is involved).
Latches are simpler to design as there is no clock signal (no careful routing of clock signal is required).	When compare to latches, flip – flops are more complex to design as they have clock signal and it has to be carefully routed. This is because all the flip – flops in a design should have a clock signal and the delay in the clock reaching each flip – flop must be minimum or negligible.
The operation of a latch is faster as they do not have to wait for any clock signal.	Flip - flops are comparatively slower than latches due to clock signal.
The power requirement of a latch is less.	Power requirement of a flip – flop is more.
A latch works based on the enable signal.	A flip – flop works based on the clock signal.

construction of an **positive edge-triggered D flip-flop uses three SR latches**

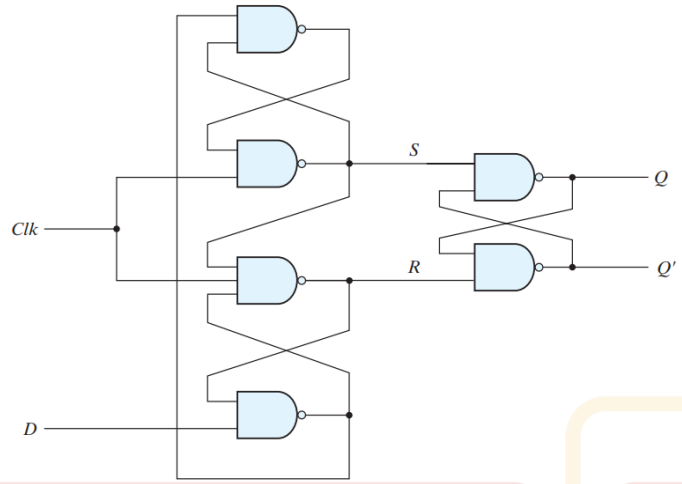


FIGURE 5.10
D-type positive-edge-triggered flip-flop

Clk	D	S	R	Q	Q'
				0	1
				Assume(previous output)	
0	0	1	1	0	1
				No Change	
0	1	1	1	No Change	
1	1	0	1	1	0
0	0	1	1	No change	
1	0	1	0	0	1

JK FLIPFLOP

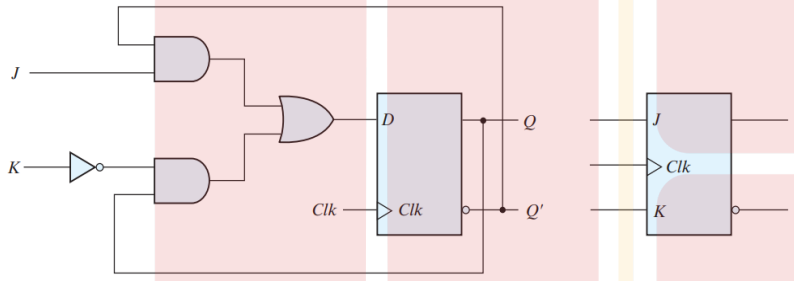


FIGURE 5.12
JK flip-flop

Table 5.1
Flip-Flop Characteristic Tables

***JK* Flip-Flop**

J	*K*	*Q*(*t* + 1)	
0	0	*Q*(*t*)	No change
0	1	0	Reset
1	0	1	Set
1	1	*Q*′(*t*)	Complement

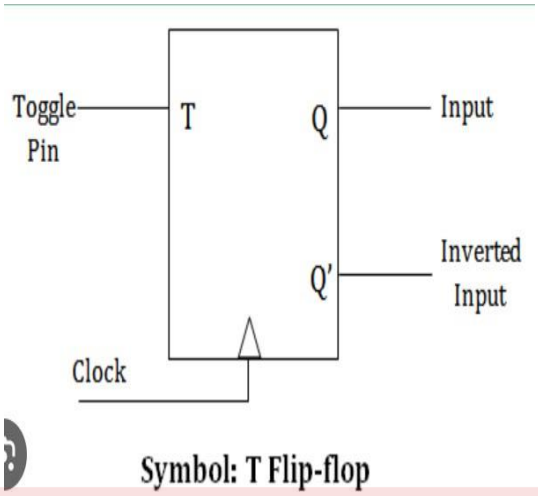
When J = 1 and K = 0, D = Q' + Q = 1, so the next clock edge sets the output to 1.

When J = 0 and K = 1, D = 0, so the next clock edge resets the output to 0.

When both J = K = 1 and D = Q, the next clock edge complements the output.

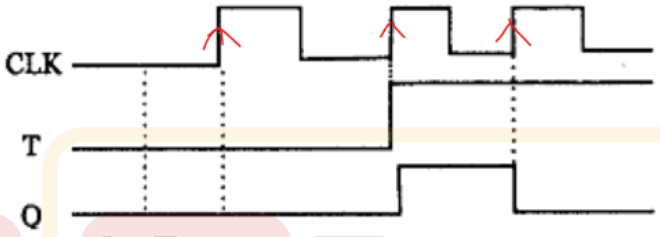
When both J = K = 0 and D = Q, the clock edge leaves the output unchanged.

T Flipflop



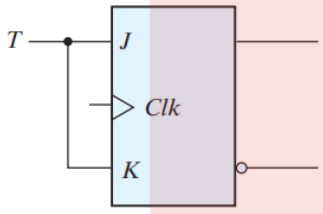
Symbol: T Flip-flop

Inputs		Outputs	
CLK	T	Q_{n+1}	Action
0	X	Qn	No change
1	0	Qn	No change
1	1	$\overline{Qn}$	Toggle



T Flipflop using JK Flipflop

T = 0 (J = K = 0), a clock edge does not change the output. When T = 1 (J = K = 1), a clock edge complements the output. The complementing flip-flop is useful for designing binary counters.



(a) From *JK* flip-flop

FIGURE 5.13
T flip-flop

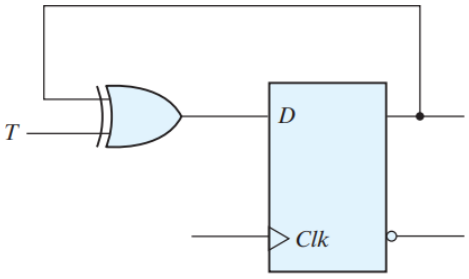
Clk	T	Q	Q'
↑	0	No Change	
↑	1	toggle	

Implementation of TFF using DFF

The T flip-flop can be constructed with a D flip-flop and an exclusive-OR gate as shown in Fig. (b). The expression for the D input is D = T ⊕ Q = T'Q + TQ' When T = 0, D = Q and there is no change in the output. When T = 1, D = Q' and the output complements.

D=T^Q

D=T'Q+TQ'



(b) From *D* flip-flop

- Characteristic tablesA characteristic table defines the logical properties of a flip-flop by describing its operation in tabular form. They define the next state (i.e., the state that results from a clock transition) as a function of the inputs and the present state
- Q(t) denotes the state of the flip-flop immediately before the clock edge, and
- Q(t + 1) denotes the state that results from the clock transition.

Table 5.1
Flip-Flop Characteristic Tables

JK Flip-Flop

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

D Flip-Flop

D	Q(t + 1)	
0	0	Reset
1	1	Set

T Flip-Flop

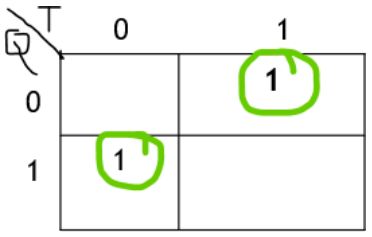
T	Q(t + 1)	
0	$Q(t)$	No change
1	$Q'(t)$	Complement

Characteristic equation

- It is the Boolean expression in terms of its input and output which determines the next state of the flipflop.

T FF

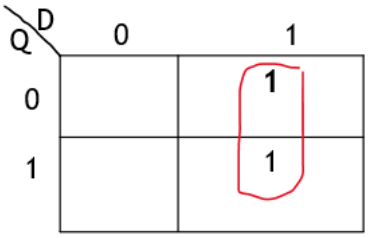
Q	T	Q(t+1)
0	0	0
0	1	1
1	0	1
1	1	0



Q(t+1)=TQ'+QT'

DFF

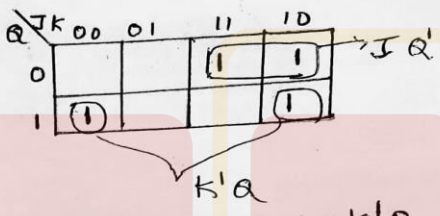
Q	D	Q(t+1)
0	0	0
0	1	1
1	0	0
1	1	1



Q(t+1)=D

JKFF

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



$Q(t+1) = JQ' + K'Q$

Write Verilog code for Flipflops

SR flipflop

```
module sr(clk,s,r,q);
 input clk,s,r;
 output  q;
 reg q;
 always @(posedge clk)
  begin

 case ({s,r})
 2'b00: q <= q; // No change
 2'b01: q <= 1'b0; // reset
 2'b10: q <= 1'b1; // set
 2'b11: q <= 1'bx; // Invalid inputs
 endcase
 end
 endmodule
```

JK Flipflop

```
module jk( input j, input k, input clk, output reg q);

always @ (posedge clk)
case ({j,k})
 2'b00 : q <= q;
 2'b01 : q <= 0;
 2'b10 : q <= 1;
 2'b11 : q <= ~q;
 endcase
endmodule
```

D flipflop

```
module dataff ( clk,d,q);
input clk,d;
output reg q;
always @(posedge clk )
 begin
 if(d ==  0)
      q <=0 ;
 else
     q = 1;
 end
 endmodule
```

T Flipflop

```
module toggleff (clk,t,q);
input clk,t;
output reg q;
always @(posedge clk )
 begin
 if(t ==0)
      q <=q;
 else
     q =~q;
 end
 endmodule
```

----------**********----------

MODULE – 3

Basic Structure of Computers, Instructions&Programs

Topics:

Functional Units
Basic Operational Concepts
Bus Structures
Performance –Processor Clock
Basic Performance Equation
Clock Rate
Performance Measurement.
Memory Location and Addresses
Memory Operations
Instructions and Instruction Sequencing
Addressing Modes

Introduction

- Computer Organization explains the function and design of the various units of digital computers that store and process information.
- It also deals with the input units of the computer which receive information from external sources and the output units which send computed results to external destinations.
- The input, storage, processing, and output operations are governed by a list of instructions that constitute a program.
- It deals about computer hardware and computer architecture.
- Computer hardware consists of electronic circuits, magnetic and optical storage devices, displays, electromechanical devices, and communication facilities.

Computer architecture encompasses the specification of an instruction set and the functional behavior of the hardware units that implement the instructions

Classes of Computers

- Desktop/laptop computers
  - General purpose, variety of software
  - Subject to cost/performance tradeoff
- Workstations
  - More computing power used in engg. applications, graphics etc.
- Enterprise System/ Mainframes
  - Used for business data processing
- Server computers (Low End Range)
  - Network based
  - High capacity, performance, reliability
  - Range from small servers to building sized
- Supercomputer (High End Range)
  - Large scale numerical calculation such as weather forecasting, aircraft design
- Embedded computers
  - Hidden as components of systems
  - Stringent power/performance/cost constraints

Functional Units

- A computer consists of five functionally independent main parts: input, memory, arithmetic and logic, output, and control units, as shown in Figure 1.1.

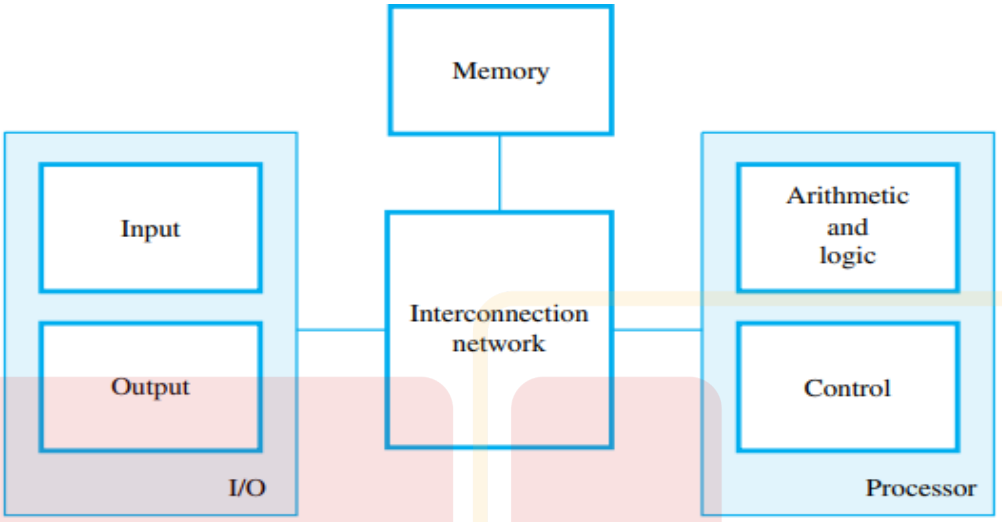


Figure 1.1 Basic functional units of a computer.

- The **input unit** accepts coded information from human operators using devices such as keyboards, or from other computers over digital communication lines.
- The information received is stored in the computer‟s memory, either for later use or to be processed immediately by the **arithmetic and logic unit.**
- ***The processing steps are specified by a program that is also stored in the memory.***
- ***Finally, the results are sent back to the outside world through the output unit.***
- All of these actions are coordinated by the **control unit.**
- An **interconnection network** provides the means for the functional units to exchange information and coordinate their actions.

Input Units

- Computers accept coded information through input units.
- The most common input device is the keyboard.
- Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted to the processor.
- Many other kinds of input devices for human-computer interaction are available,including the touchpad, mouse, joystick, and trackball.
- These are often used as graphic input devices in conjunction with displays.

- Microphones can be used to capture audio input which is then sampled and converted into digital codes for storage and processing.
- Similarly, cameras can be used to capture video input.
- Digital communication facilities, such as the Internet, can also provide input to acomputer from other computers and database servers.

- The function of the memory unit is to **store programs and data.**
- There are two classes of storage, called **primary** and secondary.

Primary Memory :

- Primary memory, also called **main memory**, is a fast memory that operates at electronic speeds.
- Programs must be stored in this memory while they are being executed.
- The memory consists of a large number of semiconductor storage cells, each capable of storing one bit of information. These cells are rarely read or written individually.
- Instead, they are handled in groups of fixed size called **words.**
- The memory is organized so that one word can be stored or retrieved in one basic operation.
- The number of bits in each word is referred to as the word length of the computer, typically **16, 32, or 64 bits.**
  - To provide easy access to any word in the memory, a distinct address is associated with each word location.
  - Addresses are consecutive numbers, starting from 0, that identify successive locations.
  - A particular word is accessed by specifying its address and issuing a control command to the memory that starts the storage or retrieval process.
  - Instructions and data can be written into or read from the memory under the control of the processor.
  - It is essential to be able to access any word location in the memory as quickly as possible.
  - A memory in which any location can be accessed in a short and fixed amount of time after specifying its address is called a random-access memory (RAM). The time required to access one word is called the memory access time. This time is independent of the location of the word being accessed.
  - It typically ranges from a few nanoseconds (ns) to about 100 ns for current RAM units**.**

Cache Memory

- As an adjunct to the main memory, a smaller, faster RAM unit, called a **cache,** is used to hold sections of a program that are currently being executed, along with any associated data.
- The cache is tightly coupled with the processor and is usually contained on the same integrated-circuit chip.
- The purpose of the cache is to facilitate high instruction execution rates.
- At the start of program execution, the cache is empty.
- All program instructions and any required data are stored in the main memory.
- As execution proceeds, instructions are fetched into the processor chip, and a copy of each is placed in the cache.
- When the execution of an instruction requires data located in the main memory, thedata are fetched and copies are also placed in the **cache.**

Secondary Storage

- Primary memory is essential, it tends to be expensive and does not retain information when power is turned off.
- Thus additional, less expensive, permanent secondary storage is used when large amounts of data and many programs have to be stored, particularly for information that is accessed infrequently.
- Access times for secondary storage are longer than for primary memory.
- A wide selection of secondary storage devices is available, including magnetic disks, optical disks (DVD and CD), and flash memory devices.

Arithmetic and Logic Unit

- Most computer operations are executed in the **arithmetic and logic unit (ALU)** of the processor.
- Any arithmetic or logic operation, such as addition, subtraction, multiplication, division, or comparison of numbers, is initiated by bringing the required operandsinto the processor, where the operation is performed by the ALU.
- **For example,** if two numbers located in the memory are to be added, they are brought into the processor, and the addition is carried out by the ALU.
- The sum may then be stored in the memory or retained in the processor for immediate use.
- When operands are brought into the processor, they are stored in high-speed storage elements called **registers.**
- Each register can store one word of data.
- Access times to registers are even shorter than access times to the cache unit on the processor chip.

Output Unit

- The output unit is the counterpart of the input unit.
- Its function is to send processed results to the outside world.
- A familiar example of such a device is a **printer.**
- Most printers employ either photocopying techniques, as in laser printers, or ink jetstreams.
- Such printers may generate output at speeds of 20 or more pages per minute.
- However, printers are mechanical devices, and as such are quite slow compared tothe electronic speed of a processor.
- Some units, such as graphic displays, provide both an output function, showing text
- and graphics, and an input function, through touchscreen capability.

Control Unit

- The memory, arithmetic and logic, and I/O units store and process information and perform input and output operations.
- The operation of these units must be coordinated in some way.
- This is the responsibility of the **control unit.**
- The control unit is effectively the nerve center that sends control signals to other units and senses their states.
- I/O transfers, consisting of input and output operations, are controlled by program instructions that identify the devices involved and the information to be transferred.
- Control circuits are responsible for generating the timing signals that govern the transfers and determine when a given action is to take place.
- Data transfers between the processor and the memory are also managed by the control unit through timing signals.
- A large set of control lines (wires) carries the signals used for timing and synchronization of events in all units.

- The operation of a computer can be summarized as follows:
  - The computer accepts information in the form of programs and data through an input unit and stores it in the memory.
  - Information stored in the memory is fetched under program control into anarithmetic and logic unit, where it is processed
  - Processed information leaves the computer through an output unit
  - All activities in the computer are directed by the control unit

1. BASIC OPERATIONAL CONCEPTS:

The program to be executed is stored in memory. Instructions are accessed from memory to the processor one by one and executed.

STEPS FOR INSTRUCTION EXECUTION

Consider the following instruction

Ex: 1 Add LOCA, R_0

This instruction is in the form of the following instruction format

Opcode Source, Destination

Where Add is the *operation code*, LOCA is the Memory operand and R_0 is Register operand

This instruction adds the contents of memory location LOCA with the contents of Register R_0 and the result is stored in R_0 Register.

The symbolic representation of this instruction is

R_0 [LOCA] + [R_0]

The contents of memory location LOCA and Register R_0 before and after the execution of this instruction is as follows

Before instruction execution	After instruction execution
LOCA = 23H	**LOCA = 23H**
R_0 = 22H	**R_0 = 45H**

The steps for instruction execution are as follows

1. Fetch the instruction from memory into the IR (instruction register in CPU).
2. Decode the instruction 1111000000 10011010
3. Access the Memory Operand
4. Access the Register Operand
5. Perform the operation according to the Operation Code.
6. Store the result into the Destination Memory location or Destination Register.

Ex:2 Add R_1, R_2, R_3

This instruction is in the form of the following instruction format

Opcode, Source-1, Source-2, Destination

Where R1 is Source Operand-1, R2 is the Source Operand-2 and R3 is the Destination. This instruction adds the contents of Register R1 with the contents of R2 and the result is placed in R3 Register.

The symbolic representation of this instruction is

R3 ← [R1] + [R2]

The contents of Registers R1,R2,R3 before and after the execution of this instruction is as follows.

Before instruction execution	After instruction execution
R1 = 24H	**R1 = 24H**
R2 = 34H	**R2 = 34H**
R3 = 38H	**R3 = 58H**

The steps for instruction execution is as follows

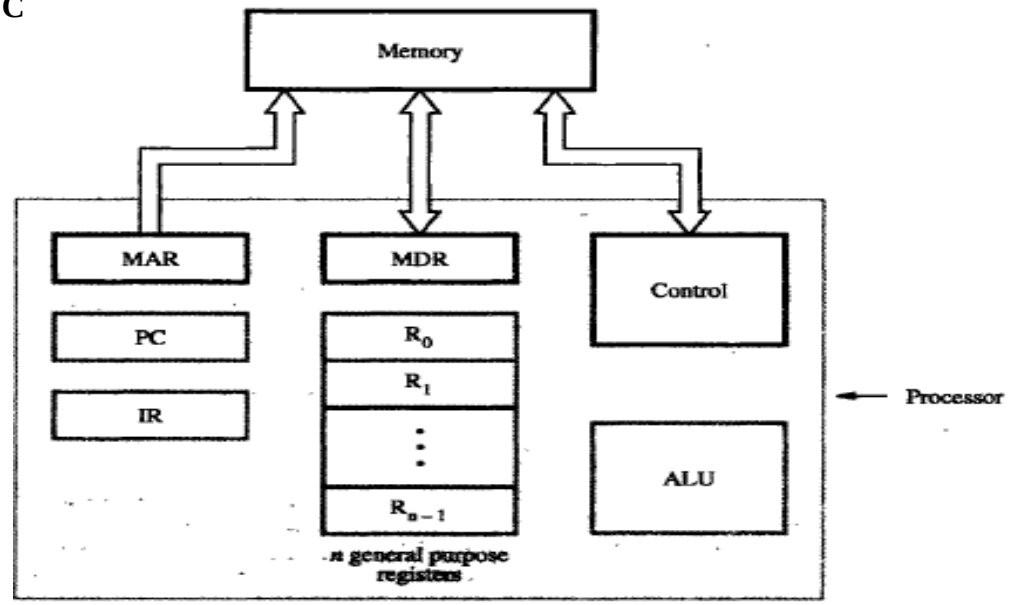
1. Fetch the instruction from memory into the IR.
2. Decode the instruction
3. Access the First Register Operand R1
4. Access the Second Register Operand R2
5. Perform the operation according to the Operation Code.
6. Store the result into the Destination Register R3.

CONNECTION BETWEEN MEMORY AND PROCESSOR

The connection between Memory and Processor is as shown in the figure.

The Processor consists of different types of registers.

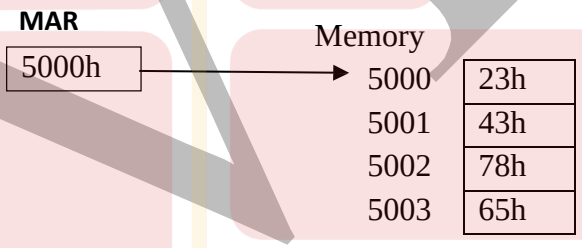
1. MAR (Memory Address Register)
2. MDR (Memory Data Register)
3. Control Unit
4. PC (Program Counter)
5. General Purpose Registers
6. IR (Instruction Register)
7. ALU (Arithmetic and Logic Unit)



The functions of these registers are as follows

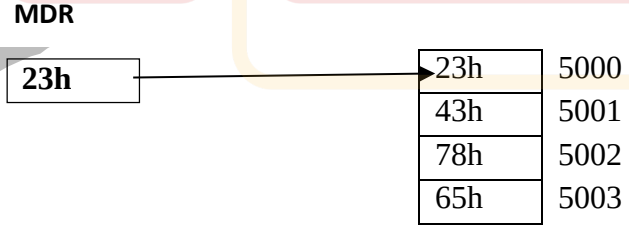
1. MAR

- It establishes communication between Memory and Processor
- It stores the address of the Memory Location as shown in the figure.



2. MDR

- It also establishes communication between Memory and the Processor.
- It stores the **contents** of the memory location (data or operand), written into or read from memory as shown in the figure.



3. CONTROL UNIT

- It controls the data transfer operations between memory and the processor.
- It controls the data transfer operations between I/O and processor.
- It generates control signals for Memory and I/O devices.

4. PC (PROGRAM COUNTER)

- It is a special purpose register used to hold the address of the next instruction to be executed.
- The contents of PC are incremented by 1 or 2 or 4, during the execution of current instruction.
- The contents of PC are incremented by 1 for 8 bit CPU, 2 for 16 bit CPU and for 4 for 32 bit CPU.

4. GENERAL PURPOSE REGISTER / REGISTER ARRAY

The structure of register file is as shown in the figure

R_0
R_1
R_2
.
R_{n-1}

- It consists of set of registers.
- A register is defined as group of flip flops. Each flip flop is designed to store 1 bit of data.
- It is a storage element.
- It is used to store the data temporarily during the execution of the program(eg: result).
- It can be used as a pointer to Memory.
- The Register size depends on the processing speed of the CPU
- EX: Register size = 8 bits for 8 bit CPU

5. IR (INSTRUCTION REGISTER

It holds the instruction to be executed. It notifies the control unit, which generates timing signals that controls various operations in the execution of that instruction.

6. ALU (ARITHMETIC and LOGIC UNIT)

- It performs arithmetic and logical operations on given data.

Steps for reading the instruction.

PC contents are transferred to MAR and read signal is sent to memory by control unit.

The data from memory location is read and sent to MDR.

The content of MDR is moved to IR.

[PC] → MAR $\xrightarrow[\text{CU (read signal)}]{}$ Memory → MDR → IR

2. BUS STRUCTURE

Bus is defined as set of parallel wires used for data communication between different parts of computer. Each wire carries 1 bit of data. There are 3 types of buses, namely

1. Address bus
2. Data bus and
3. Control bus1.

1. Address bus :

- It is unidirectional.
- The processor (CPU) sends the address of an I/O device or Memory device by means of this bus.

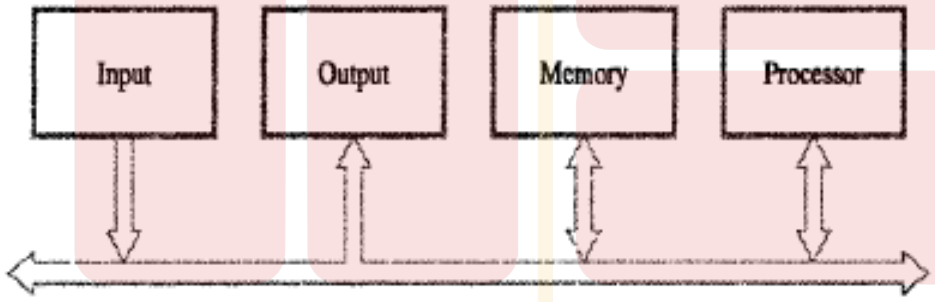
2. Data bus

- It is a bidirectional bus.
- The CPU sends data from Memory to CPU and vice versa as well as from I/O to CPU and vice versa by means of this bus.

3. Control bus:

- This bus carries control signals for Memory and I/O devices. It generates control signals for Memory namely MEMRD and MEMWR and control signals for I/O devices namely IORD and IOWR.

The structure of single bus organization is as shown in the figure.



- The I/O devices, Memory and CPU are connected to this bus is as shown in the figure.
- It establishes communication between two devices, at a time.

Features of Single bus organization are

- Less Expensive
- Flexible to connect I/O devices.
- Poor performance due to single bus.

There is a variation in the devices connected to this bus in terms of speed of operation. Few devices like keyboard, are very slow. Devices like optical disk are faster. Memory and processor are faster, but all these devices uses the same bus. Hence to provide the synchronization between two devices, a buffer register is attached to each device. It holds the data temporarily during the data transfer between two devices.

3. *PERFORMANCE*

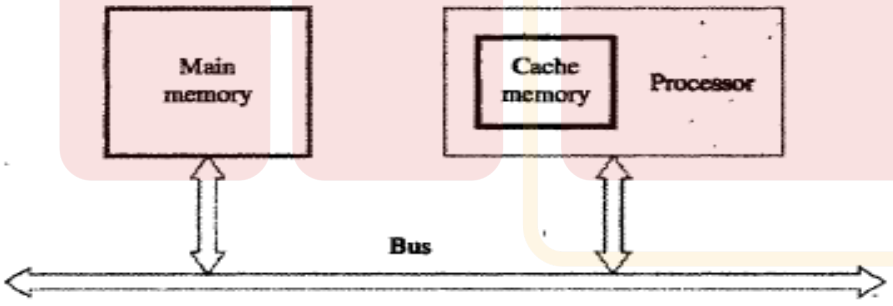
- The performance of a Computer System is based on hardware design of the processor and the instruction set of the processors.
- To obtain high performance of computer system it is necessary to reduce the execution time of the processor.
- Execution time: It is defined as total time required executing one complete program.
- The processing time of a program includes time taken to read inputs, display outputs, system services, execution time etc.
- The performance of the processor is inversely proportional to execution time of the processor.

More performance = Less Execution time.
Less Performance = More Execution time.

The Performance of the Computer System is based on the following factors

1. *Cache Memory*
2. *Processor clock*
3. *Basic Performance Equation*
4. *Instructions*
5. *Compiler*

***CACHE MEMORY*:** It is defined as a *fast access memory* located in between CPU and Memory. It is part of the processor as shown in the fig



The processor needs more time to read the data and instructions from main memory because main memory is away from the processor as shown in the figure. Hence it slowdown the performance of the system.

The processor needs less time to read the data and instructions from Cache Memory because it is part of the processor. Hence it improves the performance of the system.

***PROCESSOR CLOCK*:** The processor circuits are controlled by timing signals called as Clock. It defines constant time intervals and are called as Clock Cycles. To execute one instruction there are 3 basic steps namely

1. Fetch

2. Decode
3. Execute.

The processor uses one clock cycle to perform one operation as shown in the figure

Clock Cycle → T1 T2 T3
Instruction → Fetch Decode Execute

The performance of the processor depends on the length of the clock cycle. To obtain high performance reduce the length of the clock cycle. Let „ P " be the number of clock cycles generated by the Processor and „ R „ be the Clock rate .

The Clock rate is inversely proportional to the number of clock cycles.

i.e R = 1/P.

Cycles/second is measured in Hertz (Hz). Eg: 500MHz, 1.25GHz.

Two ways to increase the clock rate –

- Improve the IC technology by making the logical circuit work faster, so that the time taken for the basic steps reduces.
- Reduce the clock period, P.

BASIC PERFORMANCE EQUATION

Let „T „be *total time* required to execute the program.

Let „N „be the *number of instructions* contained in the program.

Let „S „be the *average number of steps* required to one instruction.

Let „R" be number of clock cycles per second generated by the processor to execute one program.

Processor Execution Time is given by

T = N * S / R

This equation is called as Basic Performance Equation.

For the programmer the value of T is important. To obtain high performance it is necessary to reduce the values of N & S and increase the value of R

Performance of a computer can also be measured by using **benchmark** programs.

SPEC (System Performance Evaluation Corporation) is an non-profitable organization, that measures performance of computer using SPEC rating. The organization publishes the application programs and also time taken to execute these programs in standard systems.

$$SPEC = \frac{Running\ time\ of\ reference\ Computer}{Running\ time\ of\ computer\ under\ test}$$

DIFFERENCES MULTIPROCESSOR AND MULTICOMPUTER

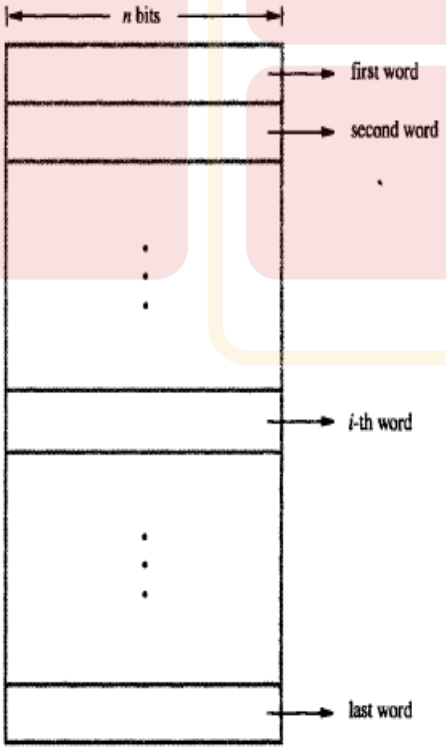
MULTIPROCESSOR	MULTICOMPUTER
1. It is a process of interconnection of two or more processors by means of system bus.	It is a process of interconnection of two or more computers by means of system bus.
2. It uses common memory to hold the data and instructions.	It has its own memory to store data and instructions.
3. Complexity in hardware design.	Not much complexity in hardware design.
4. Difficult to program for multiprocessor system.	Easy to program for multiprocessor system

4. MEMORY LOCATIONS AND ADDRESSES

1. Memory is a storage device. It is used to store character operands, data operands and instructions.
2. It consists of number of semiconductor cells and each cell holds 1 bit of information. A group of 8 bits is called as byte and a group of 16 or 32 or 64 bits is called as word.

World length = 16 for 16 bit CPU and World length = 32 for 32 bit CPU. Word length is defined as number of bits in a word.

- Memory is organized in terms of bytes or words.
- The organization of memory for 32 bit processor is as shown in the fig.



The contents of memory location can be accessed for read and write operation. The memory is accessed either by specifying address of the memory location or by name of the memory location.

- **Address space :** It is defined as number of bytes accessible to CPU and it depends on the number of address lines.

5. BYTE ADDRESSABILITY

Each byte of the memory are addressed, this addressing used in most computers are called byte addressability. Hence Byte Addressability is the process of assignment of address to successive bytes of the memory. The successive bytes have the addresses 1, 2, 3, 4…….2^n-1. The memory is accessed in words.

In a 32 bit machine, each word is 32 bit and the successive addresses are 0,4,8,12,… and so on.

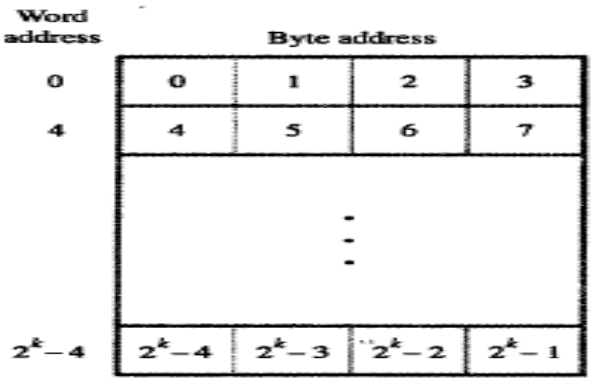
Address	32 – bit word			
0000	0th byte	1st byte	2nd byte	3rd byte
0004	4th byte	5th byte	6th byte	7th byte
0008	8th byte	9th byte	10th byte	11th byte
0012	12th byte	13th byte	14th byte	15th byte
…..	…..	…..	…..	…..
n-3	n-3th byte	n-2th byte	n-1th byte	nth byte

BIG ENDIAN and LITTLE ENDIAN ASSIGNMENT

Two ways in which byte addresses can be assigned in a word.
Or
Two ways in which a word is stored in memory.

1. Big endian
2. Little endian

BIG ENDIAN ASSIGNMENT



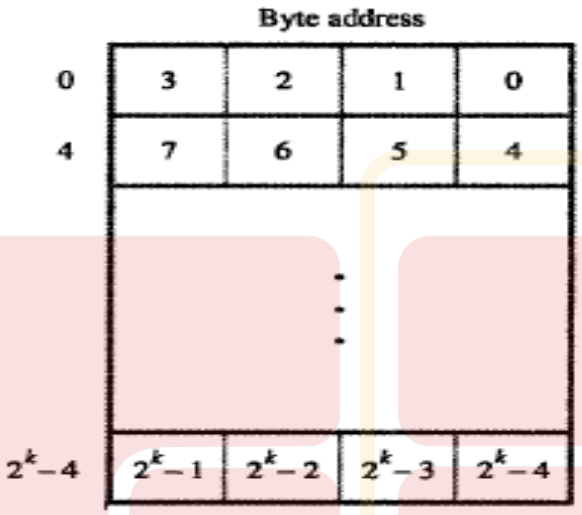
In this technique lower byte of data is assigned to higher address of the memory and higher byte of data is assigned to lower address of the memory.

The structure of memory to represent 32 bit number for big endian assignment is as shown in the above figure.

LITTLE ENDIAN ASSIGNMENT

In this technique lower byte of data is assigned to lower address of the memory and higher byte of data is assigned to higher address of the memory.

The structure of memory to represent 32 bit number for little endian assignment is as shown in the fig.



Eg – store a word "JOHNSENA" in memory starting from word 1000, using Big Endian and Little endian.

Bigendian -

1000	J 1000	O 1001	H 1002	N 1003
1004	S 1004	E 1005	N 1006	A 1007

Little endian -

1000	J 1003	O 1002	H 1001	N 1000
1004	S 1007	E 1006	N 1005	A 1004

WORD ALLIGNMENT

Word size → 16 bit, 32 bit, 64 bit

The structure of memory for 16 bit CPU, 32 bit CPU and 64 bit CPU are as shown in the figures 1,2 and 3 respectively

For 16 bit CPU

5000	34H
5002	65H
5004	86H
5006	93H
5008	45H

For 32 bit CPU

5000	34H
5004	65H
5008	86H
5012	93H
5016	45H

For 64 bit CPU

5000	34H
5008	65H
5016	86H
5024	93H
5032	45H

It is process of assignment of addresses of two successive words and this address is the number of bytes in the word is called as Word alignment.

ACCESSING CHARACTERS AND NUMBERS

The character occupies 1 byte of memory and hence byte address for memory.
The numbers occupies 2 bytes of memory and hence word address for numbers.

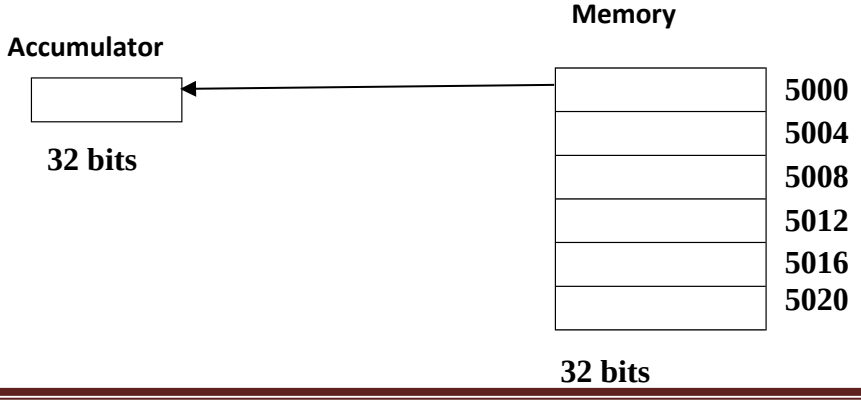
6. MEMORY OPERATION

Both program instructions and operands are in memory. To execute each instruction has to be read from memory and after execution the results must be written to memory. There are two types of memory operations namely 1. Memory read and 2. Memory write

Memory read operation [Load/ Read / Fetch]
Memory write operation [Store/ write]

1. MEMORY READ OPERATION:

- ✓ It is the process of transferring of 1 word of data from memory into Accumulator (GPR).
- ✓ It is also called as Memory fetch operation.
- ✓ The Memory read operation can be implemented by means of LOAD instruction.
- ✓ The LOAD instruction transfers 1 word of data (1 word = 32 bits) from Memory into the Accumulator as shown in the fig.



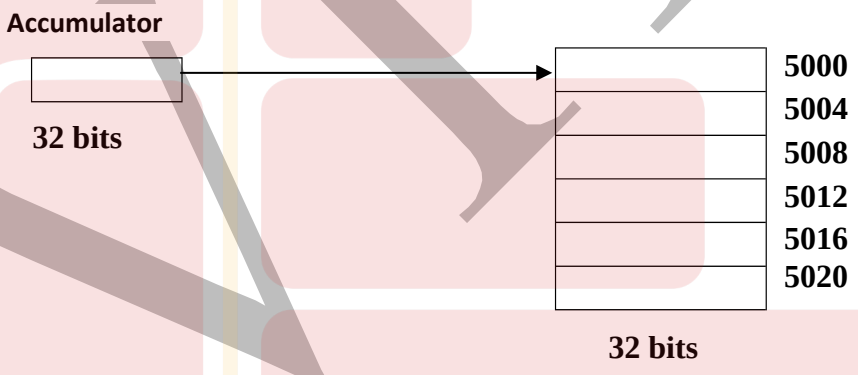
Steps for Memory Read Operation

(1) The processor loads MAR (Memory Address Register) with the address of the memory location.
(2) The Control unit of processor issues memory read control signal to enable the memory component for read operation.
(3) The processor reads the data from memory into the Accumulator by means of bi-directional data bus.

[MAR] → Memory → Accumulator

MEMORY WRITE OPERATION

- It is the process of transferring the 1 word of data from Accumulator into the Memory.
- The Memory write operation can be implemented by means of STORE instruction. The STORE instruction transfers 1 word of data from Accumulator into the Memory location as shown in the fig.



Steps for Memory Write Operation

- The processor loads MAR with the address of the Memory location.
- The Control Unit issues the Memory Write control signal.
- The processor transfers 1 word of data from the Accumulator into the Memory location by means of bi-directional data bus.

7. COMPUTER OPERATIONS (OR) INSTRUCTIONS AND INSTRUCTION EXECUTION

The Computer is designed to perform 4 types of operations, namely

- Data transfer operations
- ALU Operations
- Program sequencing and control.
- I/O Operations.

1. Data Transfer Operations

a) Data transfer between two registers.

Format: Opcode Source1 , Destination

The processor uses MOV instruction to perform data transfer operation between two registers The mathematical representation of this instruction is R1 → R2.

Ex : MOV R_1, R_2 : R1 and R2 are the registers.

Where MOV is the operation code, R1 is the source operand and R2 is the destination operand. This instruction transfers the contents of R1 to R2.

EX: Before the execution of MOV R1,R2, the contents of R1 and R2 are as follows

R1 = 34h and R2 = 65h

After the execution of MOV R1, R2, the contents of R1 and R2 are as follows

R1 = 34H and R2 = 34H

b) Data transfer from memory to register

The processor uses **LOAD** instruction to perform data transfer operation from memory to register. The mathematical representation of this instruction is

[LOCA] → ACC. Where ACC is the Accumulator.

Format : opcode operand

Ex: LOAD LOCA

For this instruction Memory Location is the source and Accumulator is the destination.

c) Data transfer from Accumulator register to memory

The processor uses **STORE** instruction to perform data transfer operation from Accumulator register to memory location. The mathematical representation of this instruction is

[ACC] → LOCA. Where, ACC is the Accumulator.

Format: opcode operand

Ex: STORE LOCA

For this instruction accumulator is the source and memory location is the destination.

2. ALU Operations

The instructions are designed to perform arithmetic operations such as Addition, Subtraction, Multiplication and Division as well as logical operations such as AND, OR and NOT operations.

Ex1: ADD R_0, R_1

The mathematical representation of this instruction is as follows:

$R_1 \leftarrow [R_0] + [R_1]$; Adds the content of R_0 with the content of R_1 and result is placed in R_1.

Ex2: SUB R_0, R_1

The mathematical representation of this instruction is as follows:

$R_1 \leftarrow [R_0] - [R_1]$; Subtracts the content of R_0 from the content of R_1 and result is placed in R_1.

EX3: AND R_0, R_1 ; It Logically multiplies the content of R_0 with the content of R_1 and result is stored in R_1. ($R_1 = R_0$ AND R1)

Ex4: NOT R_0 ; It performs the function of complementation.

3. I/O Operations: The instructions are designed to perform INPUT and OUTPUT operations. The processor uses MOV instruction to perform I/O operations.

The input Device consists of one temporary register called as DATAIN register and output register consists of one temporary register called as DATAOUT register.

a) Input Operation: It is a process of transferring one WORD of data from DATA IN register to processor register.

Ex: MOV DATAIN, R0

The mathematical representation of this instruction is as follows,

$R_0 \leftarrow$ [DATAIN]

b) Output Operation: It is a process of transferring one WORD of data from processor register to DATAOUT register.

Ex: MOV R_0, DATAOUT

The mathematical representation of this instruction is as follows,

[R_0] → DATAOUT

REGISTER TRANSFER NOTATION

- There are 3 locations to store the operands during the execution of the program namely 1. Register 2. Memory location 3. I/O Port. Location is the storage space used to store the data.
- The instructions are designed to transfer data from one location to another location.

Consider the first statement to transfer data from one location to another location

- " Transfer the contents of Memory location whose symbolic name is given by AMOUNT into processor register R_0."
- The mathematical representation of this statement is given by

$R_0 \leftarrow$ [AMOUNT]

Consider the second statement to add data between two registers

- "Add the contents of R_0 with the contents of R_1 and result is stored in R_2"
- The mathematical representation of this statement is given by

$R_2 \leftarrow [R_0] + [R_1]$.

Such a notation is called as "Register Transfer Notation".

It uses two symbols

1. A pair of square brackets [] to indicate the contents of Memory location and
2. ← to indicate the data transfer operation.

ASSEMBLY LANGUAGE NOTATION

Consider the first statement to transfer data from one location to another location

- " Transfer the contents of Memory location whose symbolic name is given by AMOUNT into processor register R_0."
- The assembly language notation of this statement is given by

MOV AMOUNT, R_0

Opcode Source Destination

This instruction transfers 1 word of data from Memory location whose symbolic name is given by AMOUNT into the processor register R_0.

- The mathematical representation of this statement is given by

$R_0 \leftarrow$ [AMOUNT]

Consider the second statement to add data between two registers

- "Add the contents of R_0 with the contents of R_1 and result is stored in R_2"
- The assembly language notation of this statement is given by

ADD	R_0 ,	R_1,	R_2
Opcode	source1,	Source2,	Destination

This instruction adds the contents of R_0 with the contents of R_1 and result is stored in R_2.

- The mathematical representation of this statement is given by

  $R_2 \leftarrow [R_0] + [R_1]$.

Such a notations are called as "Assembly Language Notations"

BASIC INSTRUCTION TYPES

There are 3 types basic instructions namely

1. Three address instruction format
2. Two address instruction format
3. One address instruction format

Consider the arithmetic expression Z = A + B, Where A,B,Z are the Memory locations.

Steps for evaluation

1. Access the first memory operand whose symbolic name is given by A.
2. Access the second memory operand whose symbolic name is given by B.
3. Perform the addition operation between two memory operands.
4. Store the result into the 3rd memory location Z.
5. The mathematical representation is $Z \leftarrow [A] + [B]$.

a) Three address instruction format : Its format is as follows

opcode	**Source-1**	**Source-2**	**destination**

Destination ← [source-1] + [source-2]

Ex: ADD A, B, Z

Z ← [A] + [B]

a) Two address instruction format : Its format is as follows

opcode	**Source**	**Source/destination**

Destination ← [source] + [destination]

The sequence of two address m/c instructions to evaluate the arithmetic expression Z ← A + B are as follows

```
MOV  A,  R0
MOV  B,  R1
ADD  R0, R1
MOV  R1, Z
```

b) One address instruction format : Its format is as follows

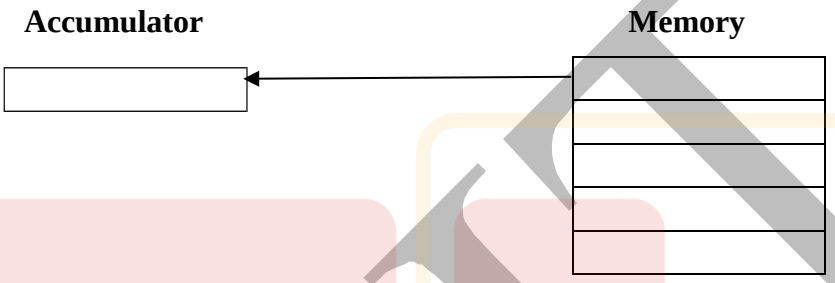
opcode	operand

Ex1: LOAD B

This instruction copies the contents of memory location whose symbolic name is given by „B‟ into the Accumulator as shown in the figure.

The mathematical representation of this instruction is as follows

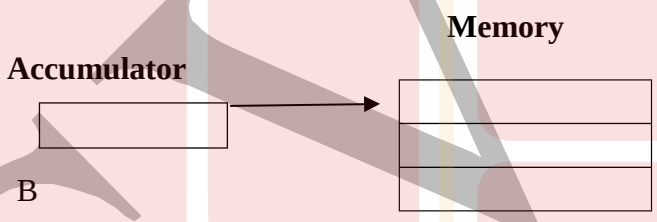
ACC ← [B]



Ex2: STORE B

This instruction copies the contents of Accumulator into memory location whose symbolic name is given by „B‟ as shown in the figure. The mathematical representation is as follows

B ← [ACC].



Ex3: ADD B

- This instruction adds the contents of Accumulator with the contents of Memory location „B‟ and result is stored in Accumulator.
- The mathematical representation of this instruction is as follows

ACC ←[ACC]+ [B]

STRIGHT LINE SEQUENCING AND INSTRUCTION EXECUTION

Consider the arithmetic expression

C = A+B , Where A,B,C are the memory operands.

The mathematical representation of this instruction is

C = [A] + [B].

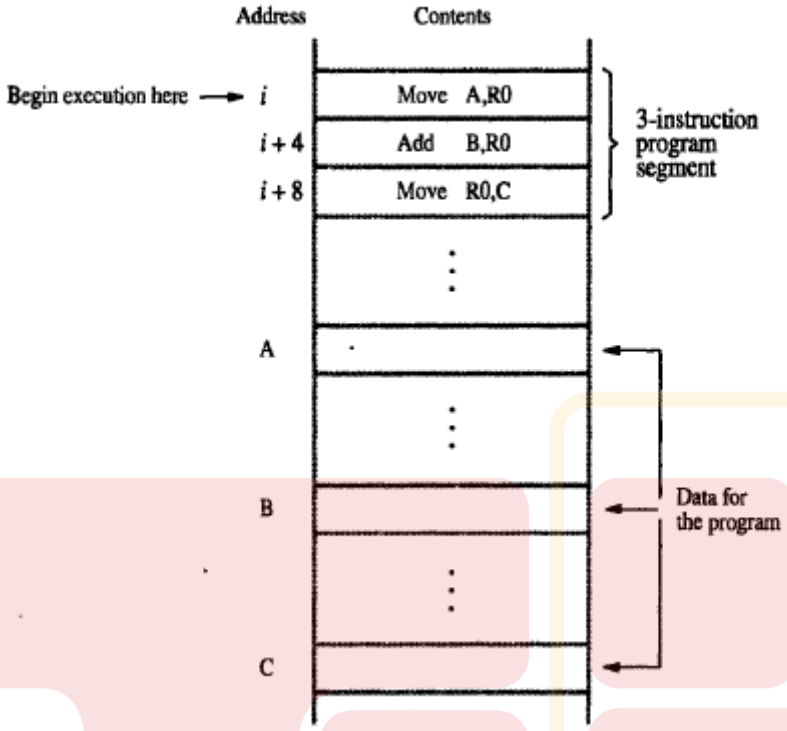
The sequence of instructions using two address instruction format are as follows

MOV A, R_0
ADD B, R_0
MOV R_0, C

Such a program is called as 3 instruction program.

NOTE: The size of each instruction is 32 bits.

- The 3 instruction program is stored in the successive memory locations of the processor is as shown in the fig.



- The system bus consists of uni-directional address bus,bi-directional data bus and control bus

"It is the process of accessing the 1st instruction from memory whose address is stored in program counter into Instruction Register (IR) by means of bi-directional data bus and at the same time after instruction access the contents of PC are incremented by 4 in order to access the next instruction. Such a process is called as "Straight Line Sequencing".

INSTRUCTION EXECUTION

There are 4 steps for instruction execution

1 Fetch the instruction from memory into the Instruction Register (IR) whose address is stored in PC.

IR ← [[PC]]

2 Decode the instruction.

3 Perform the operation according to the opcode of an instruction

4 Load the result into the destination.

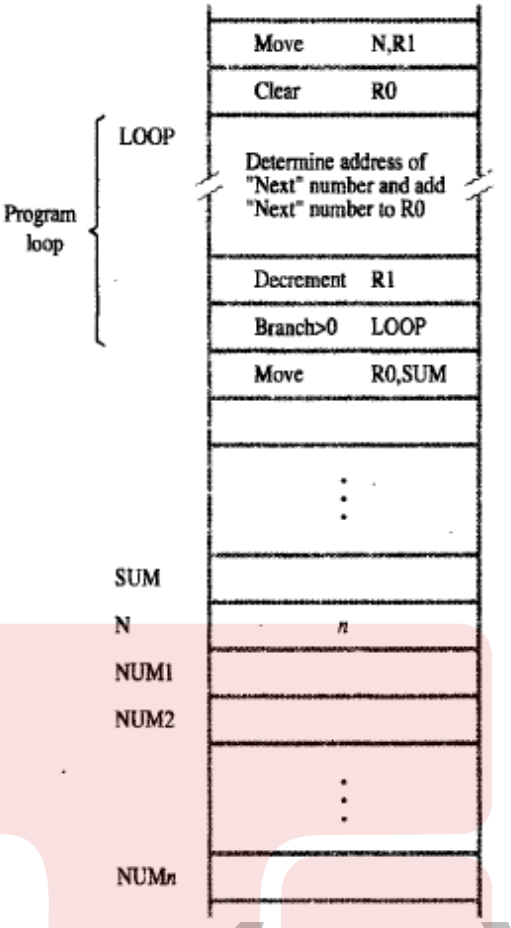
5 During this process, Increment the contents of PC to point to next instruction (In 32 bit machine increment by 4 address)

PC ← [PC] + 4.

6 The next instruction is fetched, from the address pointed by PC.

BRANCHING

Suppose a list of „N" numbers have to be added. Instead of adding one after the other, the add statement can be put in a loop. The loop is a straight-line of instructions executed as many times as needed.



The „N" value is copied to R1 and R1 is decremented by 1 each time in loop. In the loop find the value of next elemet and add it with Ro.

In conditional branch instruction, the loop continues by coming out of sequence only if the condition is true. Here the PC value is set to „LLOP" if the condition is true.

Branch > 0 LOOP // if >0 go to LOOP

The PC value is set to LOOP, if the previous statement value is >0 ie. after decrementing R1 value is greater than 0.

If R1 value is not greater than 0, the PC value is incremented in a mormal sequential way and the next instruction is executed.

CONDITION CODE BITS

- The processor consists of series of flip-flops to store the status information after ALU operation.
- It keeps track of the results of various operations, for subsequent usage.
- The series of flip-flip-flops used to store the status and control information of the processor is called as "Condition Code Register". It defines 4 flags. The format of condition code register is as follows.

C	V	Z	N

1 N (NEGATIVE) Flag:
It is designed to differentiate between positive and negative result.
It is set 1 if the result is negative, and set to 0 if result is positive.

2 Z (ZERO) Flag:
It is set to 1 when the result of an ALU operation is found to zero, otherwise it is cleared.

3 V (OVER FLOW) Flag:
In case of 2^s Complement number system n-bit number is capable of representing a range of numbers and is given by -2^{n-1} to $+2^{n-1.}$. The Over-Flow flag is set to 1 if the result is found to be out of this range.

4 C (CARRY) Flag :
This flag is set to 1 if there is a carry from addition or borrow from subtraction, otherwise it is cleared.

8. Addressing Modes

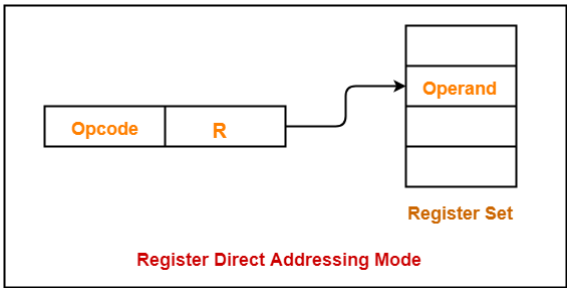
The various formats of representing operand in an instruction or location of an operand is called as "Addressing Mode". The different types of Addressing Modes are

a) Register Addressing
b) Direct Addressing
c) Immediate Addressing
d) Indirect Addressing
e) Index Addressing
f) Relative Addressing
g) Auto Increment Addressing
h) Auto Decrement Addressing

a. REGISTER ADDRESSING:

In this mode operands are stored in the registers of CPU. The name of the register is directly specified in the instruction.

Ex: MOVE R_1,R_2 Where R1 and R2 are the Source and Destination registers respectively. This instruction transfers 32 bits of data from R1 register into R2 register. This instruction does not refer memory for operands. The operands are directly available in the registers.

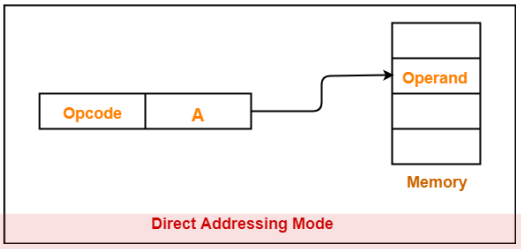


b. DIRECT ADDRESSING

It is also called as Absolute Addressing Mode. In this addressing mode operands are stored in the memory locations. The name of the memory location is directly specified in the instruction.

Ex: MOVE LOCA, R_1 : Where LOCA is the memory location and R1 is the Register.

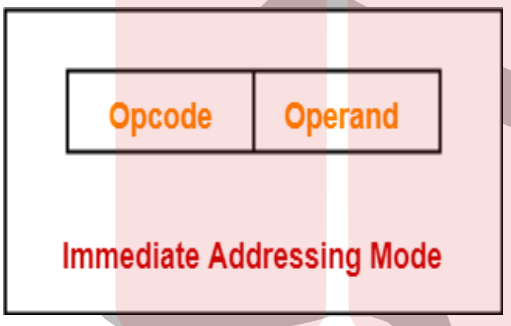
This instruction transfers 32 bits of data from memory location X into the General Purpose Register R1.



c. IMMEDIATE ADDRESSING

In this Addressing Mode operands are directly specified in the instruction. The source field is used to represent the operands. The operands are represented by # (hash) sign.

Ex: MOVE #23, R0



d. INDIRECT ADDRESSING

In this Addressing Mode effective address of an operand is stored in the memory location or General Purpose Register.

The memory locations or GPRs are used as the memory pointers.

Memory pointer: It stores the address of the memory location.

There are two types Indirect Addressing

i) Indirect through GPRs
ii) Indirect through memory location

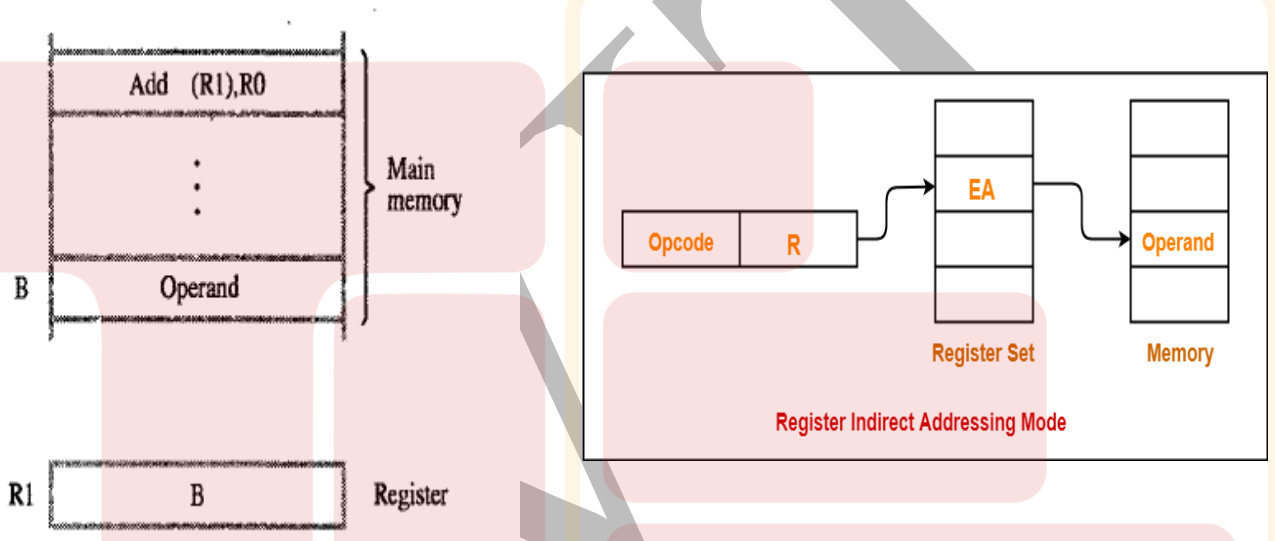
i) Indirect Addressing Mode through GPRs

In this Addressing Mode the effective address of an operand is stored in the one of the General Purpose Register of the CPU.

Ex: ADD (R_1), R_0 ; Where R_1 and R_0 are GPRs

This instruction adds the data from the memory location whose address is stored in R_1 with the contents of R_0 Register and the result is stored in R_0 register as shown in the fig.

The diagrammatic representation of this addressing mode is as shown in the fig.



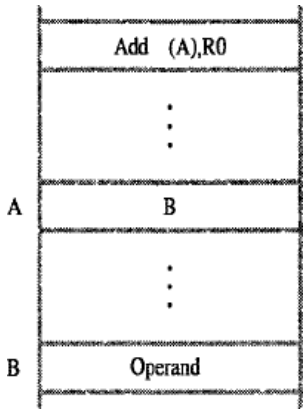
ii) Indirect Addressing Mode through Memory Location.

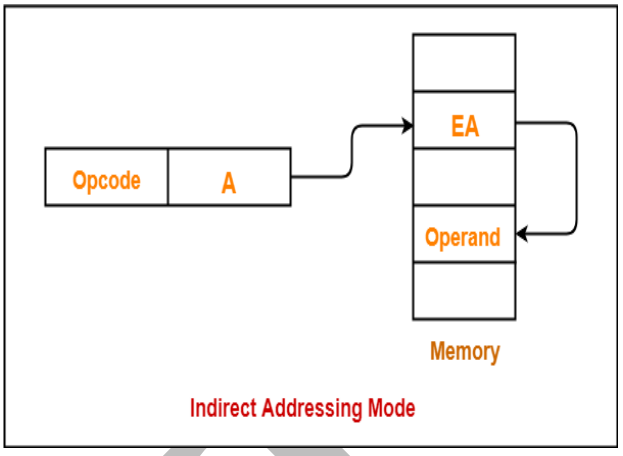
In this Addressing Mode, effective address of an operand is stored in the memory location.

Ex: ADD (X), R_0

This instruction adds the data from the memory location whose address is stored in „X" memory location with the contents of R_0 and result is stored in R_0 register.

The diagrammatic representation of this addressing mode is as shown in the fig.





e. INDEX ADDRESSING MODE

In this addressing mode, the effective address of an operand is computed by adding constant value with the contents of Index Register and any one of the General Purpose Register namely R_0 to R_{n-1} can be used as the Index Register. The constant value is directly specified in the instruction.

The symbolic representations of this mode are as follows

1. X (R_i) where X is the Constant value and R_j is the GPR.

It can be represented as

EA of an operand = X + (R_i)

2. (R_i , R_J) Where R_i and R_j are the General Purpose Registers used to store addresses of an operand and constant value respectively. It can be represented as

The EA of an operand is given by

EA = (R_i) + (R_j)

3. X (R_i , R_j) Where X is the constant value and R_I and R_J are the General Purpose Registers used to store the addresses of the operands.It can be represented as

The EA of an operand is given by

EA = (R_i) + (R_j) + X

There are two types of Index Addressing Modes

i) Offset is given as constant.

ii) Offset is in Index Register.

Note : Offset : It is the difference between the starting effective address of the memory location and the effective address of the operand fetched from memory.

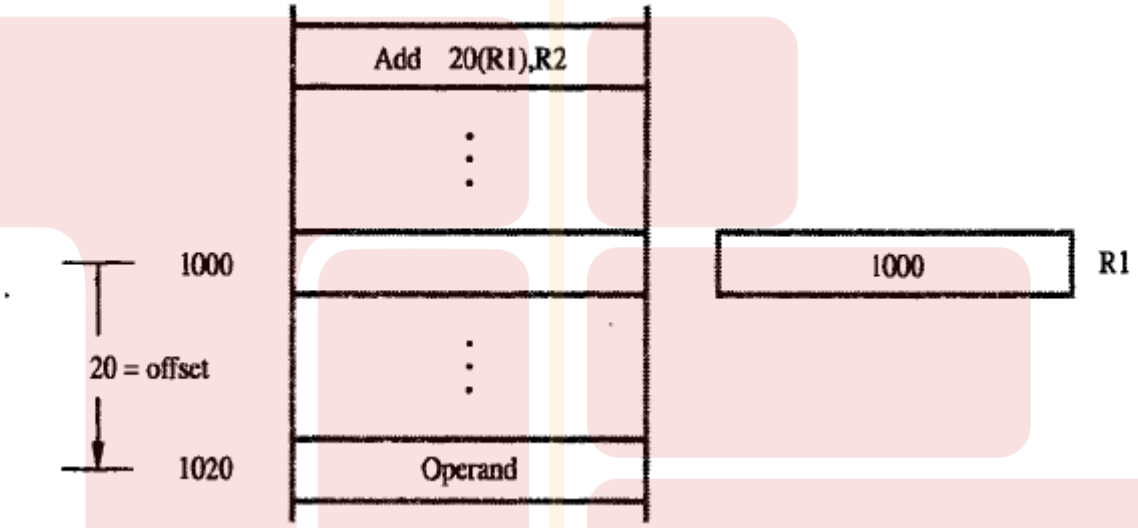
i) Offset is given as constant

Ex: ADD $20(R_1)$, R_2

The EA of an operand is given by

$EA = 20 + [R_1]$

This instruction adds the contents of memory location whose EA is the sum of contents of R_1 with 20 and with the contents of R_2 and result is placed in R_2 register. The diagrammatic representation of this mode is as shown in the fig.



ii) Offset is in Index Register

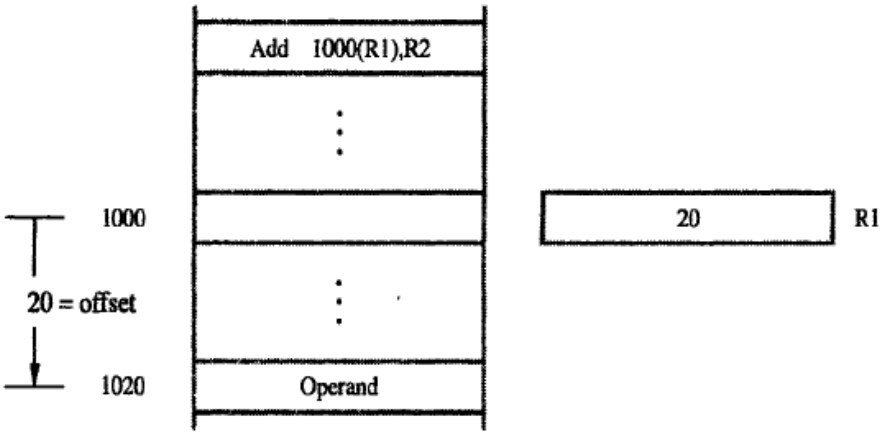
Ex: ADD $1000(R_1)$, R_2 R_1 holds the offset address of an operand.

The EA of an operand is given by

$EA = 1000 + [R_1]$

This instruction adds the data from the memory location whose address is given by [1000 + [R1] with the contents of R_2 and result is placed in R_2 register.

The diagrammatic representation of this mode is as shown in the fig.



f. RELATIVE ADDRESSING MODE:

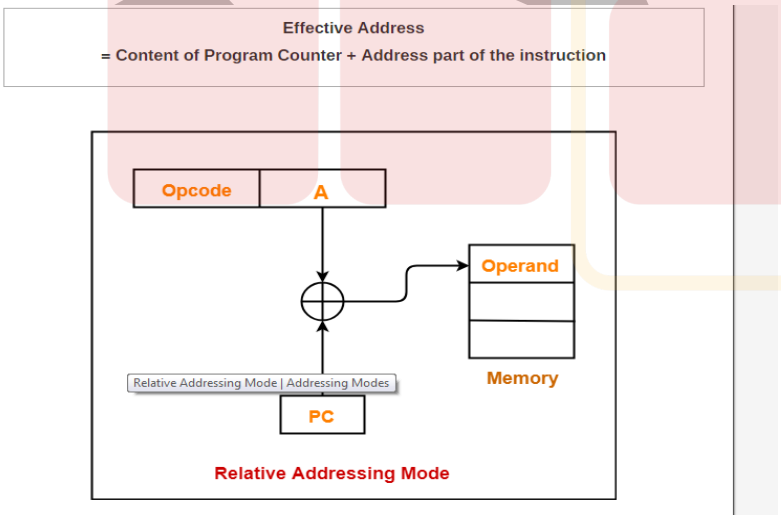
In this Addressing Mode EA of an operand is computed by the Index Addressing Mode. This Addressing Mode uses PC (Program Counter)to store the EA of the next instruction instead of GPR.

The symbolic representation of this mode is X (PC).Where X is the offset value and PC is the Program Counter to store the address of the next instruction to be executed.

It can be represented as

EA of an operand = X + (PC).

This Addressing Mode is useful to calculate the EA of the target memory location.



g. AUTO INCREMENT ADDRESSING MODE

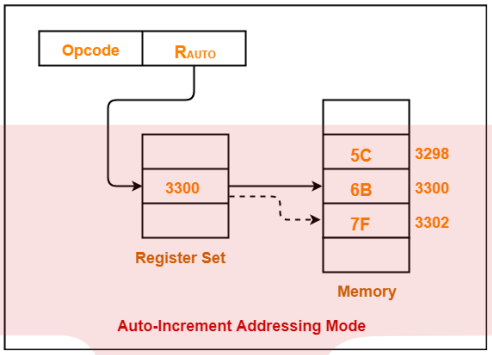
In this Addressing Mode , EA of an operand is stored in the one of the GPR^s of the CPU. This Addressing Mode increment the contents of memory register by 4 memory locations after operand access.

The symbolic representation is

$(R_I)+$ Where R_i is the one of the GPR.

Ex: MOVE $(R_1)+$, R_2

This instruction transfer"s data from the memory location whose address is stored in R_1 into R_3 register and then it increments the contents of R_1 by 4 memory locations.



h. AUTO DECREMENT ADDRESSING MODE

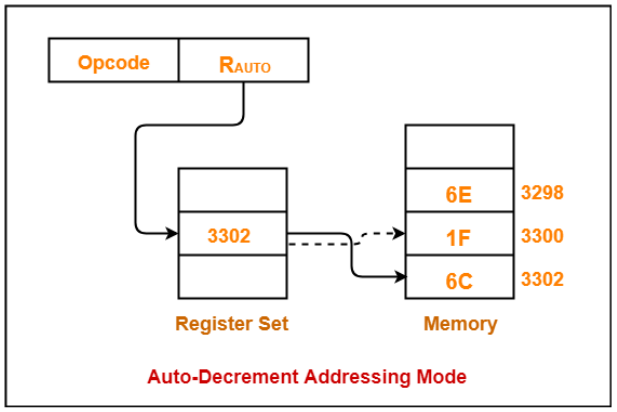
In this Addressing Mode , EA of an operand is stored in the one of the GPR^s of the CPU. This Addressing Mode decrements the contents of memory register by 4 memory locations and then transfers the data to destination.

The symbolic representation is

$-(R_I)$ Where R_i is the one of the GPR.

Ex: MOVE - (R_1) , R_2

This instruction first decrements the contents of R_1 by 4 memory locations and then transfer"s data of that location to destination register.



MODULE 4
INPUT/OUTPUT ORGANIZATION

There are a number of input/output (**I/O**) devices, that can be connected to a computer. The input maybe from a keyboard, a sensor, switch, mouse etc. Similarly output may be a speaker, monitor, printer, a digital display etc.

These variety of I/O devices exchange information in varied format, having different word length, transfer speed is different, but are connected to the same system and exchange information with the same computer. Computer must be capable of handling these wide variety of devices.

ACCESSING I/O-DEVICES

A **single bus-structure** can be used for connecting I/O-devices to a computer. The simple arrangement of connecting set of I/O devices to memory and processor by means of system bus is as shown in the figure. Such an arrangement is called as Single Bus Organization.

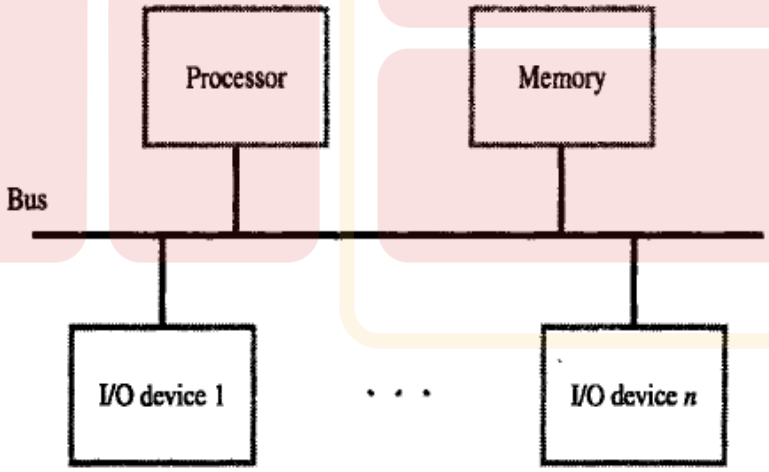


Fig: A Single Bus structure

- The single bus organization consists of
    - Memory
    - Processor
    - System bus
    - I/O device
- The **system bus** consists of **3 types of buses:**
    - Address bus (Unidirectional)
    - Data bus (Bidirectional)
    - Control bus (Bidirectional)
- The system bus enables all the devices connected to it to involve in the data transfer operation.
- The system bus establishes data communication between I/O device and processor.
- Each I/O device is assigned a unique set of address.
- When processor places an address on address-lines, the intended-device responds to the command.
- The processor requests either a read or write-operation.
- The requested data are transferred over the data-lines

Steps for input operation:

- The address bus of system bus holds the address of the input device.
- The control unit of CPU generates $\overline{\text{IORD}}$ Control signal.
- When this control signal is activated the processor reads the data from the input device (DATAIN) into the CPU register.

Steps for output operation:

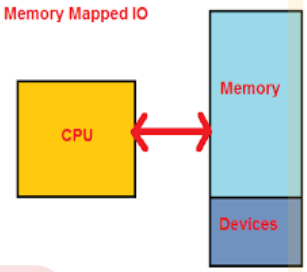
- The address bus of system bus holds the address of the output device.
- The control unit of CPU generates $\overline{\text{IOWR}}$ control signal.
- When this control signal is enabled CPU transfers the data from processor register to output device (DATAOUT)

There are 2 schemes available to connect I/O devices to CPU

1. **Memory mapped I/O:**

- In this technique, both memory and I/O devices use **the common bus** to transfer the data to CPU .
- same address space is used for both memory and I/O interface. They have only one set of read and write signals.
- All memory related instructions are used for data transfer between I/O and processor.
- In case of memory mapped I/O input operation can be implemented as, MOVE DATAIN , R0

Source destination



This instruction sends the contents of location DATAIN to register R0.

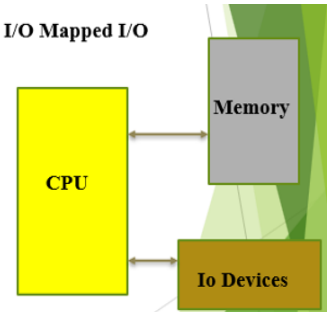
- Similarly output can be implemented as, MOVE R DATAOUT

Source Destination

- The data is written from R0 to DATAOUT location (address of output buffer.

2) I/O Mapped I/O:

- In this technique, a separate bus is used for I/O devices and memory to transfer the data to CPU. Address space for memory and I/O devices are different.
- Hence two sets of instruction are used for data transfer.
- One set for memory operations and another set for I/O operations. Whole address space is available for the program.
- Eg – IN AL, DX



Memory Mapped I/O	I/O Mapped I/O
Memory & I/O share the entire address range of processor	Processor provides separate address range for memory & I/O
Processor provides more address lines for accessing memory	Less address lines for accessing I/O
More Decoding is required	Less decoding is required
Memory control signals used to control Read & Write I/O operations	I/O control signals are used to control Read & Write I/O operations

I/O INTERFACE

The hardware arrangement of connecting i/p device to the system bus is as shown in the fig.

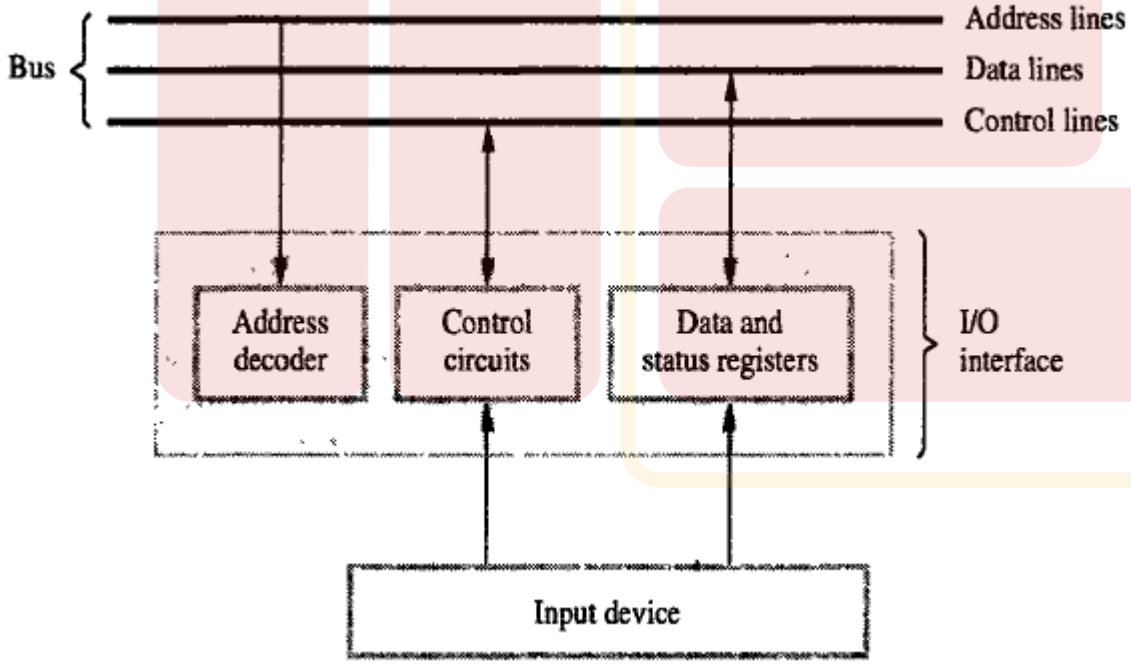


Fig: I/O interface for an input device

This hardware arrangement is called as I/O interface. The I/O interface consists of 3 functional devicesnamely:

1) **Address Decoder:**

- Its function is to decode the address, in-order to recognize the input device whose address isavailable on the unidirectional address bus.
- The recognition of input device is done first, and then the control and data registers becomes active.
- The unidirectional address bus of system bus is connected to input of the address decoder asshown in figure

2) **Control Circuit:**

- The control bus of system bus is connected to control circuit as shown in the fig.
- The processor sends commands to the I/O system through the control bus.
- It **controls the read write operations** with respect to **I/O device**.

3) **Status & Data register:**

- It specifies type of operation (either read or write operation) to be performed on I/O device. Itspecifies the position of operation.

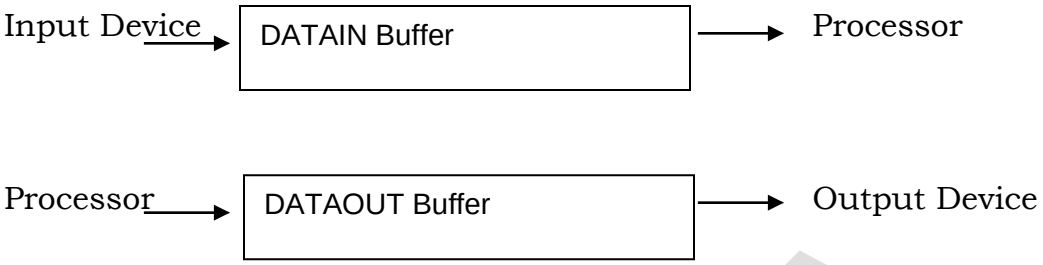
4) **Data Register:**

- The data bus carries the data from the I/O devices to or from the processor. The data bus isconnected to the data/ status register.
- The data register stores the data, read from input device or the data, to be written into outputdevice. There are 2 types:

DATAIN - Input-buffer associated with keyboard.

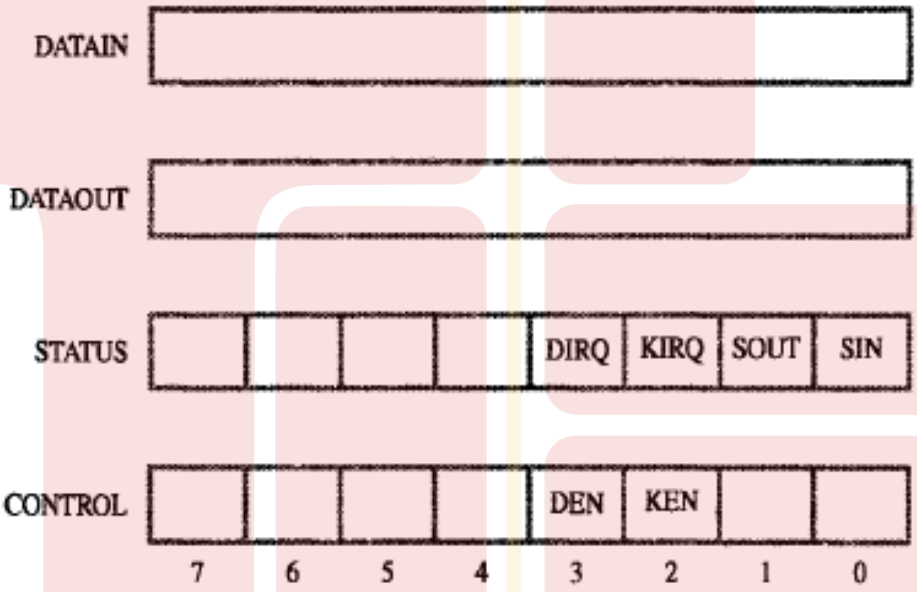
DATAOUT -Output data buffer of a display/printer.

Data buffering is an essential task of an I/O interface. Data transfer rates of processor and memory are high, when compared with the I/O devices, hence the data are buffered at the I/O interface circuit and then forwarded to output device, or forwarded to processor in case of input devices.



Input & Output registers –

Various registers in keyboard and display devices -



DATAIN register: is a part of input device. It is used to store the ASCII characters read from keyboard.

DATAOUT register: is a part of output device. It is used to store the ASCII characters to bedisplayed on the output device.

STATUS register stores the status of working of I/O devices –

- **SIN flag – This flag is set to 1**, when **DATAIN buffer contains the data** from keyboard. **The flag is set to 0**, after the data is passed from DATAIN buffer to the processor.
- **SOUT flag – This flag is set to 1**, when **DATAOUT buffer is empty** and the data can be added to it by processor. The flag is set to 0, when DATAOUT buffer has the data to be displayed.
- **KIRQ (Keyboard Interrupt Request)** – By setting this flag to 1, keyboard requests the processor to obtain its service and an interrupt is sent to the processor. It is used along with the SIN flag.

- **DIRQ(Display Interrupt Request)** – The output device request the processor to obtain its service for output operation, by activating this flag to 1.

Control registers

KEN (keyboard Enable) – Enables the keyboard for input operations.

DEN (Display Enable) – Enables the output device for input operations.

Program Controlled I/O

- In this technique **CPU is responsible for executing data from the memory for output and storing data in memory for executing of Programmed I/O**
- **Drawback** of the Programmed I/O: was that the **CPU has to monitor the units all the times when the program is executing.** Thus, the CPU stays in a program loop until the I/O unit indicates that it is ready for data transfer.
- This is a time-consuming process and the CPU time is wasted a lot in keeping an eye to the executing of program.

- It is the process of controlling the input and output operations by executing 2 sets of instruction,one set for input operation and the next set for output operation.

- The program checks the status of I/O register and reads or displays data. Here the I/O operationis controlled by program.

```
WAITK     TestBit #0, STATUS    (Checks SIN
          flag)Branch = 0 WAITK
          Move DATAIN, R0       (Read character)[
          *Code to read a character from DATAIN to R0]
```

This code checks the SIN flag, and if it is set to 0 (ie. If no character in DATAIN Buffer), then move back to WAITK label. This loop continues until SIN flag is set to 1. When SIN is 1, data ismoved from DATAIN to R0 register. Thus the program, continuously checks for input operation.

Similarly code for Output operation,

```
WAITD     TestBit #0, STATUS    (Checks SOUT flag)
          Branch = 0 WAITD
          Move R0, DATAOUT      (Send character for
                                display)
```

The code checks the SOUT flag, and if it is set to 1 (ie. If no character in DATAOUT Buffer), then move back to WAITK label. This loop continues until SOUT flag is set to 0. When SOUT is 0, data is moved from R0 register to DATAOUT (ie. Sent by processor).

Module-2

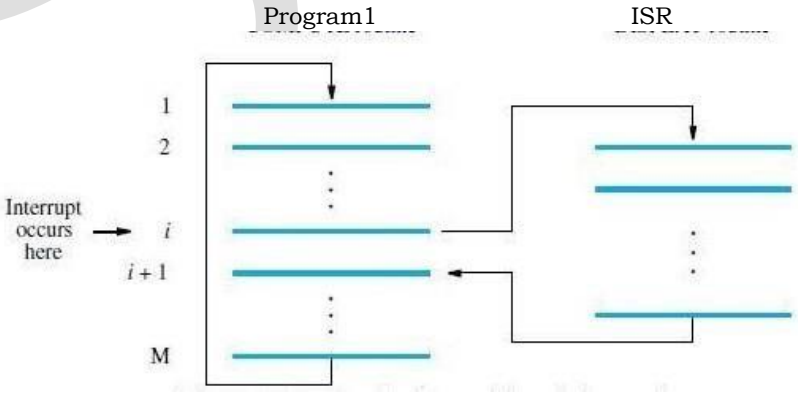
Illustrate a program that reads one line from the keyboard, stores it in memory buffer, and echoes it back to the display in an I/O interfaces. (10 Marks)

```
        Move        #LINE,R0          Initialize memory pointer.
WAITK   TestBit     #0,STATUS         Test SIN.
        Branch=0    WAITK             Wait for character to be entered.
        Move        DATAIN,R1         Read character.
WAITD   TestBit     #1,STATUS         Test SOUT.
        Branch=0    WAITD             Wait for display to become ready.
        Move        R1,DATAOUT        Send character to display.
        Move        R1,(R0)+          Store charater and advance pointer.
        Compare     #$0D,R1           Check if Carriage Return.
        Branch≠0    WAITK             If not, get another character.
        Move        #$0A,DATAOUT      Otherwise, send Line Feed.
        Call        PROCESS           Call a subroutine to process the
                                        the input line.
```

Figure 4.4 A program that reads one line from the keyboard, stores it in memory buffer, and echoes it back to the display.

Interrupt

- It is an event which suspends the execution of one program and begins the execution of another program.
- In program controlled I/O, a program should continuously check whether the I/O device is free. By this continuous checking the processor execution time is wasted. It can be avoided by I/O device **sending an 'interrupt' to the processor, when I/O device is free**.
- The interrupt invokes a subroutine called **Interrupt Service Routine (ISR)**, which resolves the cause of interrupt.
- The occurrence of interrupt causes the processor to transfer the execution control from user program to ISR.



The following steps takes place when the interrupt related instruction is executed:

- It suspends the execution of current instruction i.
- Transfer the execution control to sub program from main program.
- Increments the content of PC by 4 memory location.
- It decrements SP by 4 memory locations.
- Pushes the contents of PC into the stack segment memory whose address is stored in SP.
- It loads PC with the address of the first instruction of the sub program.

The following steps takes place when '*return*' instruction is executed in ISR -

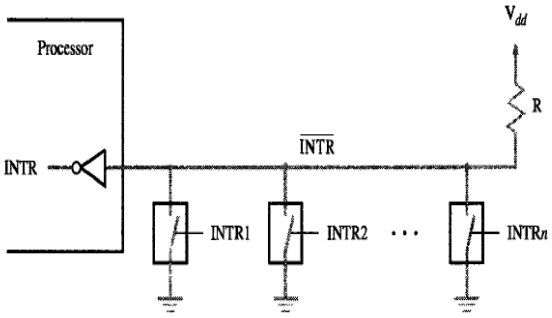
- It transfers the execution control from ISR to user program.
- It retrieves the content of stack memory location whose address is stored in SP into the PC.
- After retrieving the return address from stack memory location into the PC it increments the Content of SP by 4 memory location.

Interrupt Latency / interrupt response time is the delay between the time taken for receiving an interrupt request and start of the execution of the ISR.

Generally, the long interrupt latency is unacceptable.

INTERRUPT HARDWARE

- The external device (I/O device) sends interrupt request to the processor by activating a bus line and called as interrupt request line.
- All I/O device uses the same single interrupt-request line.
- One end of this interrupt request line is connected to input power supply by means of a register.
- The another end of interrupt request line is connected to INTR (Interrupt request) signal of processor as shown in the fig.



- The I/O device is connected to interrupt request line by means of switch, which is grounded as shown in the fig.
- When all the switches are open the voltage drop on interrupt request line is equal to the V_{DD} and INTR value at process is 0.
- This state is called as in-active state of the interrupt request line.
- The I/O device interrupts the processor by closing its switch.

- When switch is closed the voltage drop on the interrupt request line is found to be zero, as the switch is grounded, hence INTR=0 and $\overline{INTR}$=1.
- The signal on the interrupt request line is logical OR of requests from the several I/O devices. Therefore, $\overline{INTR}$ = $\overline{INTR1}$ + $\overline{INTR2}$ + + $\overline{INTRn}$

ENABLING AND DISABLING THE INTERRUPTS

The arrival of interrupt request from external devices or from within a process, causes the suspension of on-going execution and start the execution of another program.

- Interrupt arrives at any time and it alters the sequence of execution. Hence the interrupt to be executed must be selected carefully.
- All computers can enable and disable interruptions as desired.
- When an interrupt is under execution, other interrupts should not be invoked. This is performed in a system in different ways.
- 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.
  2) Processor should automatically disable interrupts before starting the execution of the ISR.
  3) Processor has a special INTR line for which the interrupt-handling circuit. Interrupt-circuit responds only to leading edge of signal. Such line is called 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. And 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.

HANDLING MULTIPLE DEVICES

While handling multiple devices, the issues concerned are:

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

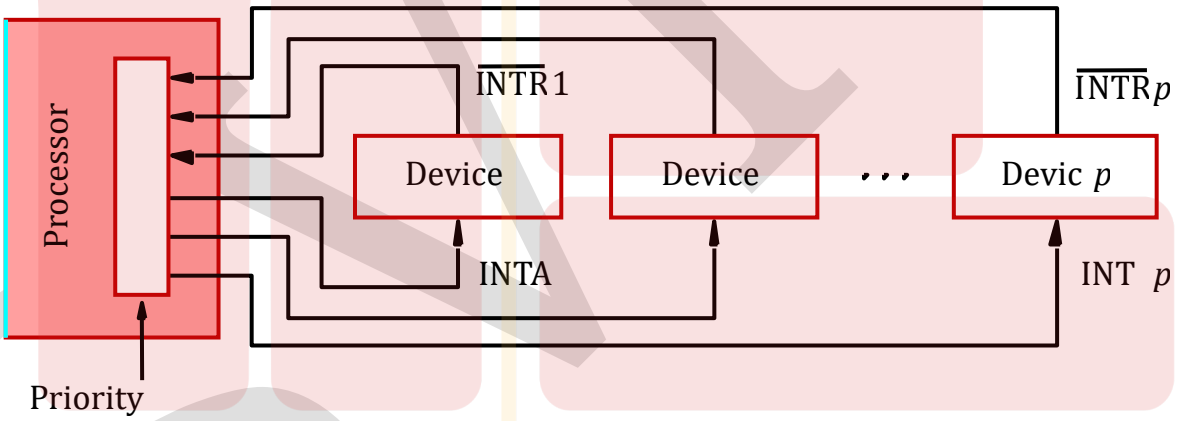
VECTORED INTERRUPT

- 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 staring address to ISR.

- The staring 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

- A multiple-priority scheme is implemented by using separate INTR & INTA lines for each device
- Each INTR line is assigned a different priority-level as shown in Figure.



- Each device has a separate interrupt-request and interrupt-acknowledge line.
- Each interrupt-request line is assigned a different priority level.
- Interrupt requests received over these lines are sent to a priority arbitration circuit in the processor.
- If the interrupt request has a higher priority level than the priority of the processor, then the request is accepted.
- 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
  → accidently 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**.

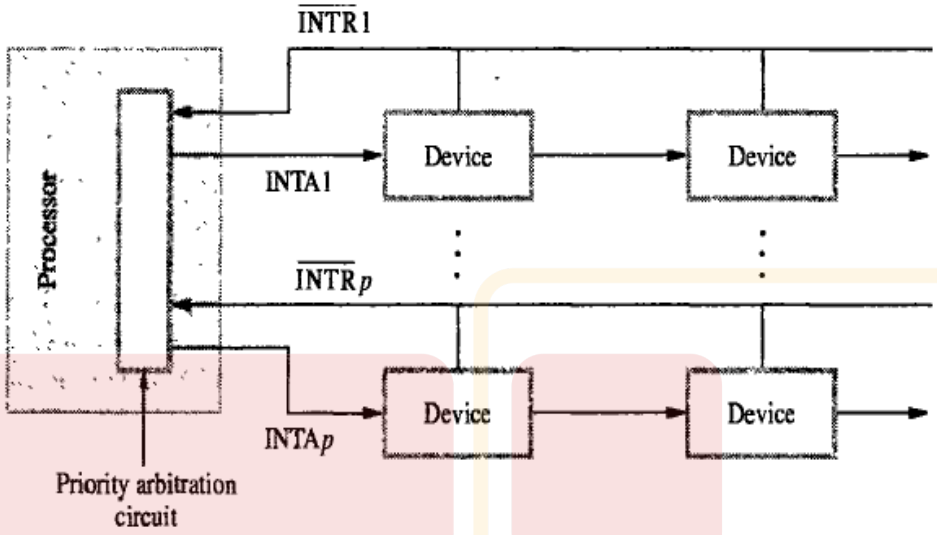
SIMULTANEOUS REQUESTS

DAISY CHAIN

- The daisy chain with multiple priority levels is as shown in the figure.
- The interrupt request line INTR is common to all devices as shown in the fig.
- The interrupt acknowledge line is connected in a daisy fashion as shown in the figure.
- This signal propagates serially from one device to another device.
- The several devices raise an interrupt by activating INTR signal. In response to the signal, processortransfers its device by activating INTA signal.
- This signal is received by device 1. The device-1 blocks the propagation of INTA signal to device-2,when it needs processor service.
- The device-1 transfers the INTA signal to next device when it does not require the processor service.
- In daisy chain **arrangement device-1 has the highest priority.**

Advantage: It requires fewer wires than the individual connection

ARRANGEMENT OF PRIORITY GROUPS



- In this technique, devices are organizes in a group and each group is connected to the processor at adifferent priority level.
- With in a group devices are connected in a daisy chain fashion as shown in the figure.

Direct Memory Access (DMA)

- It is the process of **transferring the block of data at high speed in between main memory and externaldevice (I/O devices) without continuous intervention of CPU is called as DMA.**
- The DMA operation is performed by one control circuit and is part of the I/O interface.
- This control circuit is called DMA controller. Hence DMA transfer operation is performed by DMAcontroller.
- To initiate Directed data transfer between main memory and external devices DMA controller needsparameters from the CPU.
- These 3 Parameters are:

1)Starting address of the memory block. 2)No of words to be transferred.
3)Type of operation (Read or Write).

After receiving these 3 parameters from CPU, DMA controller establishes directed data transferoperation between main memory and external devices without the involvement of CPU.

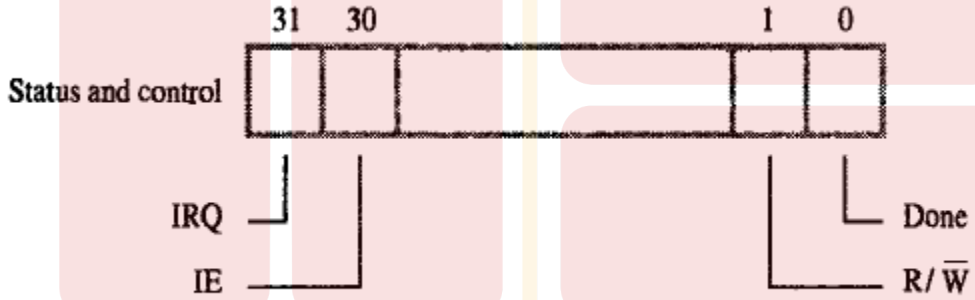
- **Register of DMA Controller:**
  It consists of 3 type of register:

Starting address register:
The format of starting address register is as shown in the fig. It is used to store the starting address of the memory block.

.

Word-Count register:
The format of word count register is as shown in fig. It is used to store the no of words to be transferred from main memory to external devices and vice versa.

Status and Controller register:
The format of status and controller register is as shown in fig.



a) DONE bit:

- The DMA controller sets this bit to 1 when it completes the direct data transfer between main memory and external devices.
- This information is informed to CPU by means of DONE bit.

b) R/W (Read or Write):

- This bit is used to differentiate between memory read or memory write operation.
- The R/W = 1 for read operation and
  = 0 for write operation.
- When this bit is set to 1, DMA controller transfers the one block of data from external device to main memory.
- When this bit is set to 0, DMA controller transfers the one block of data from main memory to external device.

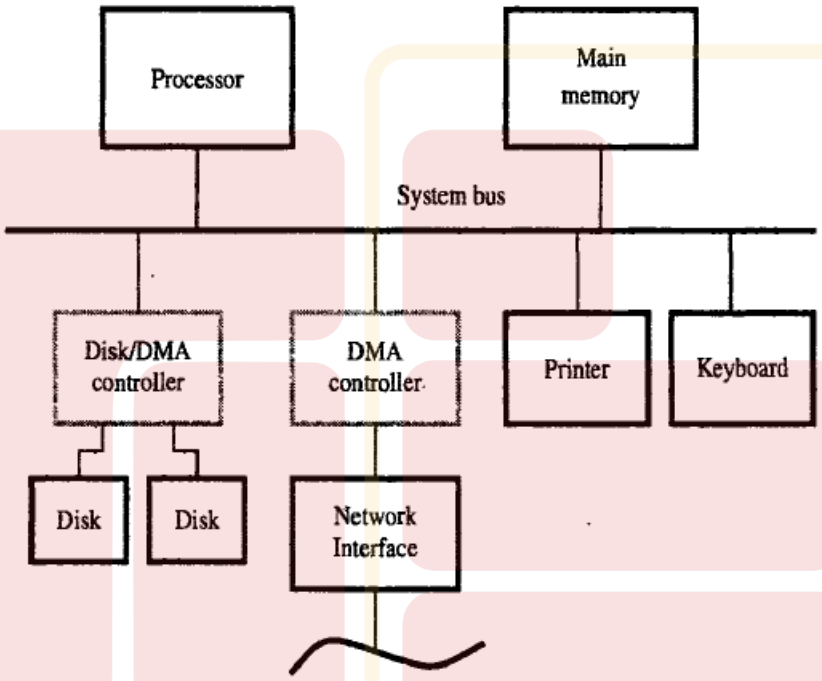
c) IE (Interrupt enable) bit:

- The DMA controller enables the interrupt enable bit after the completion of

DMA operation d)**Interrupt request (IRQ):**

- The DMA controller requests the CPU to transfer new block of data from source to destination by activating this bit.

The computer with DMA controller is as shown in the fig.:



- The DMA controller connects two external devices namely disk 1 and disk 2 to system bus as shown in the above fig.
- The DMA controller also interconnects high speed network devices to system bus as shown in the above fig.
- Let us consider direct data transfer operation by means of DMA controller without the involvement of CPU in between main memory and disk 1 as indicated by dotted lines (in the fig.).

- To establish direct data transfer operation between main memory and disk
  1. DMA controller request the processor to obtain **3 parameters** namely:
     1)**Starting address of the memory block.**
     2)No of words to be transferred.
     3)Type of operation (Read or Write).

- After receiving these 3 parameters from processor, DMA controller directly transfers block of data main memory and external devices (disk 1).

- This information is informed to CPU by setting respective bits in the status and controllerregister of DMA controller.
  These are 2 types of request with respect to
  system bus 1). CPU request.
  2). DMA request.
  Highest priority will be given to DMA request.

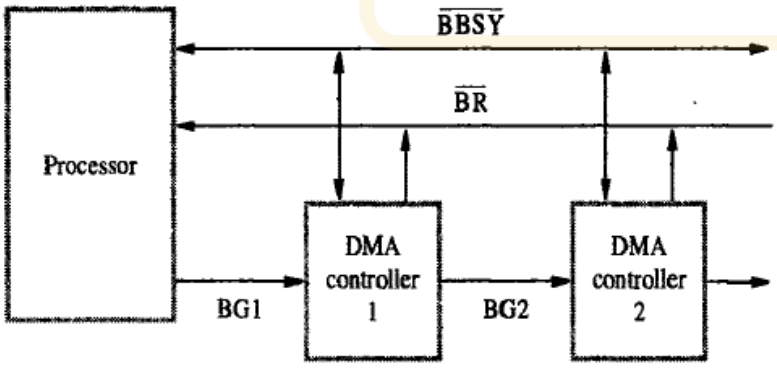
- Actually the CPU generates memory cycles to perform read and write operations. The DMA controller steals memory cycles from the CPU to perform read and write operations. This approach is called as **"Cycle stealing".**

- An exclusive option will be given for DMA controller to transfer block of data from external devices to main memory and from main memory to external devices. This technique is called as "**Burst mode of operation."**

BUS ARBITRATION

- Any device which initiates data transfer operation on bus at any instant of time is called as Bus-Master.
- When the bus mastership is transferred from one device to another device, the next device isready to obtain the bus mastership.
- The bus-mastership is transferred from one device to another device based on the principle ofpriority system. There are two types of bus-arbitration technique:

a)Centralized bus arbitration:
In this technique CPU acts as a bus-master or any control unit connected to bus can be acts as a busmaster.



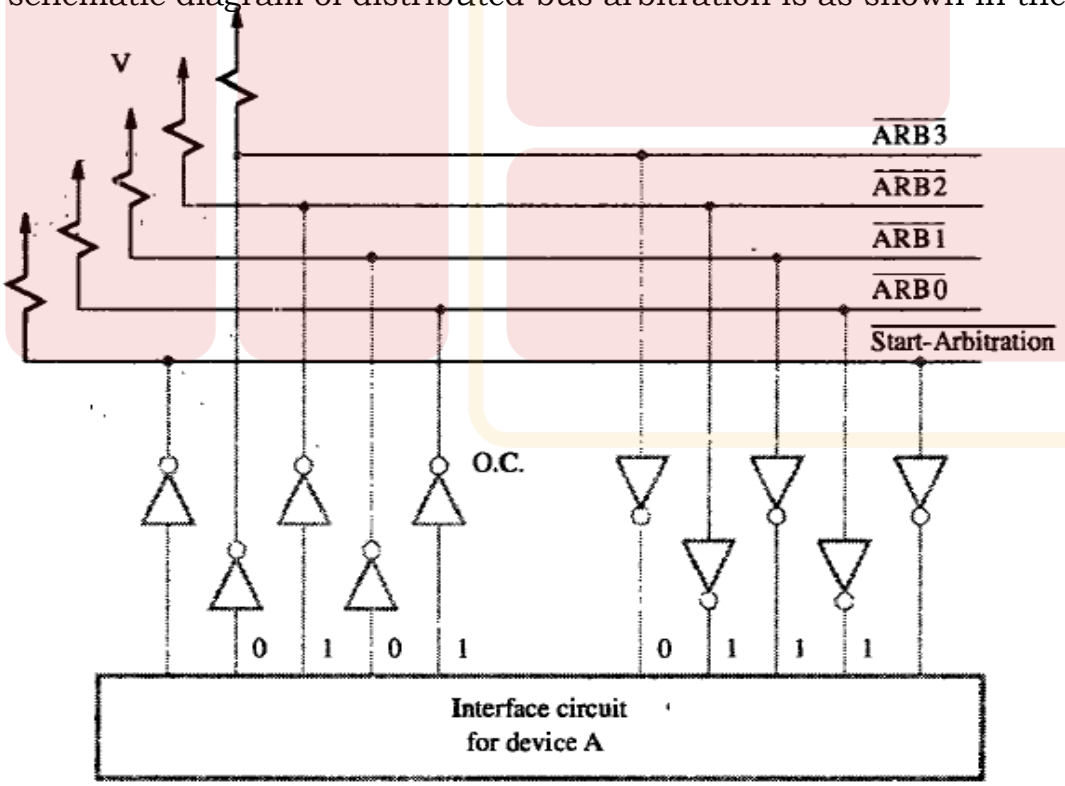
The schematic diagram of **centralized bus arbitration** is as shown in the fig.:

The following steps are necessary to transfer the bus mastership from CPU to one of the DMAcontroller:

- The DMA controller request the processor to obtain the bus mastership by activating BR (Busrequest) signal
- In response to this signal the CPU transfers the bus mastership to requested devices DMAcontroller1 in the form of BG (Bus grant).
- When the bus mastership is obtained from CPU the DMA controller1 blocks the propagation of busgrant signal from one device to another device.
- The BG signal is connected to DMA controller2 from DMA controller1 in as daisy fashion style isas shown in the figure.
- When the DMA controller1 has not sent BR request, it transfers the bus mastership to DMAcontroller2 by unblocking bus grant signal.
- When the DMA controller1 receives the bus grant signal, it blocks the signal from passing to DMA controller2 and enables BBSY signal. When BBSY signal is set to 1 the set of devices connected to system bus doesn't have any rights to obtain the bus mastership from the CPU.

b) Distributed bus arbitration:

- In this technique 2 or more devices trying to access system bus at the same time may participatein bus arbitration process.
- The schematic diagram of distributed bus arbitration is as shown in the figure



- The external device requests the processor to obtain bus mastership by enabling start arbitration signal.
- In this technique 4 bit code is assigned to each device to request the CPU in order to obtain bus mastership.
- Two or more devices request the bus by placing 4 bit code over the system bus.
- The signals on the bus interpret the 4 bit code and produces winner as a result from the CPU.
- When the input to the one driver = 1, and input to the another driver = 0, on the same bus line, this state is called as "Low level voltage state of bus".
- Consider 2 devices namely A & B trying to access bus mastership at the same time. Let assigned code for devices A & B are 5 (0101) & 6 (0110) respectively.
- The device A sends the pattern (0101) and device B sends its pattern (0110) to master. The signals on the system bus interpret the 4 bit code for devices A & B produces device B as a winner.
- The device B can obtain the bus mastership to initiate direct data transfer between external devices and main memory.

The Memory System

Speed, Size and Cost

The block diagram of memory hierarchy is as shown in the figure below.

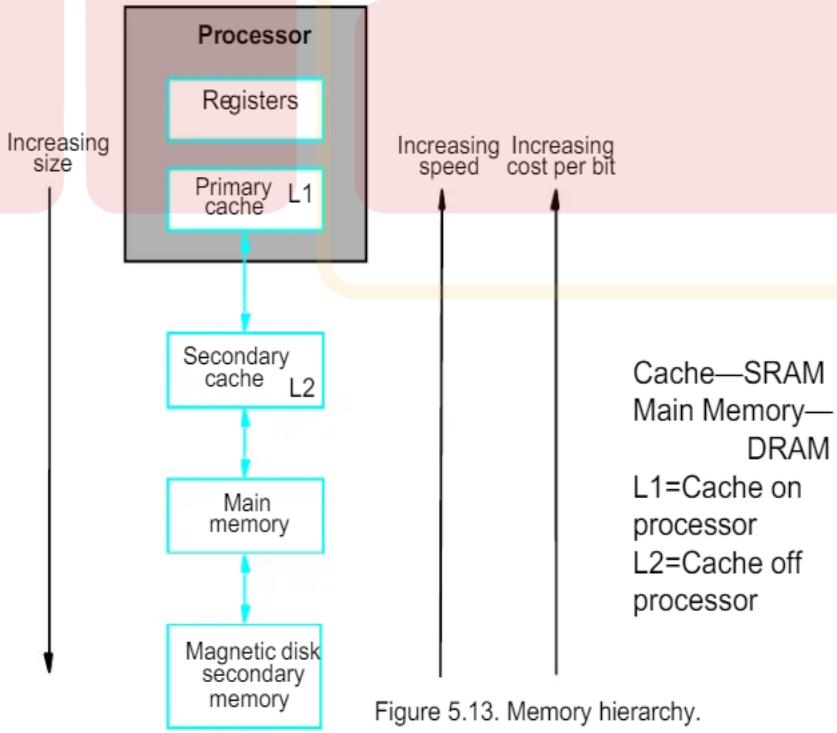


Figure 5.13. Memory hierarchy.

- Registers: The fastest access is to data held in registers. Hence registers are part of the memory hierarchy. More speed, small size and cost per bit is also more.
- At the next level of hierarchy, small amount of memory can be directly implemented on the processor chip.
- This memory is called as **processor cache**. **It holds the copy of recently accessed data and instructions.**

There are **2 levels of caches viz level-1 and level-2.**

Level-1 cache is **part of the processor** and **level-2 cache** is placed in **between level-1 cache and main memory.**

- The level-2 cache is implemented using SRAM chips
- The next level in the memory hierarchy is called as **main memory**. It is implemented using dynamic memory components (DRAM). **The main memory is larger but slower than cache memory.** The access time for main memory is ten times longer than the cache memory
- The level next in the memory hierarchy is called as **secondary memory**. It **holds huge amount of data.**

Characteristics	SRAM	DRAM	Magnetis Disk
Speed	Very Fast	Slower	Much slower than DRAM
Size	Large	Small	Small
Cost	Expensive	Less Expensive	Low price

Memory	Speed	Size
Registers	Very high	Lower
Primary cache	High	Lower
Secondary cache	Low	Low
Main memory	Lower than Seconadry cache	High
Secondary Memory	Very low	Very High

- The **main-memory is built with DRAM**
- **SRAMs are used in cache memory, where speed is essential.**
- The Cache-memory is of 2 types:

1) Primary/Processor Cache (Level1 or L1 cache)

➢ It is always located on the processor-chip.

2) Secondary Cache (Level2 or L2 cache)

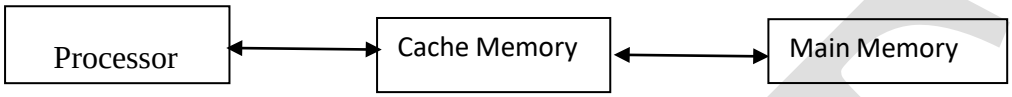
➢ It is placed between the primary-cache and the rest of the memory.

- The memory is implemented using the dynamic components (SIMM, DIMM).

The access time for main-memory is about 10 times longer than the access time for L1 cache.

Cache Memory

It is the **fast access memory located in between processor and main memory**



as shown in the fig. It is designed to reduce the access time.

The cache memory holds the copy of recently accessed data and instructions.

- The processor needs less access time to read the data and instructions from the cache memory as compared to main memory .
- Hence by incorporating cache memory, in between processor and main memory, it is possible to enhance the performance of the system.

- The effectiveness of cache mechanism is based on the property of "**Locality of Reference".**

Locality of Reference

- Many instructions in the localized areas of program are executed repeatedly during some time of execution
- Remainder of the program is accessed relatively infrequently
- There are 2 types of locality reference:

1) **Temporal**

  - The recently executed instructions are likely to be executed again and again.
  - Eg –instruction in loops, nested loops and few function calls.

2) **Spatial**

  - Instructions in close proximity to recently executed instruction are likely to be executed soon. (near by instructions)

- If active segment of program is placed in cache-memory, then total execution time can be reduced.
- **Cache Block / cache line** refers to the set of contiguous address locations of some size.
- The Cache-memory stores a reasonable number of blocks at a given time.

• This number of blocks is small compared to the total number of blocks available in main-memory.

• Correspondence b/w main-memory-block & cache-memory-block is specified by **mapping-function.**

• If the cache memory is full, one of the block should be removed to create space forthe new block, this is decided by **cache control hardware.**

• The collection of rule for selecting the block to be removed is called the **Replacement Algorithm.**

• The cache control-circuit determines whether the requested-word currently exists in the cache.

• If data is available, for read-operation, the data is read from cache.

• The **write-operation** (writing to memory) is done in **2 ways:**

1**) Write-through protocol &**

2) **Write-back protocol.**

Write-Through Protocol

➢ Here the cache-location and the main-memory-locations are updated simultaneously.

Write-Back Protocol

➢ This technique is to

→ update only the cache-location &

→ mark the cache-location with a flag bit called **Dirty/Modified Bit.**

➢ The word in memory will be updated later, when the marked-block is removed from cache.

During Read-operation

• If the requested-word currently does not exists in the cache, then **read-miss** will occur.

• To overcome the read miss, *Load–through/Early restart protocol* is used.

Load–Through Protocol

➢ The block of words that contains the requested-word is copied from the memory into cache.

➢ After entire block is loaded into cache, the requested-word is forwarded to processor.

➢

During Write-operation

- If the requested-word does not exists in the cache, then **write-miss** will occur.
  1) If **Write Through Protocol** is used, the information is written directlyinto main-memory.
  2) If **Write Back Protocol** is used,
     → then block containing the addressed word is first brought into the cache&
     → then the desired word in the cache is over-written with the newinformation.

Mapping functions

There are 3 techniques to map main memory blocks into cache memory –

1. Direct mapped cache
2. Associative Mapping
3. Set-Associative Mapping

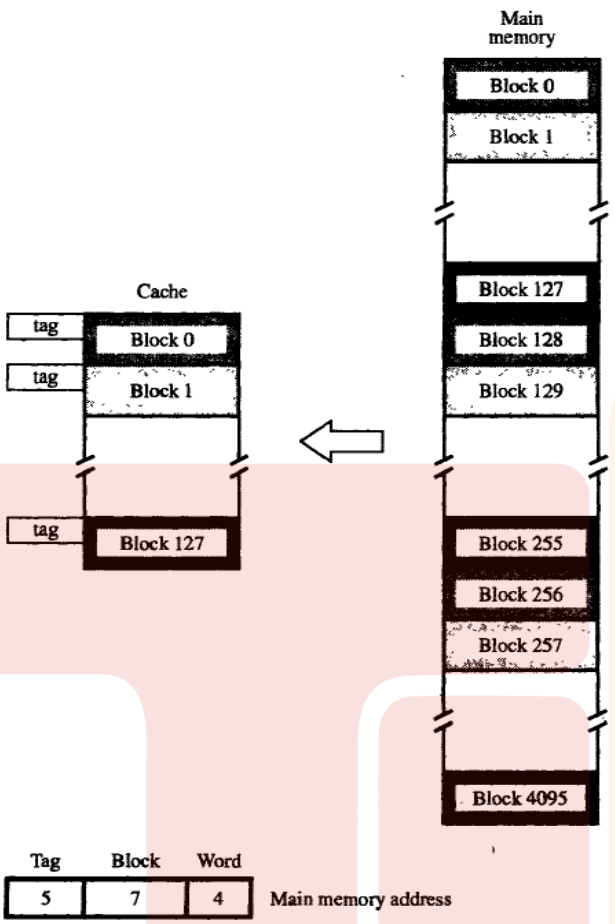
DIRECT MAPPING

- The simplest way to determine cache locations in which to store memory blocks is the direct mapping technique as shown in the figure.

- Cache block number= (block-j of main memory)%128;

- If there are 128 blocks in a cache, the block-j of the main-memory maps onto block-jmodulo-128 of the cache . When the memory-blocks 0, 128, & 256 are loaded into cache, the block is stored in cache-block 0. Similarly, memory-blocks 1, 129, 257 are stored in cache-block 1.(eg:1mod 128=1, 129 mod 128=1)
- The contention may arise
  1) Even when the cache is full.
  2) But more than one memory-block is mapped onto a given cache-block position.
- The contention is resolved by allowing the new blocks to overwrite the currently resident-block.

Memory-address determines placement of block in the cache.



main memory block has to be placed in particular cache block number by using below formula
Cache block number=main memory block number % number of blocks present in cache memory.

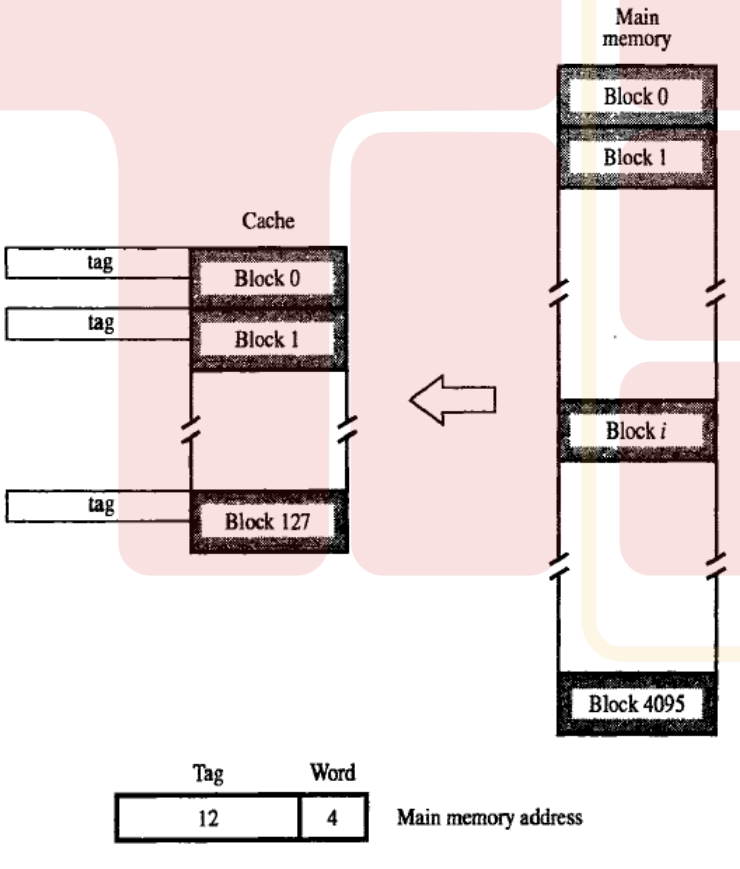
For eg: main memory block 129 has to be placed in cache block number 1 by using above formula i.e
Cache block number=129 % 128 (consider remainder that is 1).
Cache block number=258 % 128 (consider remainder that is 2).
Main memory block 258 has to be placed in cache block 2

- The main memory block is loaded into cache block by means of **memory address.** The main memory address consists of 3 fields as shown in the figure.
- **Each block consists of 16 words**. Hence least significant **4 bits** are **used to select one of the 16 words.**
- The **7bits of memory address are used to specify the position of the cache block**, location. The most significant **5 bits of the memory address are stored in the tag bits**. **The tag bits are used to map one of 2^5 = 32 blocks into cache block location** (tag bit has value 0-31).
- **The higher order 5 bits of memory address are compared with the tag bits** associated with cache location**. If they match**, then the desired word is in that block of the cache.
- **If there is no match**, then the block containing the required word must first be read from the main memory

and loaded into the cache. It is very easy to implement, but not flexible.

2. Associative Mapping:

- It is also called as associative mapped cache. It is much more flexible.
- In this technique main memory block can be placed into **any cache block position**.
- In this case , 12 tag bits are required to identify a memory block when it is resident of the cache memory.
- The Associative Mapping technique is illustrated as shown in the fig.



- In this technique 12 bits of address generated by the processor are compared with the tag bits of each block of the cache to see if the desired block is present. This is called as associative mapping technique.

3. Set Associative Mapping:

- It is the combination of **direct and associative mapping techniques.**
- The blocks of cache are divided into several groups. Such a groups are called as **sets.**
- Each set consists of number of cache blocks. A memory block is loaded into one of the cache sets.
- The **main memory address consists of three fields**, as shown in the figure.
- The **lower 4 bits of memory address** are used to **select a word from a 16 words.**
- A cache consists of **64 sets** as shown in the figure. Hence **6 bit set field** is used to select a **cache set from 64 sets.**
- As there are 64 sets, the memory is divided into groups containing 64 blocks, where each group is given a tag number.
- The most significant **6 bits of memory address is compared with the tag fields of each set to determine whether memory block is available or not.**
- The following figure clearly describes **the working principle of Set Associative Mapping technique.**

• cache that has "k" blocks per set is called as "**k-way set associative cache**".

• Each block contains a control-bit called a **valid-bit**.

• The Valid-bit indicates that whether the block contains valid-data (updated data).

• The dirty bit indicates that whether the block has been modified during its cache residency.

Valid-bit=0 - When power is initially applied to system.

Valid-bit=1 - When the block is loaded from main-memory at first time.

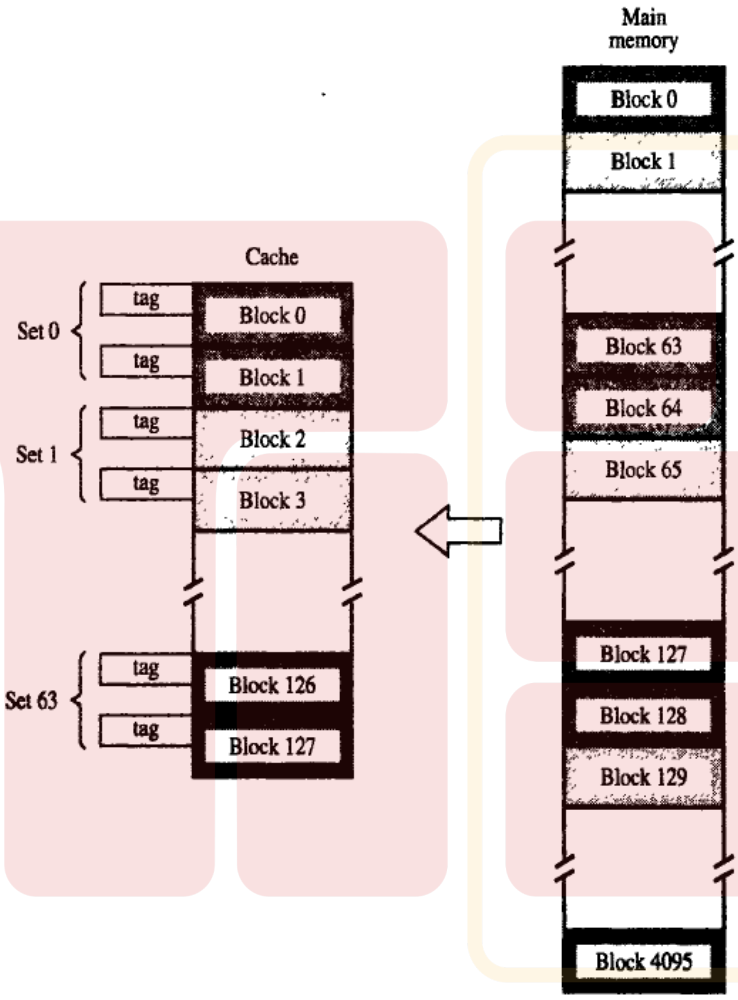
• If the main-memory-block is updated by a source & if the block in the source is already exists in the cache, then the valid-bit will be cleared to "0".

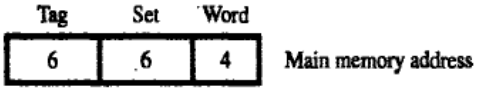
• If Processor & DMA uses the same copies of data then it is called as **Cache Coherence Problem**.

• **Advantages:**

1) Contention problem of direct mapping is solved by having few choices for block placement.

2) The hardware cost is decreased by reducing the size of associative search.





MODULE 5:

Basic Processing Unit and Pipelining

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.

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**, **processo**r 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 ← [[PC]]
  2. Increment PC by
     4. PC ←[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.**

SINGLE BUS ORGANIZATION

- Here the processor contain only a single bus for the movement of data, address and instructions.
- ALU and all the registers are interconnected via a **Single Common Bus** (Figure 7.1).
- 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:

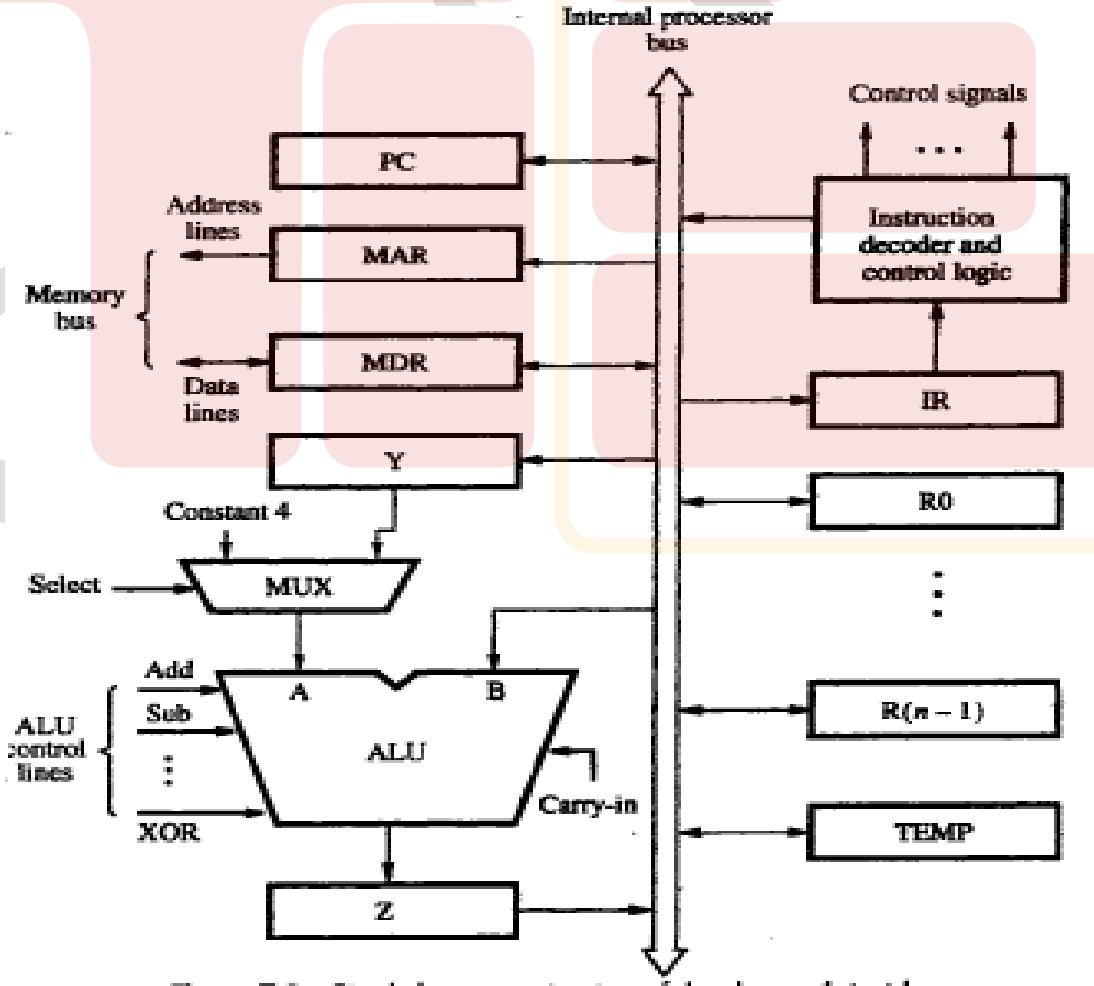


Figure 7.1 Single-bus organization of the datapath inside a processor.

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.

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: Riin & Riout. These are called **Gating Signals**
- Riin=1,the data on the bus are loaded into Ri,
- Riout=1,the contents of register are placed on the bus,
- Riout=0,the bus can be used for transferring data from other registers.

Suppose we wish to transfer the contents of register R1 to register R2. This can be accomplished as follows:

1. Enable the output of registers R1 by setting R1out to 1 (Figure 7.2). This places the contents of R1 on processor-bus.
2. Enable the input of register R4 by setting R4in to 1. This loads data from processor-bus into register R4.

- All operations and data transfers within the processor take place within time-periods defined by the **processor-clock.**

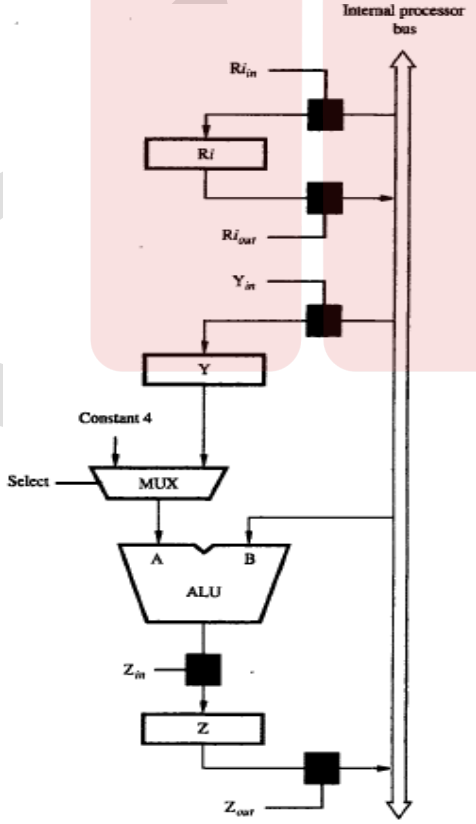


Figure 7.2 Input and output gating for the registers in Figure 7.1.

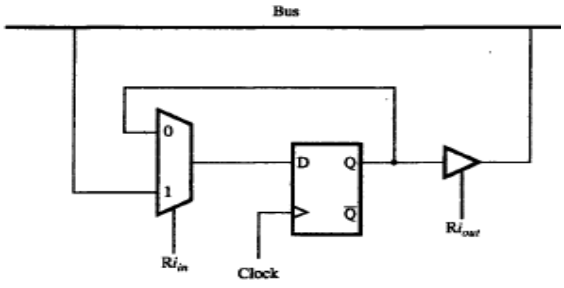


Figure 7.3 Input and output gating for one register bit.

- The control-signals that govern a particular transfer are asserted at the start of the clock cycle.

Input & Output Gating for one Register Bit
Implementation for one bit of register Ri(as shown in fig 7.3)

➨ All operations and data transfers are controlled by the **processor clock.**

- A 2-input multiplexer is used to select the data applied to the input of an edge-triggered D flip-flop.
- Riin=1,Multiplexer selects data on the bus. This data will be loaded into flip-flop at rising-edge of clock.
- Riin=0,Multiplexer feeds back the value currently stored in the flipflop
- Q output of flip-flop is connected to bus via a tri-state gate.
- When Riout=0, gates output in the high-impedance state.
- When Riout=1,gate drives the bus to 0 or 1,depending on the value of Q.

PERFORMING AN ARITHMETIC OR LOGIC OPERATION(refer fig:7.2)

- **The ALU is a combinational circuit that has no internal storage.**
- The ALU performs arithmetic and logic operations on the 2 operands applied to its A and B inputs.
- ALU gets the two operands, one is from MUX and another from bus. The result is temporarily stored in register Z.
- Therefore, a sequence of operations [R3]=[R1]+[R2].

1) R1out, Yin
2) R2out, Select Y, Add, Zin
3) Zout, R3in

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.

FETCHING A WORD FROM MEMORY

- To fetch instruction/data from memory, the processor has to specify the address of the memory location where this information is stored and request a Read operation.
- 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 in the processor.

The Connections for register MDR has shown in fig 7.4

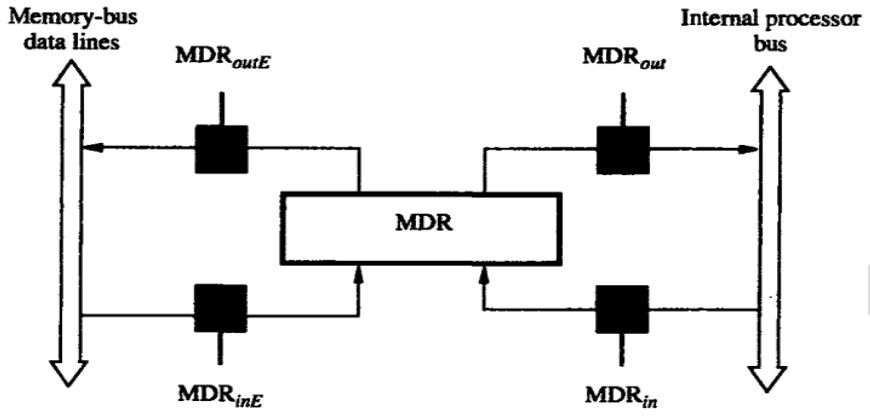


Figure 7.4 Connection and control signals for register MDR.

CONTROL-SIGNALS OF MDR

- The **MDR register has 4 control-signals** (Figure 7.4):

1) **MDRin & MDRout** control the connection to the **internal** processor data bus &
2) **MDRinE & MDRoutE** control the connection to the **external** memory Data bus.

- **Similarly, MAR register has 2 control-signals.**

1) **MARin:** controls the connection to the internal processor address bus &
2) **MARout:** controls the connection to the memory address bus.

The response time of each memory access varies. To accommodate this MFC is used(MFC= Memory Function Completed)

MFC=1 indicate that contents of specified location have been read and are available on the data lines of the memory bus.

- Consider **the instruction Move (R1),R2**. The action needed to execute this instruction are

1. MAR ← [R1]
2. Start a Read operation on the memory bus
3. Wait for the MFC response from the memory
4. Load MDR from the memory bus
5. R2 ← [MDR]

The sequence of steps is (Figure 7.5):

1) R1$_{out}$,MAR$_{in}$,Read ;desired address is loaded into MAR & Read command is issued.
2) MDR$_{inE}$,WMFC; load MDR from memory-bus & Wait for MFC response from memory.
3) MDR$_{out}$, R2$_{in}$; load R2 from MDR.

where WMFC=control-signal that causes processor's control. circuitry to wait for arrival of MFC signal.

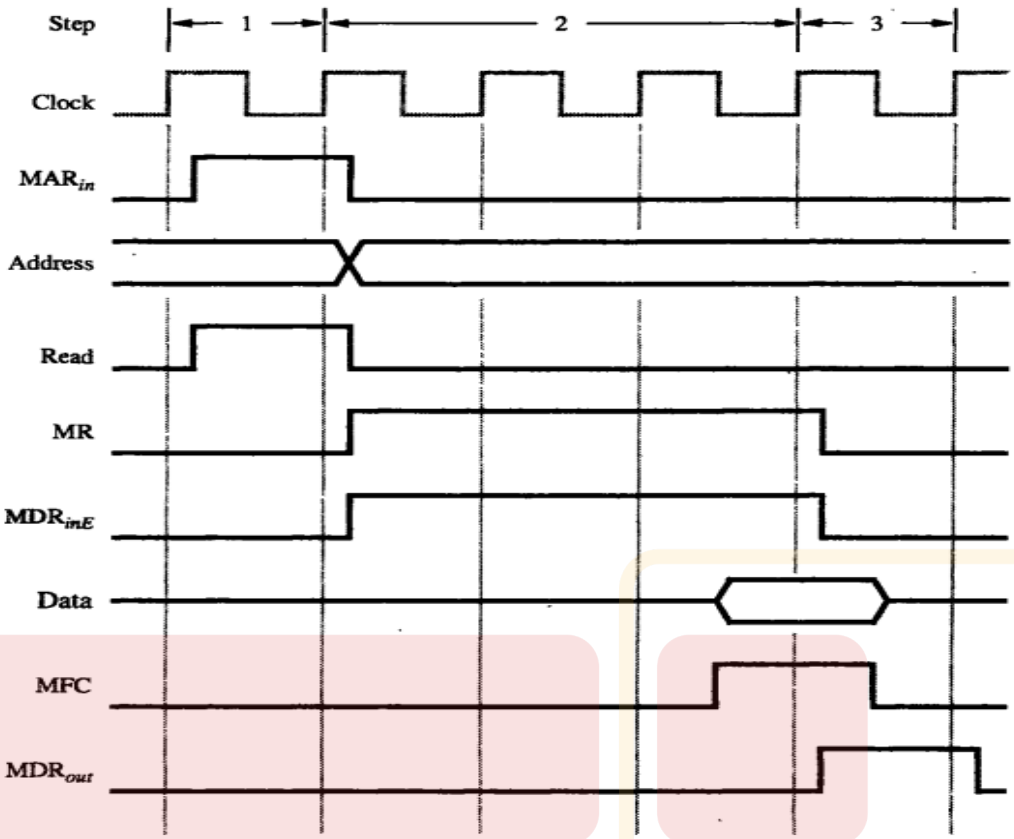


Figure 7.5 Timing of a memory Read operation.

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_{outE}, WMFC ;load data into memory-location pointed by R1 from MDR.

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.**

Fig:7.6 gives the sequence of control steps required to perform these operations for the single -bus architecture .

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

- **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,
- **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.

Pipelining:

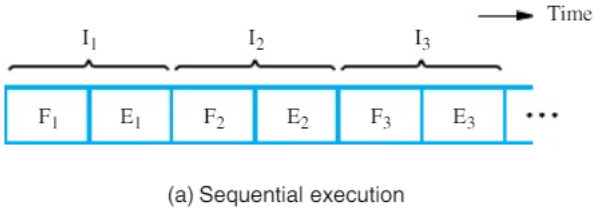
Basic Concepts:

The **speed of execution of programs is influenced by many factors**.

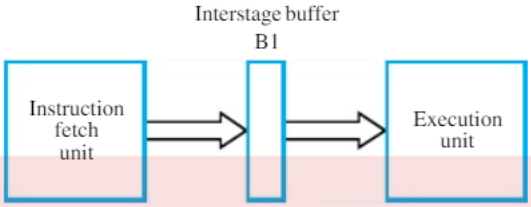
- **One way to improve** performance is to use **faster circuit technology to build the processor and the main memory.** Another possibility is to **arrange the hardware so that more than one operation can be performed at the same time.** In this way, **the number of operations performed per second is increased even though the elapsed time needed to perform any one operation is not changed.**
- **Pipelining is a particularly effective way of organizing concurrent activity in a computer system.**
- The technique of **decomposing** a sequential process into sub-operations, with each sub-operation being executed in a dedicated segment .
- **pipelining** is commonly known as an **assembly-line** operation.

Consider **how the idea of pipelining can be used in a computer**. The **processor executes a program by fetching and executing instructions, one after the other**.

Let **Fi and Ei refer to the fetch and execute steps for instruction Ii** . Execution of a program consists of a sequence of fetch and execute steps, as shown in Figure a.



(a) Sequential execution



(b) Hardware organization

Now consider a computer that has **two separate hardware units**, **one** for **fetching instructions** and **another for executing** them, as shown in Figure b. The instruction fetched by the fetch unit is deposited in an **intermediate storage buffer, B1**. This **buffer is needed to enable the execution unit** to execute the instruction while the fetch unit is fetching the next instruction. The results of execution are deposited in the destination location specified by the instruction.

The computer is controlled by a clock.

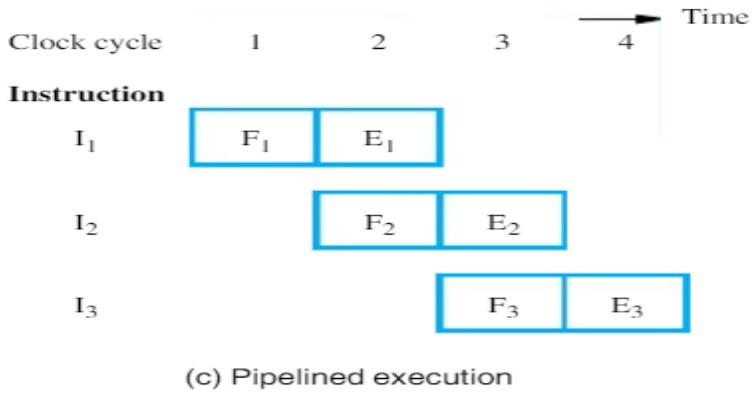
any instruction **fetch and execute** steps completed in **one clock cycle.**

Operation of the computer proceeds as in Figure 8.1c.

In the **first clock cycle**, the fetch unit fetches an instruction I1 (step F1) and stores it in buffer B1 at the end of the clock cycle.

In the **second clock cycle**, the instruction fetch unit proceeds with the fetch operation for instruction I2 (step F2). Meanwhile, the execution unit performs the operation specified by instruction I1, which is available to it in buffer B1 (step E1). By the end of the second clock cycle, the execution of instruction I1 is completed and instruction I2 is available. Instruction I2 is stored in B1, replacing I1, which is no longer needed.

Step E2 is performed by the execution unit during the **third clock cycle,** while instruction I3 is being fetched by the fetch unit. In this manner, both the fetch and execute units are kept busy all the time. If the pattern in Figure 8.1c can be sustained for a long time, the completion rate of instruction execution will be twice that achievable by the sequential operation depicted in Figure a.



(c) Pipelined execution

Figure 8.1 Basic idea of instruction pipelining.

Idea of Pipelining in a computer

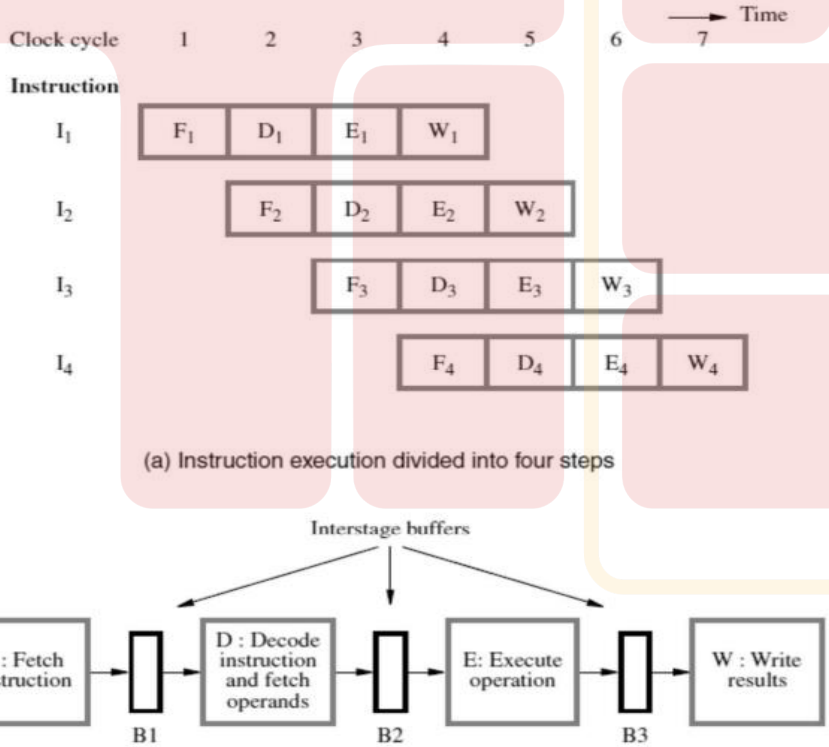
a pipelined processor may process each instruction in four steps, as follows:

F (**Fetch**): **read the instruction from the memory.**

D (Decode): decode the instruction and fetch the source operand(s).

E (Execute): perform the operation specified by the instruction.

W (Write): store the result in the destination location.



(a) Instruction execution divided into four steps

(b) Hardware organization

The sequence of events for this case is shown in Figure a. Four instructions are in progress at any given time. This means that four distinct hardware units are needed, as shown in Figure b. These units must be capable of performing their tasks simultaneously and without interfering with one another. Information is passed from one unit to the next through a storage buffer. As an instruction progresses through the pipeline, all the information needed by the stages

downstream must be passed along. For example, during clock cycle 4, the information in the buffers is as follows:

- Buffer B1 holds instruction I3, which was fetched in cycle 3 and is being decoded by the instruction-decoding unit.
- Buffer B2 holds both the source operands for instruction I2 and the specification of the operation to be performed. This is the information produced by the decoding hardware in cycle 3. The buffer also holds the information needed for the write step of instruction I2 (stepW2). Even though it is not needed by stage E, this information must be passed on to stage W in the following clock cycle to enable that stage to perform the required Write operation.
- Buffer B3 holds the results produced by the execution unit and the destination information for instruction I1.

Role of Cache Memory

Each stage in a pipeline is expected to complete its operation in **one clock cycle**. Hence, the **clock period should be sufficiently long to complete the task being performed in any stage.** If different units require different amounts of time, the clock period must allow the longest task to be completed. A unit that completes its task early is idle for the remainder of the clock period. Hence**, pipelining** is most effective in **improving performance** if the tasks being performed in different stages require about the same amount of time. This consideration is particularly important for the instruction fetch step, which is assigned one clock period in Figure a. The clock cycle has to be equal to or greater than the time needed to complete a fetch operation. However, the **access time of the main memory** may be as much as ten times greater than the time needed to perform basic pipeline stage operations inside the processor, such as adding two numbers. Thus, **if each instruction fetch required access to the main memory, pipelining would be of little value.**

The **use of cache memories solves the memory access problem**. In particular, when a cache is included on the same chip as the processor, access time to the cache is usually the same as the time needed to perform other basic operations inside the processor. This makes it possible to divide instruction fetching and processing into steps that are more or less equal in duration. Each of these steps is performed by a different pipeline stage, and the clock period is chosen to correspond to the longest one.

Pipeline Performance:

- The potential increase in performance resulting from pipelining is proportional to **the number of pipeline stages**.
- However, this increase would be achieved only if pipelined operation as depicted in Figure a could be sustained without interruption throughout program execution.
- Unfortunately, this is not the True.
- Floating point may involve many clock cycle.
- For a variety of reasons, one of the pipeline stages may not be able to complete its processing task for a given instruction in the time allotted. For example, stage E in the four stage pipeline of Figure b is responsible for arithmetic and logic operations, and one clock cycle is assigned for this task. Although this may be sufficient for most operations, some operations, such as divide, may require more time to complete. Figure shows an example in which the operation specified in instruction I2 requires three cycles to complete, from cycle 4 through cycle 6. Thus, in cycles 5 and 6, the Write stage must be told to do nothing, because it has no data to work with. Meanwhile, the information in buffer B2 must remain intact until the Execute stage has completed its operation. This means that stage 2 and, in turn, stage 1 are blocked from accepting new instructions because the information in B1 cannot be overwritten. Thus, steps D4 and F5 must be postponed as shown.

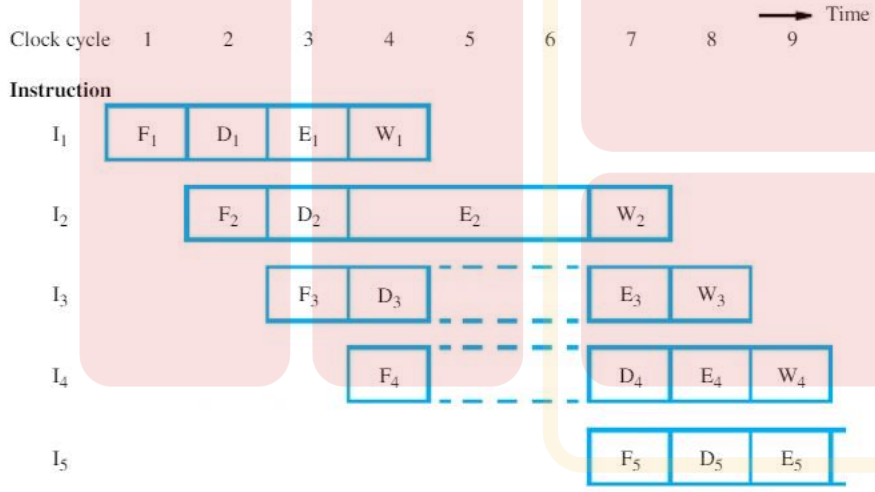


Figure 8.3 Effect of an execution operation taking more than one clock cycle.

Eg: for Data Hazard

Pipelined operation in Figure 8.3 is said to have been **stalled for two clock cycles**. Normal pipelined operation resumes in cycle 7. Any condition that causes the pipeline to stall is called **a hazard**. We have just seen an example of a data hazard.

1) **A data hazard is any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. As a result some operation has to be delayed, and the pipeline stalls.**

2) **control hazards** or **instruction hazards**: The pipeline may also be stalled because of a **delay in the availability of an instruction**.

For example, this may be a **result of a miss in the cache** .

3) A **third type of hazard** known as a **structural hazard**: **This is the situation when two instructions require the use of a given hardware resource at the same time.**

The effect of a cache miss on pipelined operation is illustrated in Figure. Instruction I1 is fetched from the cache in cycle 1, and its execution proceeds normally. However, the fetch operation for instruction I2, which is started in cycle 2, results in a cache miss. The instruction fetch unit must now suspend any further fetch requests and wait for I2 to arrive. We assume that instruction I2 is received and loaded into buffer B1 at the end of cycle 5. The pipeline resumes its normal operation at that point.

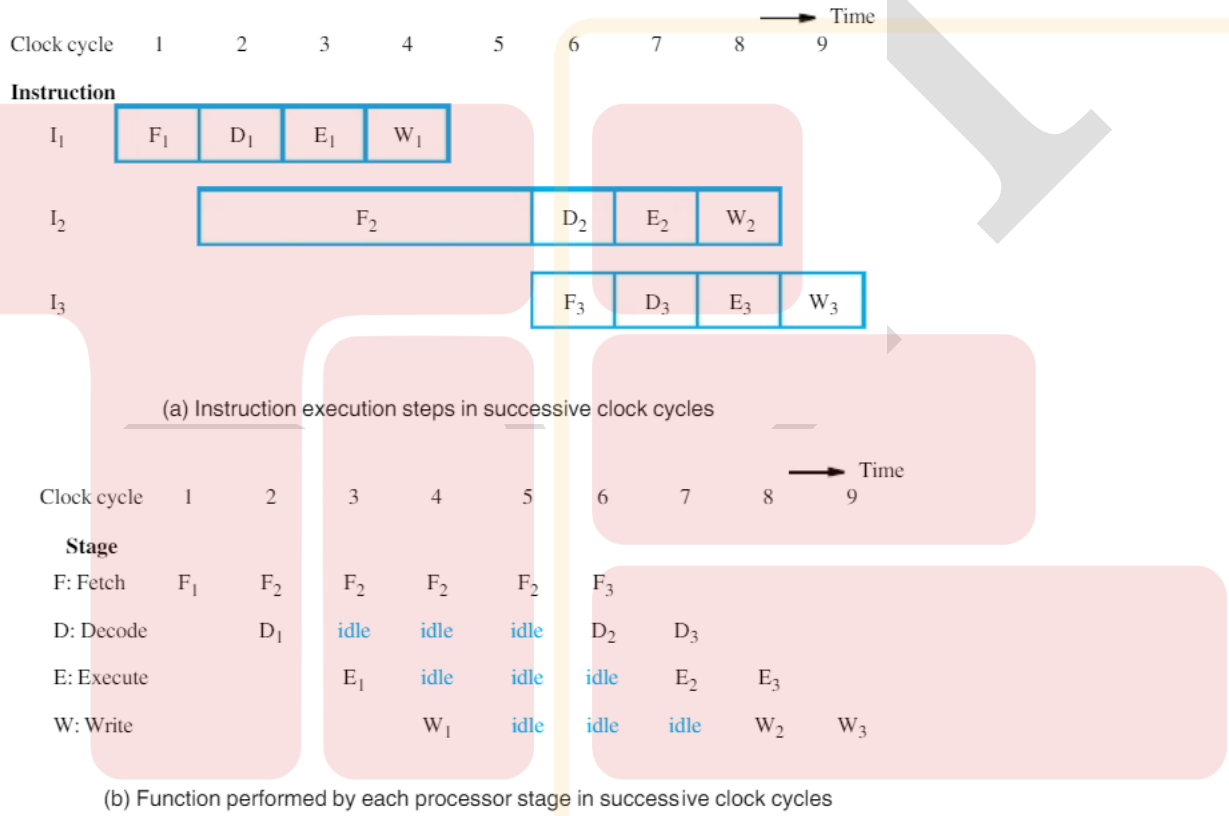


Figure 8.4 Pipeline stall caused by a cache miss in F2. **Eg: for Instruction Hazard**

An alternative representation of the operation of a pipeline in the case of a cache miss is shown in Figure b. This figure gives the function performed by each pipeline stage in each clock cycle. Note that the **Decode unit is idle in cycles 3 through 5,** the Execute unit is idle in cycles 4 through 6, and the **Write unit is idle in cycles 5 through 7**. Such **idle periods are** called **stalls.** They are **also often referred** to as **bubbles in the pipeline**.

If instructions and data reside in the same cache unit, only one instruction can proceed and the other instruction is delayed. Many processors use separate instruction and data caches to avoid this delay.

An **example of a structural hazard** is shown in Figure. This figure shows how the load instruction

Load X(R1),R2

- The memory address, X+[R1], is computed in stepE2 in cycle 4, then memory access takes place in cycle 5. The operand read from memory is written into register R2 in cycle 6. This means that the execution step of this instruction takes two clock cycles (cycles 4 and 5). It causes the pipeline to stall for one cycle, because both instructions I2 and I3 require access to the register file in cycle 6.
- Even though the instructions and their data are all available, the pipeline is stalled because **one hardware resource**, the register file, cannot handle two operations at once. If the register file had two input ports, that is, if it allowed two simultaneous write operations, the pipeline would not be stalled. In general, **structural hazards are avoided by providing sufficient hardware resources on the processor chip.**

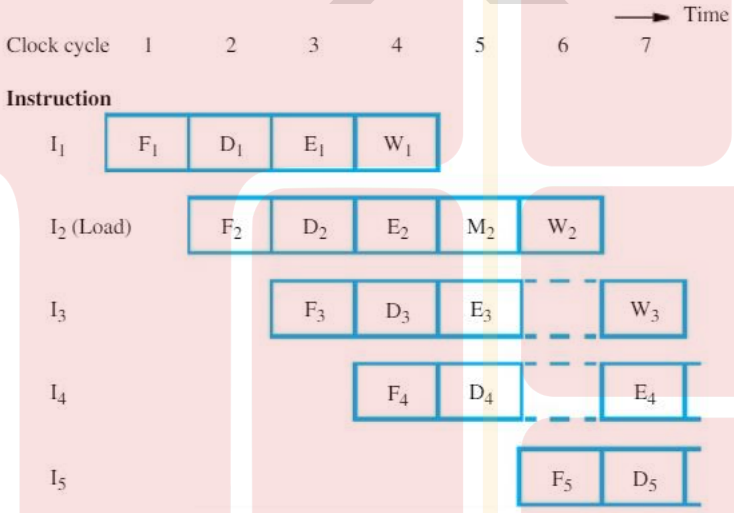


Figure 8.5 Effect of a Load instruction on pipeline timing.

It is important to understand that **pipelining does not result in individual instructions** being executed faster; rather, it is the throughput that increases, where **throughput is measured by the rate at which instruction execution is completed.**

The **pipeline stalls, causes degradation in pipeline performance.**

We need to identify all hazards that may cause the pipeline to stall and to find ways to minimize their impact.