Unit I Boolean Algebra and Logic Gates

1. Introduction
2. Basic Definitions
3. Axiomatic Definition of Boolean Algebra
4. Basic Theorems and Properties of Boolean Algebra
5. Boolean Functions
6. Canonical and Standard Forms
7. Other Logic Operations
8. Digital Logic Gates
9. Integrated Circuits

Basic Definitions

- Mathematical methods that simplify binary logics or circuits rely primarily on Boolean algebra.
- Boolean algebra: a set of elements, a set of operators, and a number of unproved axioms or postulates.
- A *set* of elements is any collection of objects, usually having a common property. A = {1, 2, 3, 4} indicates that set A has the elements of 1, 2, 3, and 4.
- A *binary operator* defined on a set S of elements is a rule that assigns, to each pair of elements from S, a unique element from S.
- The most common postulates used to formulate various algebraic structures are as follows:

1. ***Closure.*** A set S is closed with respect to a binary operator if, for every pair of elements of S, the binary operator specifies a rule for obtaining a unique element of S.
2. ***Associative law.*** A binary operator * on a set S is said to be associative whenever $(x * y) * z = x * (y * z)$ for all $x, y, z, \in S$
3. ***Commutative law.*** A binary operator * on a set S is said to be commutative whenever $x * y = y * x$ for all $x, y \in S$

4.Identity element. A set S is said to have an identity element with respect to abinary operation * on S if there exists an element $e \in S$ with the property that

$$e * x = x * e = x \text{ for every } x \in S$$

Example: The element 0 is an identity element with respect to the binary operator + on the set of integers $I = \{c, -3, -2, -1, 0, 1, 2, 3, c\}$, since $x + 0 = 0 + x = x$ for any $x \in I$

The set of natural numbers, N, has no identity element, since 0 is excluded from the set.

5.Inverse. A set S having the identity element e with respect to a binary operator

* is said to have an inverse whenever, for every $x \in S$, there exists an element $y \in S$ such that $x * y = e$

Example: In the set of integers, I, and the operator +, with $e = 0$, the inverse of an element a is $(-a)$, since $a + (-a) = 0$.

6.Distributive law. If * and • are two binary operators on a set S, * is said to be distributive over • whenever $x * (y \bullet z) = (x * y) \bullet (x * z)$

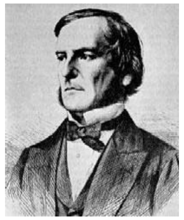
Field

- A *field* is an example of an algebraic structure.
- The field of real numbers is the basis for arithmetic and ordinary algebra.
  - The binary operator + defines addition.
  - The additive identity is 0.
  - The additive inverse defines subtraction.
  - The binary operator • defines multiplication.
  - The multiplicative identity is 1.
  - For $a \neq 0$, the multiplicative inverse of $a = 1/a$ defines division (i.e., $a \bullet 1/a = 1$).
  - The only distributive law applicable is that of • over +:

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

Axiomatic Definition of Boolean Algebra

- 1854: George Boole developed an algebraic system now called *Boolean algebra.*
- 1904: E. V. Huntington formulated a set of postulates that formally define the Boolean algebra
- 1938: C. E. Shannon introduced a two-valued Boolean algebra called switching algebra that represented the properties of bistable electrical switching circuits



- Two binary operators, + and •, (Huntington) postulates:
  1. (a) The structure is closed with respect to the operator +.
     (b) The structure is closed with respect to the operator •.
  2. (a) The element 0 is an identity element with respect to +; that is, $x + 0 = 0 + x = x$.
     (b) The element 1 is an identity element with respect to •; that is, $x \bullet 1 = 1 \bullet x = x$.
  3. (a) The structure is commutative with respect to +; that is, $x + y = y + x$.
     (b) The structure is commutative with respect to • ; that is, $x \bullet y = y \bullet x$.
  4. (a) The operator • is distributive over +; that is, $x \bullet (y + z) = (x \bullet y) + (x \bullet z)$.
     (b) The operator + is distributive over •; that is, $x + (y \bullet z) = (x + y) \bullet (x + z)$.

5. For every element $x \in B$, there exists an element $x \in B$ (called the complement of x) such that (a) $x + x = 1$ and (b) $x \bullet x = 0$.
6. There exist at least two elements $x, y \in B$ *such that* $x \neq y$.

- Comparing Boolean algebra with arithmetic and ordinary algebra

1. Huntington postulates do not include the associative law. However, this law holds for Boolean algebra and can be derived (for both operators) from the other postulates.
2. The distributive law of + over • (i.e., $x + (y \bullet z) = (x + y) \bullet (x + z)$) is valid for Boolean algebra, but not for ordinary algebra.
3. Boolean algebra does not have additive or multiplicative inverses; therefore, there are no subtraction or division operations.
4. Postulate 5 defines an operator called the complement that is not available in ordinary algebra.
5. Ordinary algebra deals with the real numbers, which constitute an infinite set of elements. Boolean algebra deals with the as yet undefined set of elements, B, but in the two-valued Boolean algebra defined next (and of interest in our subsequent use of that algebra), B is defined as a set with only two elements, 0 and 1.

Two-valued Boolean Algebra

- B = {0,1}
- The rules of operations

AND

x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

OR

x	y	$x+y$
0	0	0
0	1	1
1	0	1
1	1	1

NOT

x	x'
0	1
1	0

- Closure: the result of each operation is either 1 or 0 and 1, 0 $\in B$.
- Identity elements: 0 for + and 1 for ·
- The commutative laws are obvious from the symmetry of the binary operator tables.

- Distributive laws:
  - $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$
  - $x + (y \cdot z) = (x + y) \cdot (x + z)$

x	y	z	$y + z$	$x \cdot (y + z)$	$x \cdot y$	$x \cdot z$	$(x \cdot y) + (x \cdot z)$
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

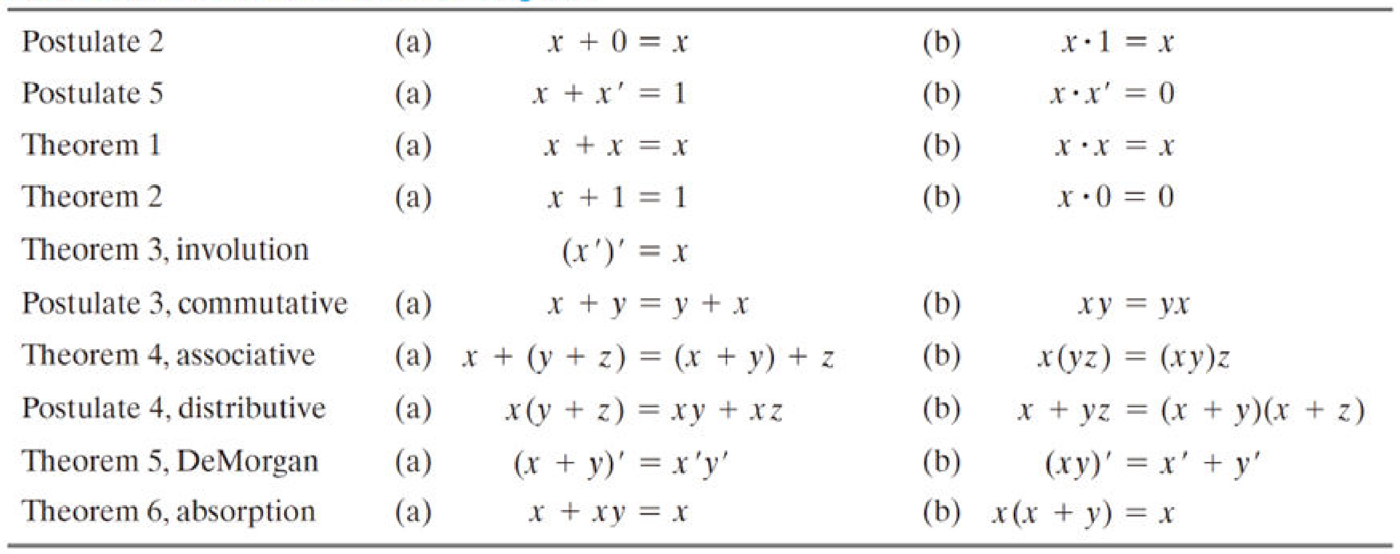
$y \cdot z$	$x+(y \cdot z)$	$x+y$	$x+z$	$(x+y) \cdot (x+z)$
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
1	1	1	1	1
0	1	1	1	1
0	1	1	1	1
0	1	1	1	1
1	1	1	1	1

- Complement
  - $x+x'=1$: $0+0'=0+1=1$; $1+1'=1+0=1$
  - $x \cdot x'=0$: $0 \cdot 0'=0 \cdot 1=0$; $1 \cdot 1'=1 \cdot 0=0$
- Has two distinct elements 1 and 0, with $0 \neq 1$
- We have just established a two-valued Boolean algebra:
  - a set of two elements
  - + : OR operation; · : AND operation
  - a complement operator: NOT operation
  - Binary logic is a two-valued Boolean algebra
  - also called “switching algebra” by engineers

Basic Theorems and Properties of Boolean Algebra

- Duality
  - the binary operators are interchanged; AND $\Leftrightarrow$ OR
  - the identity elements are interchanged; 1 $\Leftrightarrow$ 0

    Table. 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): $x+x = x$

$x+x = (x+x) \bullet 1$	by postulate: 2(b)
$= (x+x)(x+x')$	5(a)
$= x+xx'$	4(b)
$= x+0$	5(b)
$= x$	2(a)

- Theorem 1(b): $x \bullet x = x$

$x \bullet x = x\,x + 0$	by postulate: 2(a)
$= xx + xx'$	5(b)
$= x\,(x + x')$	4(a)
$= x \bullet 1$	5(a)
$= x$	2(b)

- Theorem 1(b) is the dual of theorem 1(a)

- Theorem 2(a): $x + 1 = 1$

$x + 1 = 1 \bullet (x + 1)$	by postulate: 2(b)
$= (x + x')(x + 1)$	5(a)
$= x + x' \bullet 1$	4(b)
$= x + x'$	2(b)
$= 1$	5(a)

- Theorem 2(b): $x \bullet 0 = 0$ by duality
- Theorem 3: $(x')' = x$
  - Postulate 5 defines the complement of x, $x + x' = 1$ and $x \bullet x' = 0$
  - The complement of x' is x is also $(x')'$

- Theorem 6(a): $x + xy = x$

$x + xy = x \bullet 1 + xy$	by postulate: 2(b)
$= x\,(1 + y)$	4(a)
$= x \bullet 1$	2(a)
$= x$	2(b)

- Theorem 6(b): $x\,(x + y) = x$ by duality
- By means of truth table

x	y	xy	$x + xy$
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

DeMorgan's Theorems

– $(x+y)' = x' y'$

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

– $(x\, y)' = x' + y'$

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

Operator Precedence

- The operator precedence for evaluating Boolean expressions is
  1. parentheses
  2. NOT
  3. AND
  4. OR
- Examples
  - $x\ y' + z$
  - $(x\ y + z)'$

Boolean Functions

- A Boolean function is an algebraic expression consists of
  - binary variables
  - binary operators OR and AND
  - unary operator NOT
  - parentheses
- 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.
- Examples
  - $F_1 = x + y z'$ ➔ $F_1 = 1$ if $x = 1$ <u>or</u> if $y = 0$ <u>and</u> $z = 1$, others $F_1 = 0$.
  - $F_2 = x' y' z + x' y z + x y'$ ➔

    $F_2 = 1$ if $(x = 0, y = 0, z = 1)$ or $(x = 0, y = 1, z = 1)$ or $(x = 1, y = 0)$,

    others $F_2 = 0$.

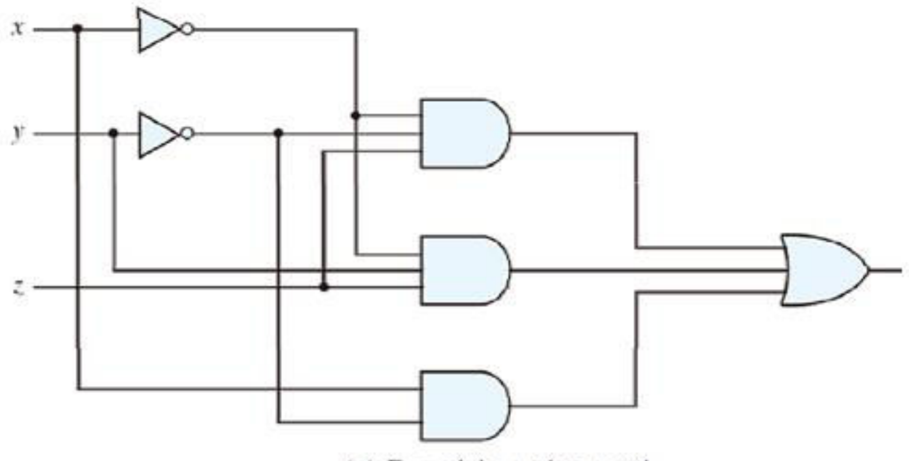
Truth Table

- Boolean function can be represented in a truth table.
- Truth table has 2^n rows where n is the number of variables in the function.
- The binary combinations for the truth table are obtained from the binary numbers by counting from 0 through $2^n - 1$.

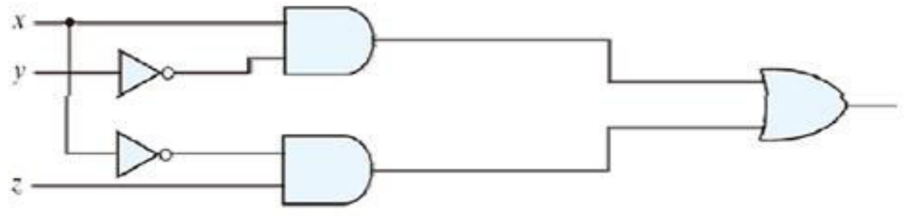
Equivalent Logics

- Boolean function can be represented in truth table only in one way.
- In algebraic form, it can be expressed in a variety of ways, all of which have equivalent logic.
- Using Boolean algebra, it is possible to obtain a simpler expression for the same function with less number of gates and inputs to the gate.
- Designers work on reducing the complexity and number of gates to significantly reduce the circuit cost.

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



(a) $F_2 = x'y'z + x'yz + xy'$



(b) $F_2 = xy' + x'z$

FIGURE 1
Implementation of Boolean function F_2 with gates

Algebraic Manipulation

- To minimize Boolean expressions
  - literal: a complemented or un-complemented variable (an input to a gate)
  - term: an implementation with a gate
  - The minimization of the number of literals and the number of terms => a circuit with less equipment

$$\begin{aligned} F_2 &= x'y'z + x'yz + xy' && \Rightarrow \text{3 terms, 8 literals} \\ &= x'z(y' + y) + xy' && \\ &= x'z + xy' && \Rightarrow \text{2 terms, 4 literals} \end{aligned}$$

- Functions of up to five variables can be simplified by the map method described in the next chapter.
- For complex Boolean functions and many different outputs, designers of digital circuits use computer minimization programs that are capable of producing optimal circuits with millions of logic gates.

Minimization of Boolean Function

EXAMPLE

Simplify the following Boolean functions to a minimum number of literals.

1. $x(x' + y) = xx' + xy = 0 + xy = xy.$
2. $x + x'y = (x + x')(x + y) = 1(x + y) = x + y.$
3. $(x + y)(x + y') = x + xy + xy' + yy' = x(1 + y + y') = x.$
4. $$\begin{aligned} xy + x'z + yz &= xy + x'z + yz(x + x') \\ &= xy + x'z + xyz + x'yz \\ &= xy(1 + z) + x'z(1 + y) \\ &= xy + x'z. \end{aligned}$$
5. $(x + y)(x' + z)(y + z) = (x + y)(x' + z)$, by duality from function 4.

Complement of a Function

- F' is obtained from an interchange of 0's for 1's and 1's for 0's in the value of F
- The complement of a function may be derived using DeMorgan's theorem.
- Three-variable DeMorgan's theorem:

$(A + B + C)' = (A + X)'$	let $B + C = X$
$= A'X'$	by DeMorgan's
$= A'(B + C)'$	$X = B + C$
$= A'(B'C')$	by DeMorgan's
$= A'B'C'$	associative

- Generalized form
  - $(A + B + C + \ldots + F)' = A'B'C' \ldots F'$
  - $(ABC \ldots F)' = A' + B' + C' + \ldots + F'$

EXAMPLE 2

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:

$$
\begin{aligned}
F_1' &= (x'yz' + x'y'z)' = (x'yz')'(x'y'z)' = (x + y' + z)(x + y + z') \\
F_2' &= [x(y'z' + yz)]' = x' + (y'z' + yz)' = x' + (y'z')'(yz)' \\
&= x' + (y + z)(y' + z') \\
&= x' + yz' + y'z
\end{aligned}
$$

EXAMPLE 2

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

Minterms and Maxterms

- A minterm (standard product): an AND term consists of all literals in their normal form or in their complement form
- For example, two binary variables *x* and *y*, has 4 minterms
  - *xy, xy', x'y, x'y'*
- *n* variables can be combined to form 2^n minterms (m_j, $j = 0 \sim 2^n\text{-}1$)
- A maxterm (standard sum): an OR term; 2^n maxterms (M_j, $j = 0 \sim 2^n\text{-}1$)
- Each maxterm is the complement of its corresponding minterm, and vice versa.

			Minterms		Maxterms	
x	**y**	**z**	**Term**	**Designation**	**Term**	**Designation**
0	0	0	$x'y'z'$	m_0	$x + y + z$	M_0
0	0	1	$x'y'z$	m_1	$x + y + z'$	M_1
0	1	0	$x'yz'$	m_2	$x + y' + z$	M_2
0	1	1	$x'yz$	m_3	$x + y' + z'$	M_3
1	0	0	$xy'z'$	m_4	$x' + y + z$	M_4
1	0	1	$xy'z$	m_5	$x' + y + z'$	M_5
1	1	0	xyz'	m_6	$x' + y' + z$	M_6
1	1	1	xyz	m_7	$x' + y' + z'$	M_7

Canonical Form: Sum of Minterms

- An Boolean function can be expressed by
  - a truth table
  - sum of minterms ➔ $f = \Sigma\, m_j$
  - product of maxterms ➔ $f = \Pi\, M_j$
  - $f_1 = x'y'z + xy'z' + xyz = m_1 + m_4 + m_7$
  - $f_2 = x'yz + xy'z + xyz' + xyz = m_3 + m_5 + m_6 + m_7$

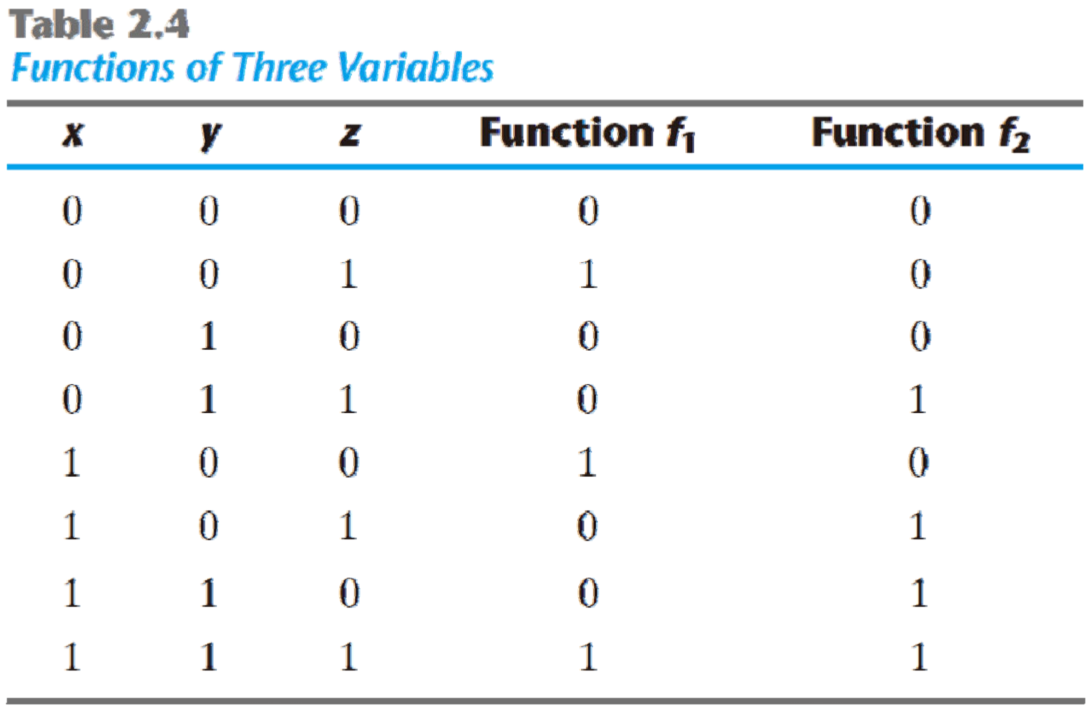


Table 2.4
Functions of Three Variables

x	y	z	Function f_1	Function f_2
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Canonical Form: Product of Maxterms

- The complement of a Boolean function
  - the minterms that produce a 0
  - $f_1' = m_0 + m_2 + m_3 + m_5 + m_6 = x'y'z' + x'yz' + x'yz + xy'z + xyz'$
  - $f_1 = (f_1')' = (x + y + z)(x + y' + z)(x + y' + z')(x' + y + z')(x' + y' + z)$
  - $= M_0 M_2 M_3 M_5 M_6$
  - $f_2 = (x + y + z)(x + y + z')(x + y' + z)(x' + y + z)$
    $= M_0 M_1 M_2 M_4$
- Canonical form: any Boolean function expressed as a sum of minterms or a product of maxterms

Minterm Expansion

- **EXAMPLE 4:** Express the Boolean function *F=A+B’C* as a sum of minterms.
  - F = A + B'C = A (B + B') + B'C = AB + AB' + B'C
  - = AB(C + C') + AB'(C + C') + (A + A')B'C
  - = ABC + ABC‘ + AB'C + AB'C‘ + A'B'C
  - = A'B'C + AB'C' + AB'C + ABC‘ + ABC
  - = $m_1 + m_4 + m_5 + m_6 + m_7$
  - F(A,B,C) = Σ (1, 4, 5, 6, 7)
  - or, built the truth table first

Table 3.5
Truth Table for F = A + B'C

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

Maxterm Expansion

EXAMPLE 5: Express the Boolean function $F = xy + x'z$ as a product of maxterms.

- $F = xy + x'z = (xy + x')(xy + z) = (x + x')(y + x')(x + z)(y + z)$
- $= (x' + y)(x + z)(y + z)$

- $x' + y = x' + y + zz' = (x' + y + z)(x' + y + z')$
- $x + z = x + z + yy' = (x + y + z)(x + y' + z)$
- $y + z = y + z + xx' = (x + y + z)(x' + y + z)$

- $F = (x + y + z)(x + y' + z)(x' + y + z)(x' + y + z') = M_0 M_2 M_4 M_5$
- $F(x,y,z) = \Pi\ (0,2,4,5)$

- check this result with truth table

Canonical Form Conversion

- Conversion between Canonical Forms
  - $F(A,B,C) = \Sigma(1,4,5,6,7)$ ➔ $F'(A,B,C) = \Sigma(0,2,3) = m_0 + m_1 + m_2$
  - By DeMorgan's theorem

    $F = (m_0 + m_1 + m_2)' = m'_0 \bullet m'_2 \bullet m'_3$

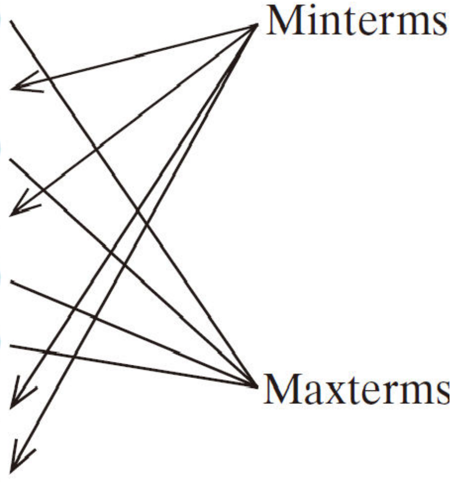
    $= M_0 M_2 M_3 = \Pi(0, 2, 3)$
  - $m_j' = M_j$
- sum of minterms $\Leftrightarrow$ product of maxterms
  - interchange the symbols Σ and Π and list those numbers missing from the original form
    - Σ of 1's $\Leftrightarrow$ Π of 0's

Conversion Example

- $F = xy + x'z$
- $F(x, y, z) = \Sigma(1, 3, 6, 7)$
- $F(x, y, z) = \Pi\,(0, 2, 4, 5)$

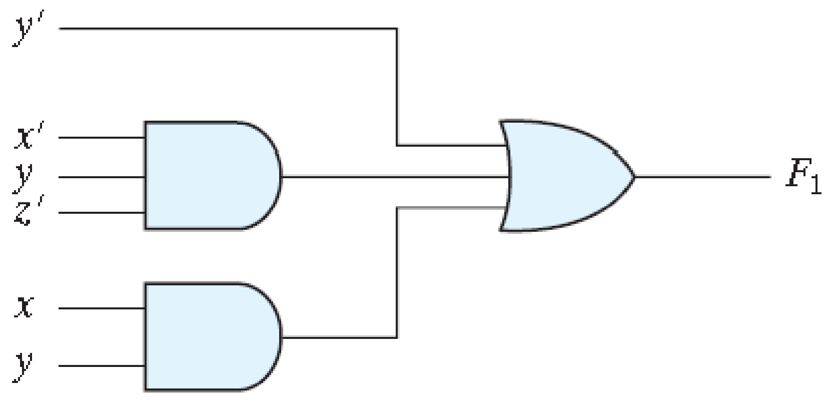
Table.4.6
Truth Table for $F = xy + x'z$

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

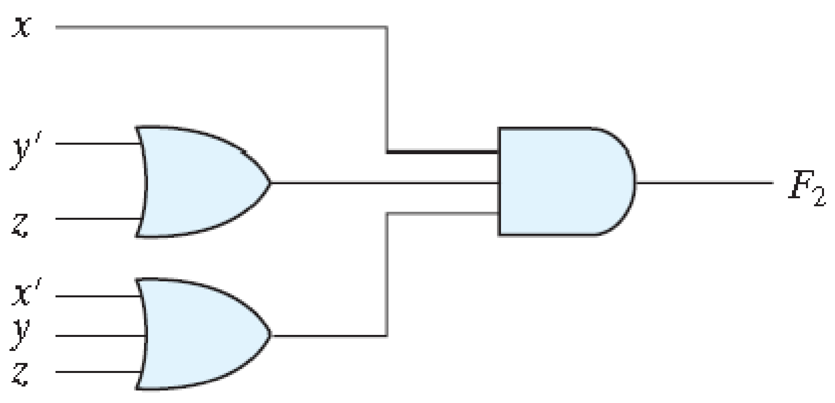


Standard Forms

- Canonical forms are baseline expression and seldom used, they are not minimum
- Two standard forms are used usually
  - sum of products $F_1 = y' + xy + x'yz'$
  - product of sums $F_2 = x(y' + z)(x' + y + z')$



(a) Sum of Products

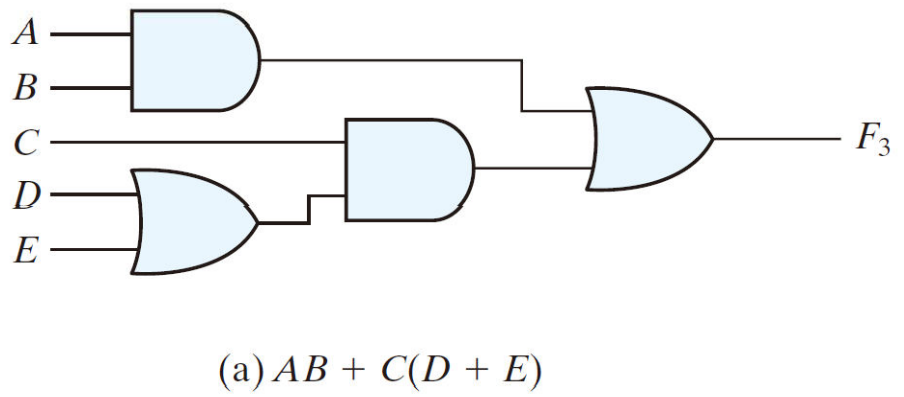


(b) Product of Sums

- This circuit configuration is referred to as a *two-level* implementation.
- In general, a two-level implementation is preferred because it produces the least amount of delay through the gates when the signal propagates from the inputs to the output. However, the number of inputs to a given gate might not be practical.

Nonstandard Forms

- $F_3 = AB + C(D + E)$
  $= AB + C(D + E) = AB + CD + CE$



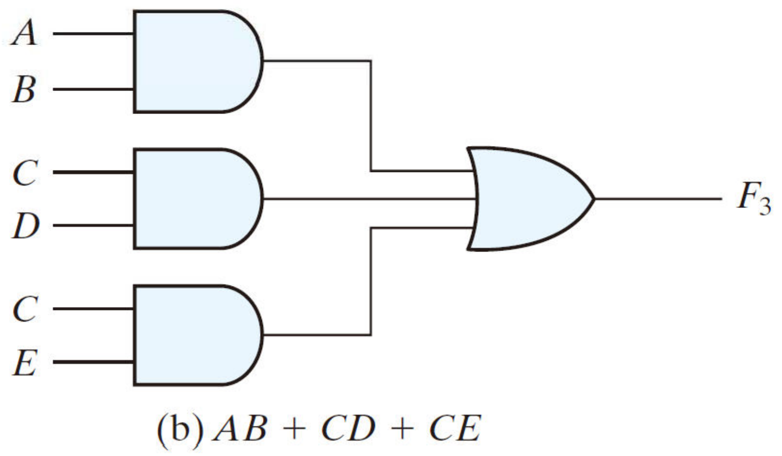


FIGURE 2.4
Three- and two-level implementation

- Which kind of gate will have the least delay (high switching speed)?
- The delay through a gate is largely dependent on the circuit design and technology, as well as manufacturing process used.

Other Logic Operations

- 2^n rows in the truth table of n binary variables
- 2^{2^n} functions for n binary variables (each row may either be 0 or 1)
- 16 (2^{2^2})functions of two binary variables

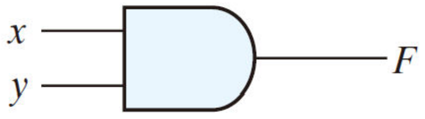
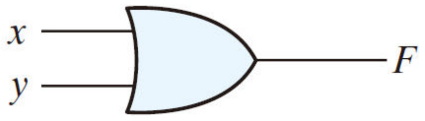
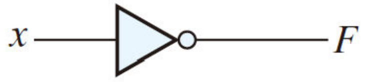
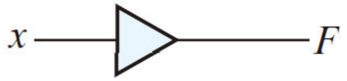
Table 3
Truth Tables for the 16 Functions of Two Binary Variables

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

Table ~~2~~5.8
Boolean Expressions for the 16 Functions of Two Variables

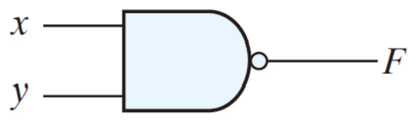
Boolean Functions	Operator Symbol	Name	Comments
$F_0 = 0$		Null	Binary constant 0
$F_1 = xy$	$x \cdot y$	AND	x and y
$F_2 = xy'$	x/y	Inhibition	x, but not y
$F_3 = x$		Transfer	x
$F_4 = x'y$	y/x	Inhibition	y, but not x
$F_5 = y$		Transfer	y
$F_6 = xy' + x'y$	$x \oplus y$	Exclusive-OR	x or y, but not both
$F_7 = x + y$	$x + y$	OR	x or y
$F_8 = (x + y)'$	$x \downarrow y$	NOR	Not-OR
$F_9 = xy + x'y'$	$(x \oplus y)'$	Equivalence	x equals y
$F_{10} = y'$	y'	Complement	Not y
$F_{11} = x + y'$	$x \subset y$	Implication	If y, then x
$F_{12} = x'$	x'	Complement	Not x
$F_{13} = x' + y$	$x \supset y$	Implication	If x, then y
$F_{14} = (xy)'$	$x \uparrow y$	NAND	Not-AND
$F_{15} = 1$		Identity	Binary constant 1

Digital Logic Gates of Two Inputs

Name	Graphic symbol	Algebraic function	Truth table
AND		$F = x \cdot y$	$\begin{array}{cc\|c} x & y & F \\ \hline 0 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{array}$
OR		$F = x + y$	$\begin{array}{cc\|c} x & y & F \\ \hline 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{array}$
Inverter		$F = x'$	$\begin{array}{c\|c} x & F \\ \hline 0 & 1 \\ 1 & 0 \end{array}$
Buffer		$F = x$	$\begin{array}{c\|c} x & F \\ \hline 0 & 0 \\ 1 & 1 \end{array}$

Digital Logic Gates of Two Inputs

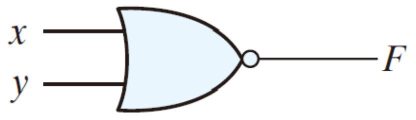
NAND



$F = (xy)'$

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

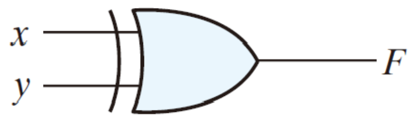
NOR



$F = (x + y)'$

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

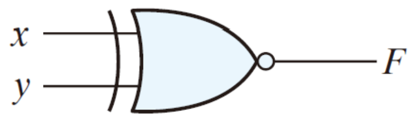
Exclusive-OR
(XOR)



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

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

Exclusive-NOR
or
equivalence

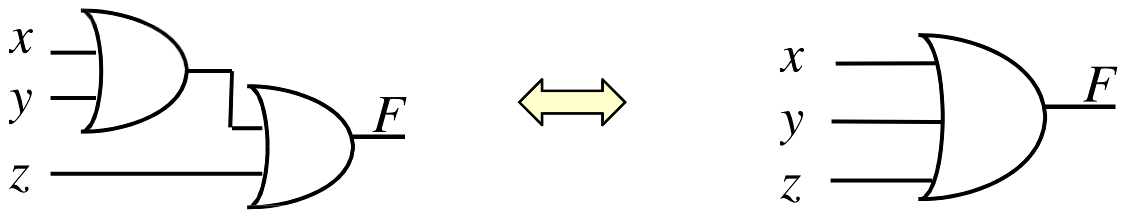


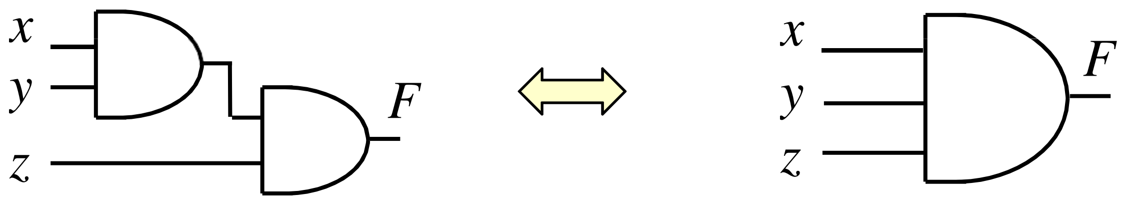
$F = xy + x'y'$
$= (x \oplus y)'$

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

Extension to Multiple Inputs

- A gate can be extended to multiple inputs
  - if its binary operation is commutative and associative
- AND and OR are commutative and associative
  - commutative: $x + y = y + x$, $xy = yx$
  - associative: $(x + y) + z = x + (y + z) = x + y + z$, $(x\,y)z = x(y\,z) = x\,y\,z$



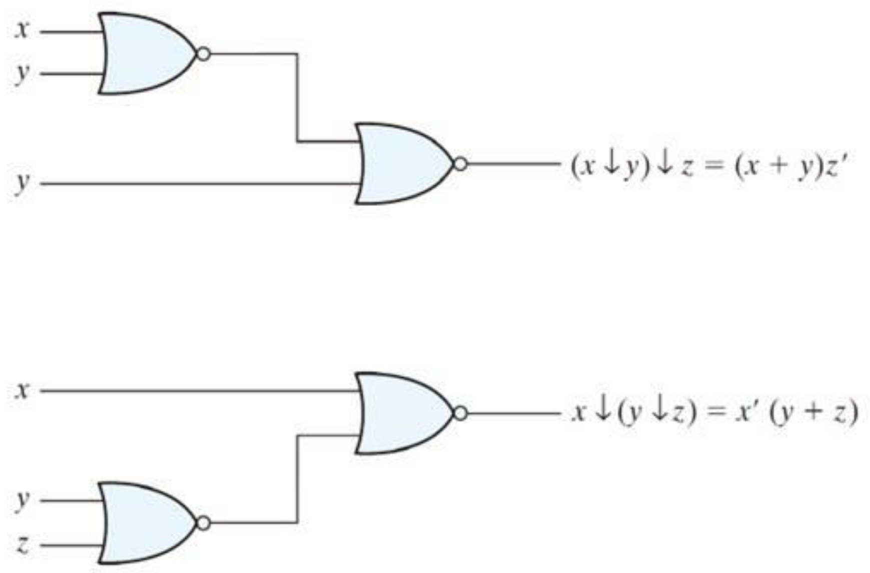


Multiple-input NOR/NAND

- NAND and NOR are commutative but not associative => they are not extendable

  $(x \downarrow y) \downarrow z = [(x + y)' + z]' = (x + y)\, z' = xz' + yz'$

  $x \downarrow (y \downarrow z) = [x + (y + z)']' = x'(y + z) = x'y + x'z$



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

FIGURE 4
Demonstrating the nonassociativity of the NOR operator: $(x \downarrow y) \downarrow z \neq x \downarrow (y \downarrow z)$

Multiple-input NOR/NAND

- Multiple-input NOR = a complement of OR gate $(x \downarrow y \downarrow z) = (x + y + z)'$
- Multiple-input NAND = a complement of AND $(x \uparrow y \uparrow z) = (xy\ z)'$
- The cascaded NAND operations = sum of products
- The cascaded NOR operations = product of sums

DeMorgan's theorems are useful here.

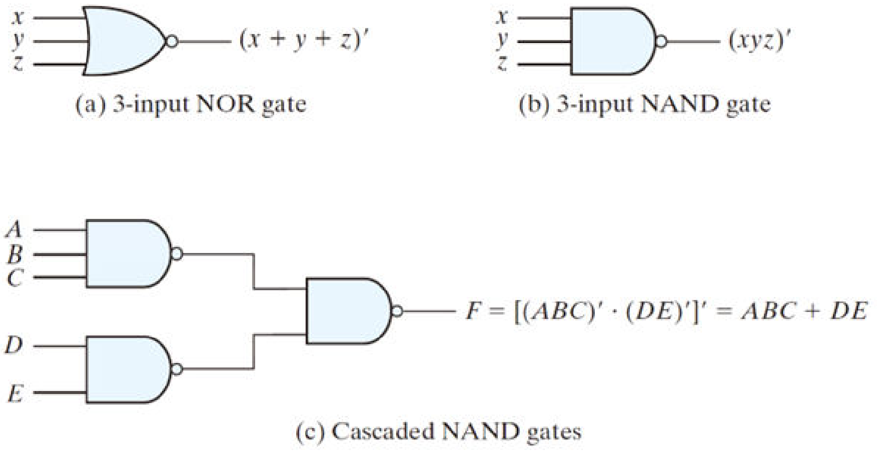
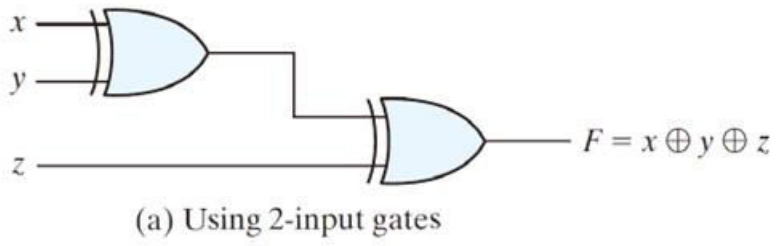


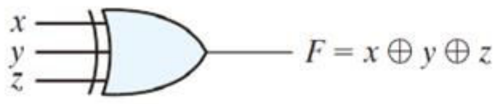
FIGURE 5
Multiple-input and cascaded NOR and NAND gates

Multiple-input XOR/XNOR

- The XOR and XNOR gates are commutative and associative
- Multiple-input XOR gates are uncommon (this is not true anymore!)
- XOR(XNOR) is an odd(even) function: it is equal to 1 if the inputs variables have an odd(even) number of 1's



(a) Using 2-input gates



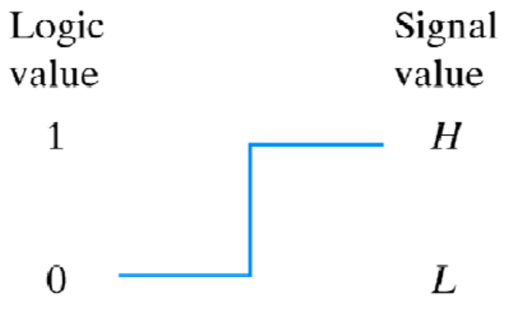
(b) 3-input gate

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

(c) Truth table

FIGURE 6
Three-input exclusive-OR gate

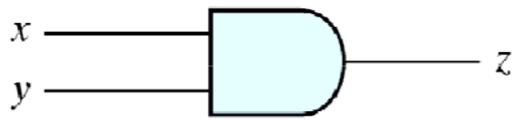
- Two signal values (High/Low) <=> two logic values (1/0)
  - positive logic: H = 1; L = 0
  - negative logic: H = 0; L = 1
- Positive logic is commonly used.



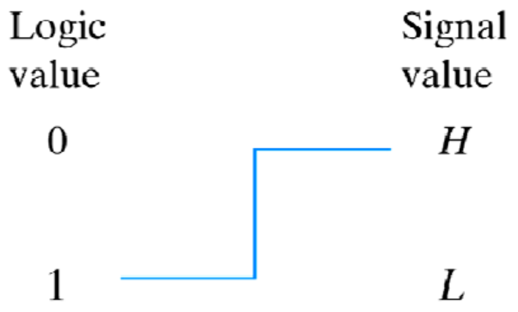
(a) Positive logic

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

(c) Truth table for positive logic



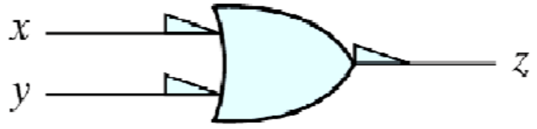
(d) Positive logic AND gate



(b) Negative logic

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

(e) Truth table for negative logic



(f) Negative logic OR gate

Unit II Gate Level Minimization

The Map Method-Four Variable K-Map-Five Variable K-Map-Product of Sums Simplification-Don't Care Condition-NAND and NOR Implementation, EX-OR Function-Tabular Minimization method.

Introduction:

Gate-level minimization is the design task of finding an optimal gate-level implementation of the Boolean functions describing a digital circuit. every boolean function can be expressed as a sum of minterms or a product of maxterms. Since the number of literals in such an expression is usually high, and the complexity of the digital logic gates that implement a Boolean function is directly related to the complexity of the algebraic expression from which the function is implemented, it is preferable to have the most simplified form of the algebraic expression. The process of simplifying the algebraic expression of a boolean function is called minimization.

The Map Method:

The map method provides a simple, straightforward procedure for minimizing Boolean functions. This method may be considered as a pictorial form of a truth table. It 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 square whose minterms are included in the function. The simplified expressions produced by the map are always in one of the two standard forms: Sum of Products or Product of sums. It is sometimes possible to find two or more expressions that satisfy the minimization criteria. The simplest algebraic expression is an algebraic expression with a minimum number of terms and with the smallest possible number of literals in each term. This expression produces a circuit diagram with a minimum number of gates and the minimum number of inputs to each gate.

Two-Variable Map:

There are four minterms for two variables; hence, the map consists of four squares one for each minterm. The 0 and 1 marked in each row and column designate the values of variables. Variable 'x' appears primed in row 0 and unprimed in row 1. Similarly, 'y' appears primed in column 0 and unprimed in column 1. Thus, a 2 * 2 square can be used for representing any two-variable function.

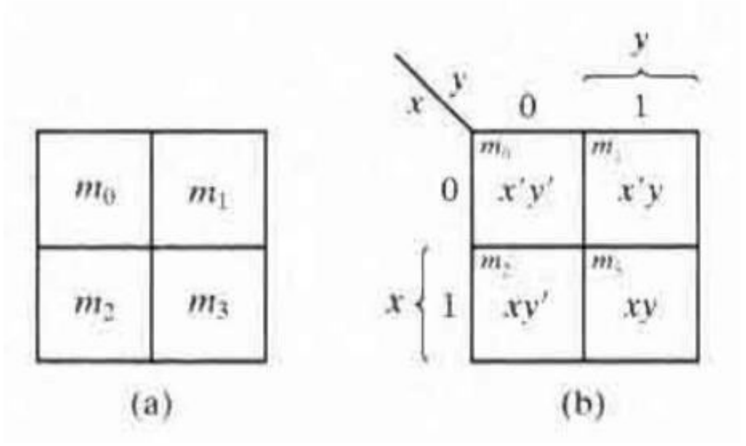


Figure 1. Two-Variable K-Map

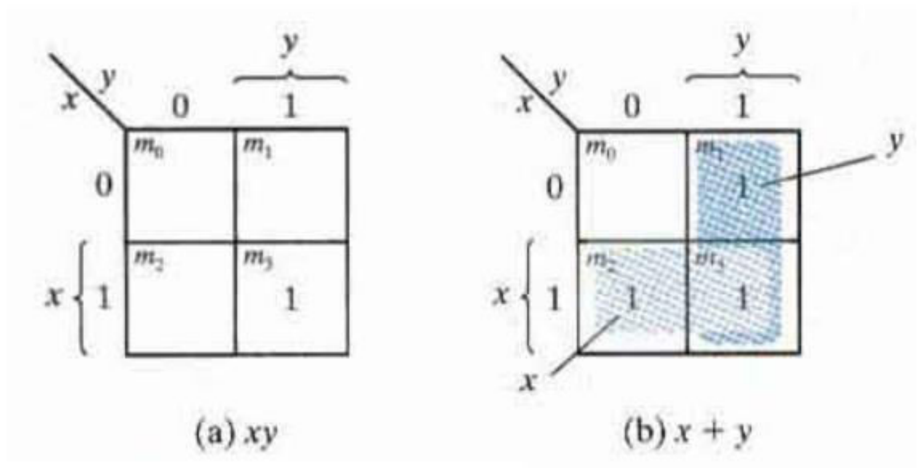


Figure 2. Two Variable K-Map with values mapped

Three-Variable K-Map:

A three-variable K-map is shown in Fig. 3. There are eight minterms for three binary variables; therefore, the map consists of eight squares. Note that the minterms are arranged, not in a binary sequence, but in a sequence similar to the Gray code. The characteristic of this sequence is that only one-bit changes in value from one adjacent column to the next. The map drawn in part (b) is marked with numbers in each row and each column to show the relationship between the squares and the three variables. For example, the square assigned to m_5 corresponds to row 1 and column 01. When these two numbers are concatenated, they give the binary number 101, whose decimal equivalent is 5. Each cell of the map corresponds to a unique minterm, so another way of looking at square $m_5 = xy'z$ is to consider it to be in the row marked x and the column belonging to y'z (column 01). Note that there are four squares in which each variable is equal to 1 and four in which each is equal to 0. The variable appears unprimed in the former four squares and primed in the latter. For convenience, we write the variable with its letter symbol under the four squares in which it is unprimed.

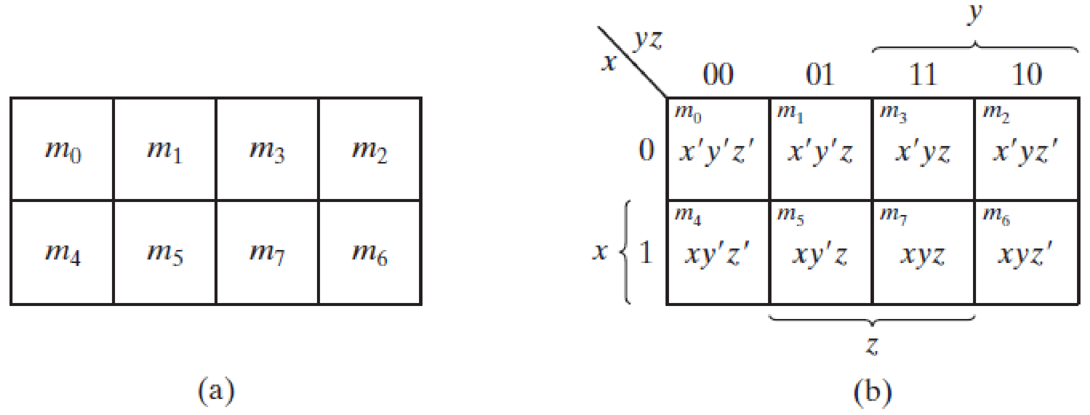


Figure 3. Three Variable K-Map

To understand the usefulness of the map in simplifying Boolean functions, we must recognize the basic property possessed by adjacent squares: Any two adjacent squares in the map differ by only one variable, which is primed in one square and unprimed in the other.

For example, m_5 and m_7 lie in two adjacent squares. Variable y is primed in m_5 and unprimed in m_7, whereas the other two variables are the same in both squares. From the postulates of Boolean algebra, it follows that the sum of two minterms in adjacent squares can be simplified to a single product term consisting of only two literals. To clarify this concept, consider the sum of two adjacent squares such as m_5 and m_7:

m_5 and $m_7 = xy'z + xyz = xz(y'+ y) = xz$.

Here, the two squares differ by the variable y, which can be removed when the sum of the two minterms is formed. Thus, any two minterms in adjacent squares (vertically or horizontally, but not diagonally, adjacent) that are ORed together will cause a removal of the dissimilar variable.

Example 1. Simplify the Boolean function $F(X,Y,Z) = \sum(2,3,4,5)$

1. First, a 1 is marked in each minterm square that represents the function. This is shown in Fig. 4, in which the squares for minterms 010, 011, 100, and 101 are marked with 1's.
2. The next step is to find possible adjacent squares. These are indicated in the map by two shaded rectangles, each enclosing two 1's.
3. The upper right rectangle represents the area enclosed by x'y. Similarly, the lower left rectangle represents the product term xy'.

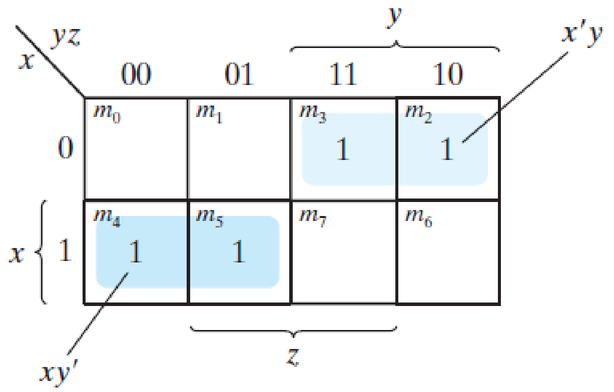


Figure 4. Map for Example 1. F (x, y, z) = (2, 3, 4, 5) = x'y + xy'

4. The logical sum of these two product terms gives the simplified expression $F(X,Y\ Z) = X'Y + XY'$

FOUR-VARIABLE K-MAP:

- The map for Boolean functions of four binary variables has 16 squares with each corresponding to a minterm. 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 m_{13}.**
- Adjacent squares are defined to be squares next to each other. The squares on the top column touch the squares in the bottom column. In addition, squares in the leftmost column touch squares in the rightmost column.
- 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 1.
- No other combination of squares can simplify the function.

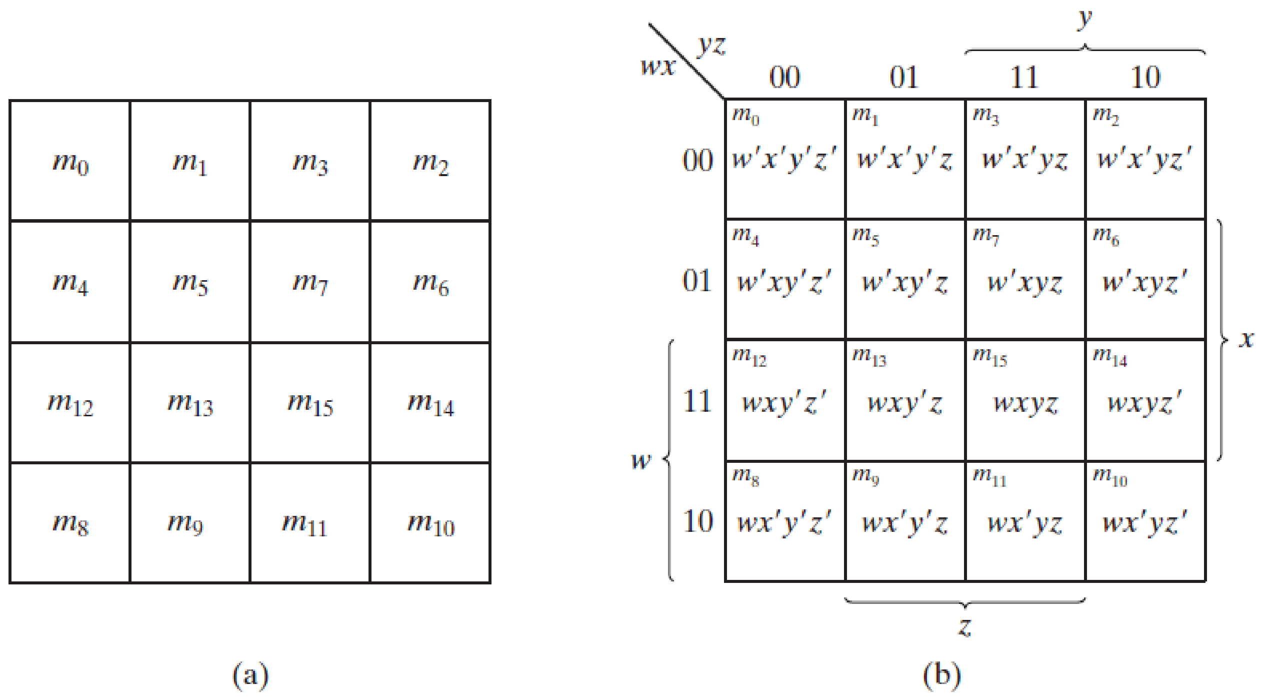

m_0 m_1 m_3 m_2
m_4 m_5 m_7 m_6
m_{12} m_{13} m_{15} m_{14}
m_8 m_9 m_{11} m_{10}
(a)
wx yz 00 01 11 10
y
00 $w'x'y'z'$ $w'x'y'z$ $w'x'yz$ $w'x'yz'$
01 $w'xy'z'$ $w'xy'z$ $w'xyz$ $w'xyz'$
11 $wxy'z'$ $wxy'z$ $wxyz$ $wxyz'$
10 $wx'y'z'$ $wx'y'z$ $wx'yz$ $wx'yz'$
x
w
z
(b)

Figure 5. Four-Variable K-Map

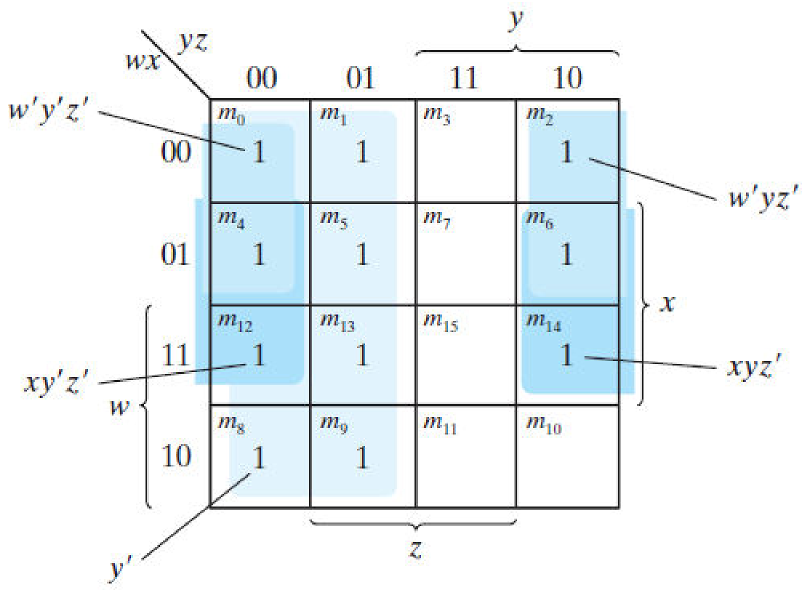
Example 2. Simplify the Boolean function $F(w, x, y, z) = \sum(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$

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.6.

- 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'*.

- Note that it is permissible to use the same square more than once. We are now left with a square marked by 1 in the third row and fourth column (square 1110). Instead of taking this square alone (which will give a term with four literals), we combine it with squares already used to form an area of four adjacent squares. These squares make up the two middle rows and the two end columns, giving the term *xz'*.



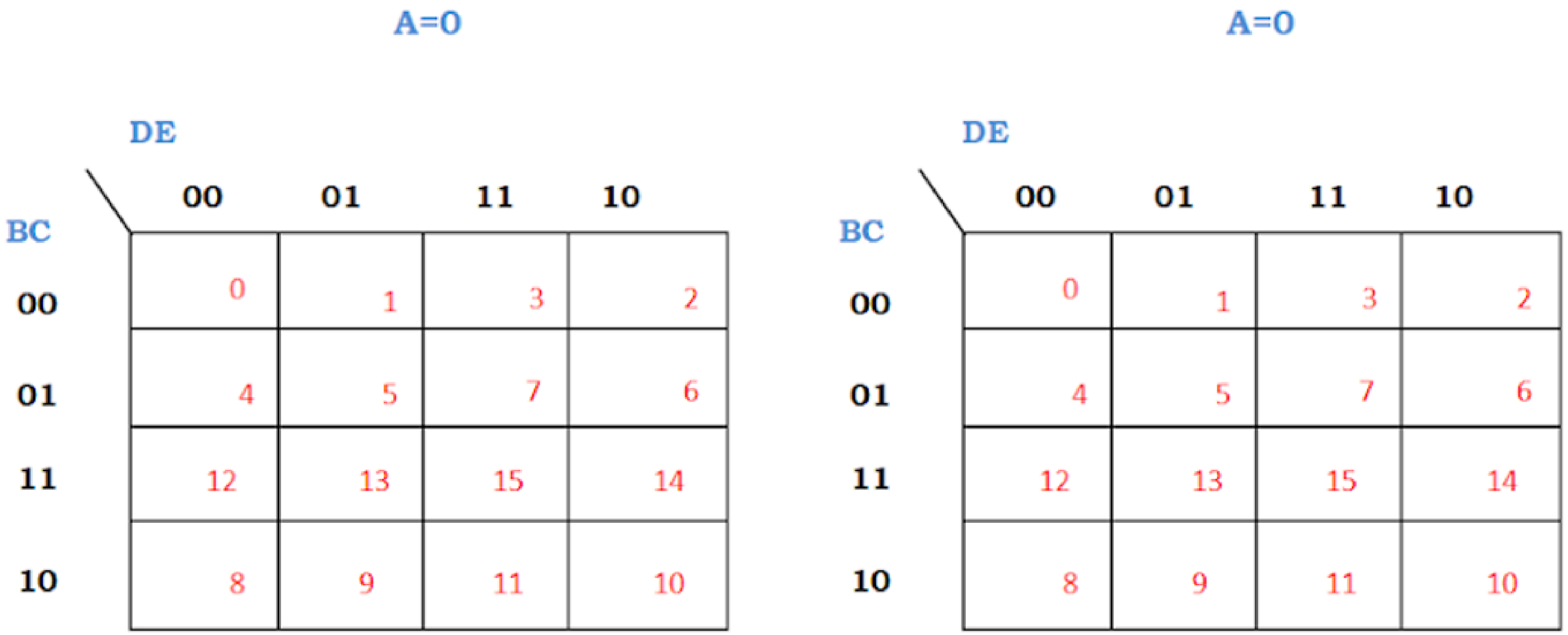
Note: $w'y'z' + w'yz' = w'z'$
$xy'z' + xyz' = xz'$

Figure 6. Map for Example 2

- The simplified function is $F = y' + w'z' + xz'$

Five-Variable Map

- Maps for more than four variables are not as simple to use as maps for four or fewer variables. A five-variable map needs 32 squares and a six-variable map needs 64 squares.
- When the number of variables becomes large, the number of squares becomes excessive and the geometry for combining adjacent squares becomes more involved.
- To solve Boolean expression with 5 variables, 2^5 cells are required. So, we need 32 cells. It can be split into two K map and can be solved. The numbering of the cells is given below.



A=0

BC \ DE	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

A=0

BC \ DE	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

Figure 7. Five-Variable K-Map

Example 3: Minimize the following 5 variable SOP function using K map:

F(A,B,C,D,E)= ∑m(0,5,6,8,9,10,11,16,20,24,25,26,27)

Follow the steps what we follow in Four-Variable K-Map here.

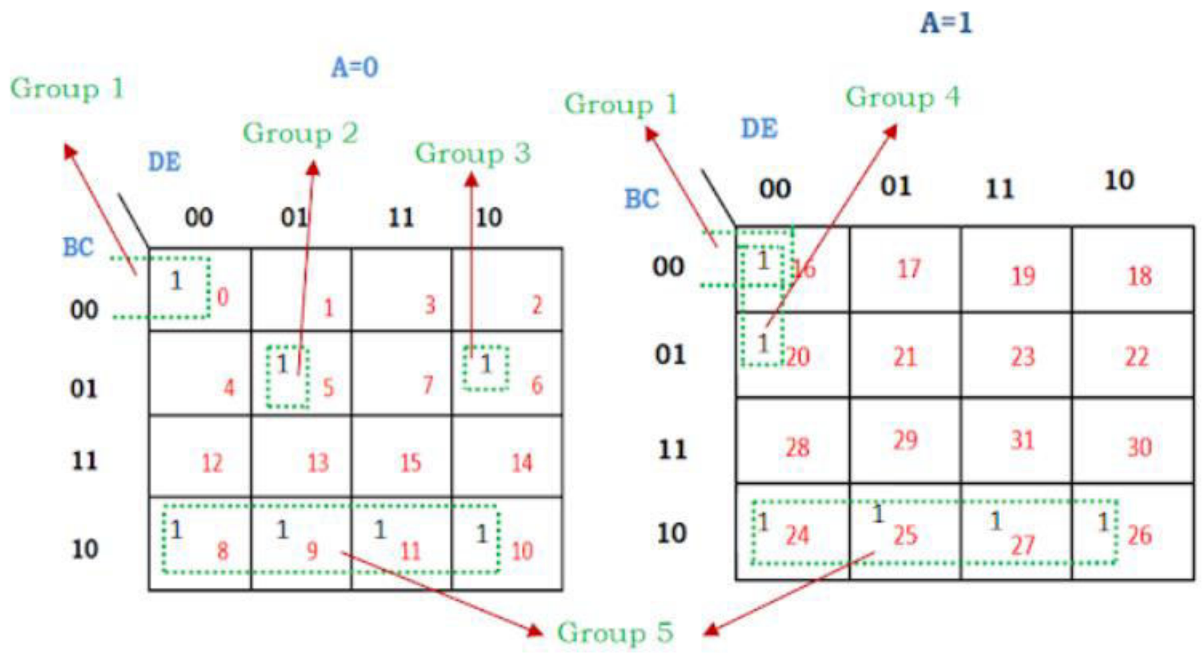


Figure 8. K map for 5 variable SOP function

- First fill the cells which are given in the POS function with ‘0’.
- Group the adjacent cells in group of 1, 2 and 4.
- There are five groups in this K map.

1. Group 1: Cells 0 and 16 are grouped. The common term is B+C+D+E.

2. Group 2: Cell number 5 is in group 2 and the Maxterm for this group is A+B+C'+D+E'.
3. Group 3: Cell number 6 is in group 3 and the Maxterm for this group is A+B+C'+D'+E.
4. Group 4: Cells 16 and 20 are grouped. The common term is A'+B+D+E.
5. Group 5: Cells 8, 9, 10, 11, 24, 25, 26 and 27 are grouped. The common term is B'+C.

So the minimized SOP expression is,

(A, B, C, D, E)= (B+C+D+E).(A+B+C'+D+E').(A+B+C'+D'+E).(A'+B+D+E).(B'+C)

PRODUCT-OF-SUMS SIMPLIFICATION:

- If we are given a function in Sum of Products form and we have to find the minimization in Product of Sum form we follow the procedure:
  - Mark all the minterms as 1 in the squares of the K-Map. The empty spaces are marked as 0's.
  - Then we use the procedure of combining adjacent squares to create larger squares or rectangles. However, we shall combine squares **containing 0's and not 1's.**
  - This is the only difference from the Sum of Products procedure.
  - Applying DeMorgan's theorem **(by taking the dual and complementing each literal),** we obtain the simplified function in product-of-sums form

Example 4. Simplify the: following Boolean function into (a) sum-of-products form and (b) product-of sums form: F(A, B, C, D) = ∑(0, 1, 2, 5, 8, 9, 10)

- The 1's marked in the figure represent all the minterms of the function. The squares marked with 0's represent the minterms not included in F and therefore denote the complement of F. Combining the squares with 1's gives the simplified function in sum-of-products form:

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

  If the squares marked with 0's are combined we obtain the simplified complemented function:

  F' = AB + CD + BD'

  Applying DeMorgan ' s theorem (by taking the dual and complementing each literal), we obtain the simplified function in product-of-sums form:

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

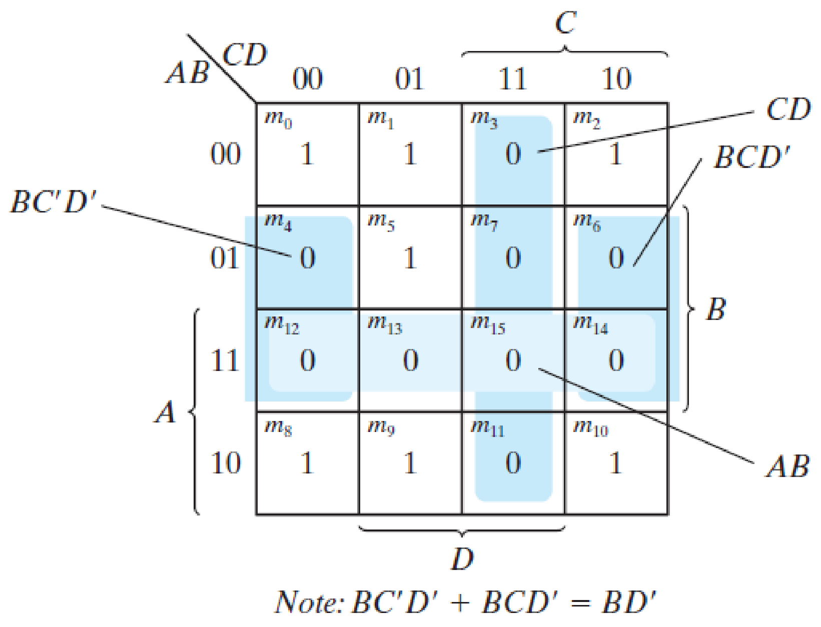
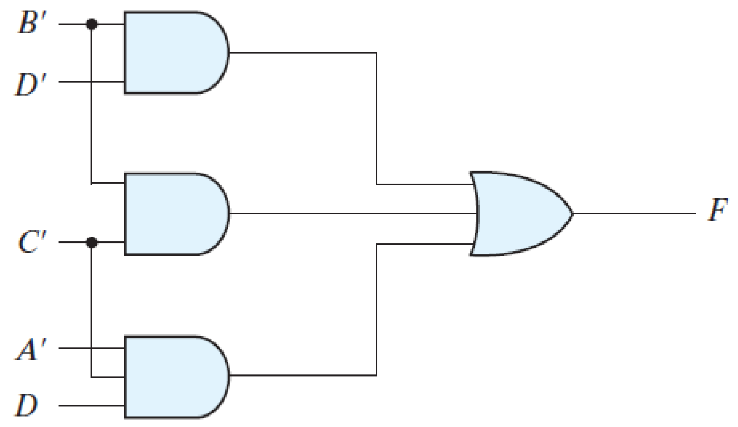
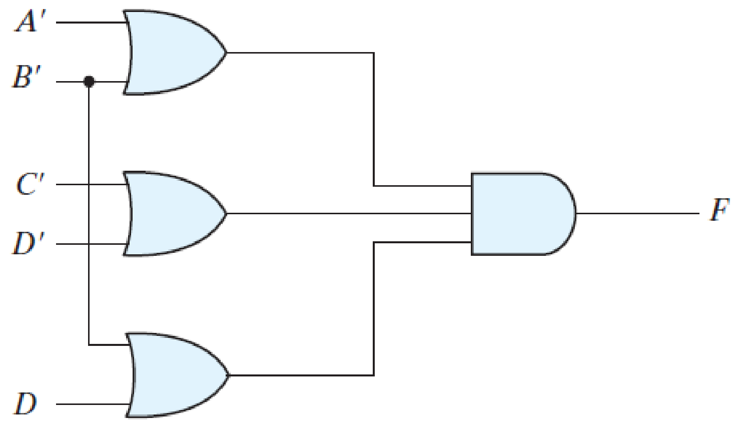


Figure 9. K-Map for SOP and POS



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



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

Figure 10. Gate Implementation of Example 4.

Table 1. Truth Table for Example 4.

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

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.
- In practice, in some applications the function is not specified for certain combinations of the variables.
- **Example:** To represent 10 decimal numbers (0-9) using binary we need 4 bits, but this leads to 6 combinations being unused i.e. from **1010 - 1111**
- 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 is marked with a **X**
- In choosing adjacent squares to simplify the function in a map. the don' t-care minterms 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.

Example 5. 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)

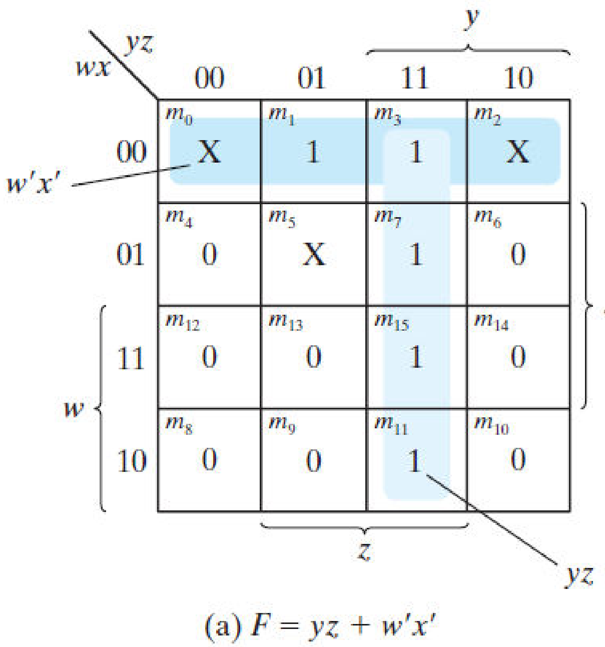
- The minterms of F are the variable combinations that make the function equal to 1. The minterms of d are the don't-care minterms which are marked as X.
- To get the simplified expression in sum-of-products form, we must include all five 1's in the map, but we may or may not include any of the X's, depending on the way the function is simplified.
-

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

  F = yz + w'x'

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

  F = yz + w'z



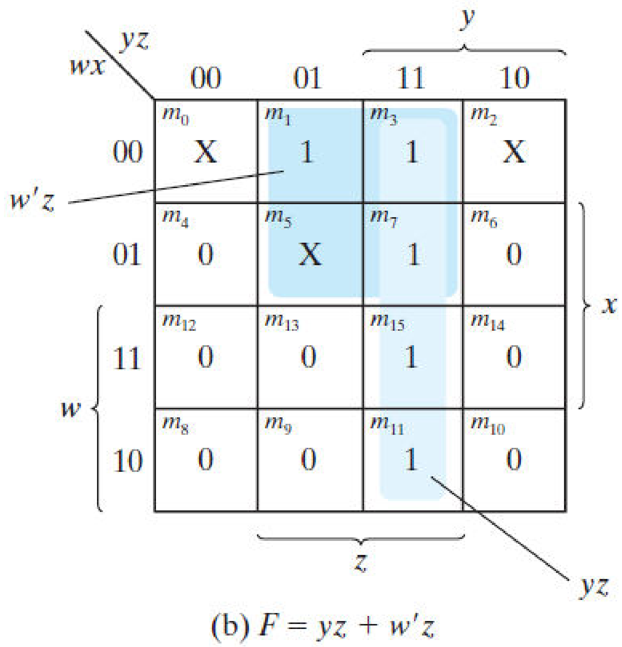


Figure 11. Example with don’t-care conditions

NAND AND NOR IMPLEMENTATION:

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

NAND Circuits:

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

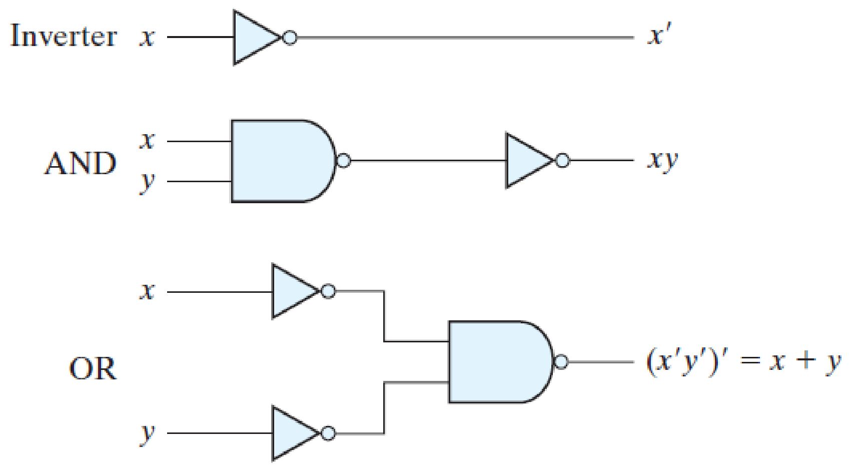


Figure 12. Logic operations with NAND gates

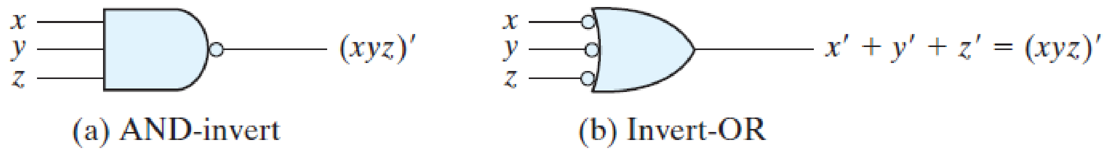


Figure 13. Two graphic symbols for a three-input NAND gate

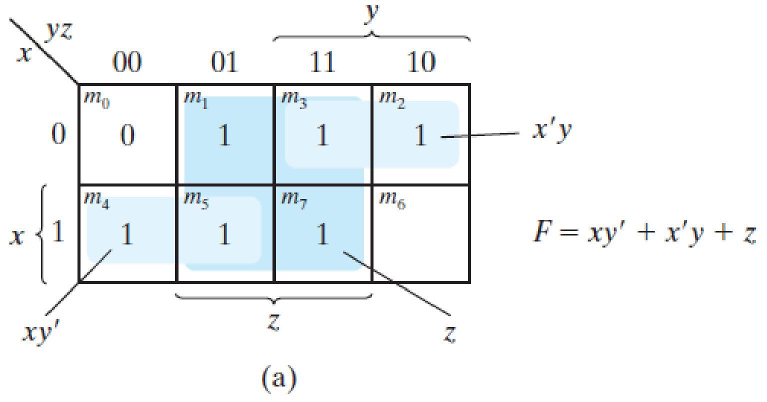
Example 6. Implement the following Boolean function with NAND gates:

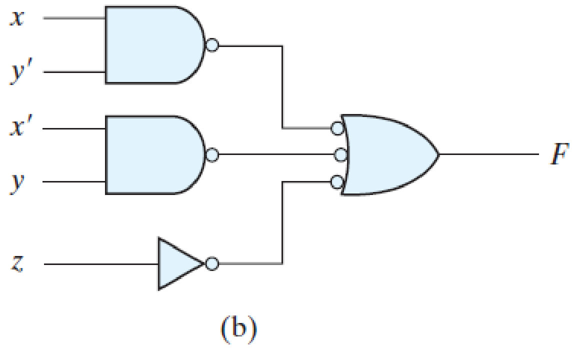
$F\ (x, y, z) = (1, 2, 3, 4, 5, 7)$

- The first step is to simplify the function into sum-of-products form. This is done by means of the map of Fig.14, from which the simplified function is obtained:

F = xy_ + x_y + z

- The two-level NAND implementation is shown in Fig. 14.(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. 14 (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'.





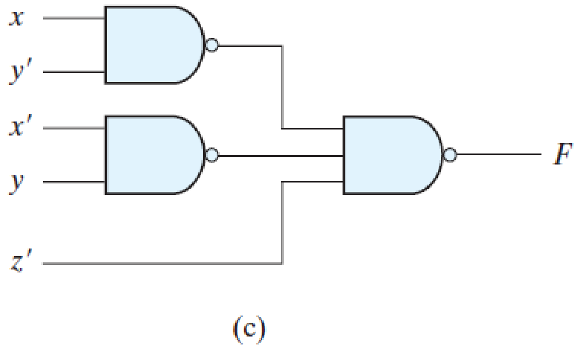


Figure 14. Solution to Example 6.

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. 15. The complement operation is obtained from a oneinput 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.

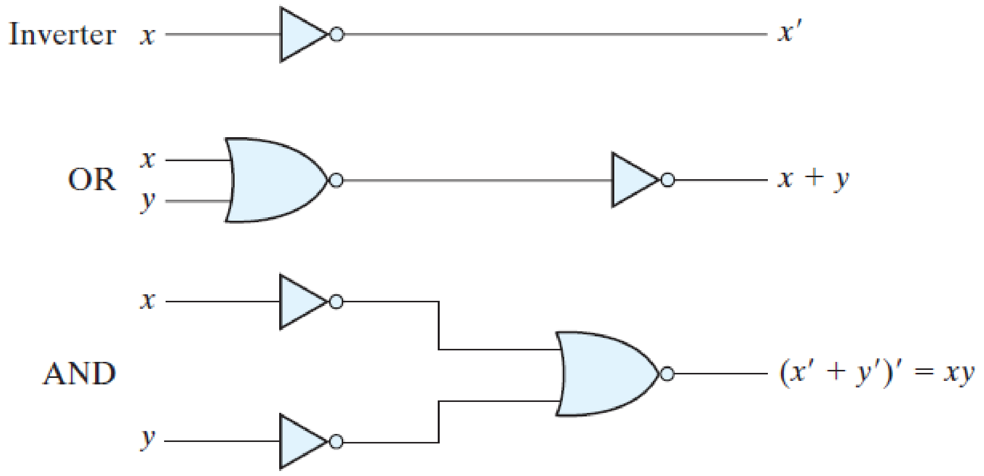


Figure 15. Logic operations with NOR gates

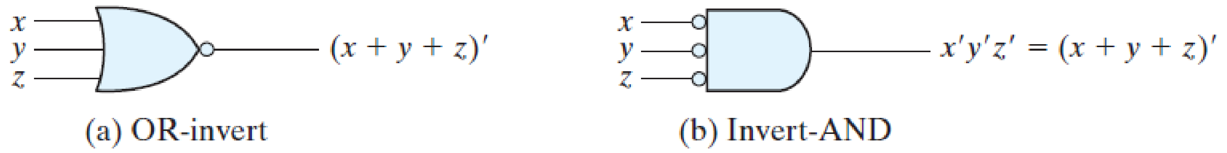


Figure 16.Two graphic symbols for the NOR gate

The two graphic symbols for the mixed notation are shown in Fig. 16. 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.

EXCLUSIVE-OR FUNCTION

The Exclusive-OR (XOR) denoted by the symbol $\oplus$ is a logical operation that performs the following Boolean operation:

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

The Exclusive-OR is equal to 1 if only x is equal to 1 or if only y is equal to 1 but not both. The Exclusive NOR also known as equivalence performs the following Boolean operation:

$$(x \oplus y)' = xy + x'y'$$

The Exclusive-NOR is equal to 1 if both x and y are equal to 1 or if both are equal to 0. The Exclusive-NOR can be shown to be the complement of the Exclusive-OR. The following identities apply to the Exclusive-OR operation:

- $x \oplus 0 = x$

- $x \oplus 1 = x'$
- $x \oplus x = 0$
- $x \oplus x' = 1$
- $x \oplus y' = x' \oplus y = (x \oplus y)'$

Exclusive-OR operation is both commutative and associative: .

- $A \oplus B = B \oplus A$
- $(A \oplus B) \oplus C = A \oplus (B \oplus C) = A \oplus B \oplus C$

Exclusive-OR is particularly useful in arithmetic operations and error detection and correction circuits.

TABULATION METHOD:

Quine-McClukey tabular method is a tabular method based on the concept of prime implicants. We know that **prime implicant** is a product or sum or sum term, which can't be further reduced by combining with any other product orsumorsum terms of the given Boolean function.

This tabular method is useful to get the prime implicants by repeatedly using the following Boolean identity.

$$xy + xy' = xy+y'y+y' = x.1 = x$$

Procedure of Quine-McCluskey Tabular Method

Follow these steps for simplifying Boolean functions using Quine-McClukey tabular method.

Step 1 − Arrange the given min terms in an **ascending order** and make the groups based on the number of ones present in their binary representations. So, there will be **at most 'n+1' groups** if there are 'n' Boolean variables in a Boolean function or 'n' bits in the binary equivalent of min terms.

Step 2 − Compare the min terms present in **successive groups**. If there is a change in only one-bit position, then take the pair of those two min terms. Place this symbol '_' in the differed bit position and keep the remaining bits as it is.

Step 3 − Repeat step2 with newly formed terms till we get all **prime implicants**.

Step 4 − Formulate the **prime implicant table**. It consists of set of rows and columns. Prime implicants can be placed in row wise and min terms can be placed in column wise. Place '1' in the cells corresponding to the min terms that are covered in each prime implicant.

Step 5 − Find the essential prime implicants by observing each column. If the min term is covered only by one prime implicant, then it is **essential prime implicant**. Those essential prime implicants will be part of the simplified Boolean function.

Step 6 − Reduce the prime implicant table by removing the row of each essential prime implicant and the columns corresponding to the min terms that are covered in that essential prime implicant. Repeat step 5 for Reduced prime implicant table. Stop this process when all min terms of given Boolean function are over.

Example 7. $f(W, X, Y, Z)=\sum m(2,6,8,9,10,11,14,15)f(W,X,Y,Z)=\sum m(2,6,8,9,10,11,14,15)$ using Quine-McClukey tabular method.

- The given Boolean function is in sum of min terms form. It is having 4 variables W, X, Y & Z.
- The given min terms are 2, 6, 8, 9, 10, 11, 14 and 15. The ascending order of these min terms based on the number of ones present in their binary equivalent is 2, 8, 6, 9, 10, 11, 14 and 15.

- The following table shows these min terms and their equivalent binary representations.

Group Name	**Min terms**	**W**	**X**	**Y**	**Z**
GA1	2	0	0	1	0
	8	1	0	0	0
GA2	6	0	1	1	0
	9	1	0	0	1
	10	1	0	1	0
GA3	11	1	0	1	1
	14	1	1	1	0
GA4	15	1	1	1	1

- The given min terms are arranged into 4 groups based on the number of ones present in their binary equivalents. The following table shows the possible merging of min terms from adjacent groups.

Group	**Min**	**W**	**X**	**Y**	**Z**

Name	**terms**				
GB1	2,6	0	-	1	0
	2,10	-	0	1	0
	8,9	1	0	0	-
	8,10	1	0	-	0
GB2	6,14	-	1	1	0
	9,11	1	0	-	1
	10,11	1	0	1	-
	10,14	1	-	1	0
GB3	11,15	1	-	1	1
	14,15	1	1	1	-

- The min terms, which are differed in only one-bit position from adjacent groups are merged. That differed bit is represented with this symbol, '-'. In this case, there are three groups and each group contains combinations of two min terms. The following table shows the possible **merging of min term pairs** from adjacent groups.

Group Name	**Min terms**	**W**	**X**	**Y**	**Z**
GB1	2,6,10,14	-	-	1	0
	2,10,6,14	-	-	1	0
	8,9,10,11	1	0	-	-
	8,10,9,11	1	0	-	-
GB2	10,11,14,15	1	-	1	-
	10,14,11,15	1	-	1	-

- The successive groups of min term pairs, which are differed in only one-bit position are merged. That differed bit is represented with this symbol, '-'. In this case, there are two groups and each group contains combinations of four min terms. Here, these combinations of 4 min terms are available in two rows. So, we can remove the repeated rows. The reduced table after removing the redundant rows is shown below.

Group Name	**Min terms**	**W**	**X**	**Y**	**Z**
GC1	2,6,10,14	-	-	1	0
	8,9,10,11	1	0	-	-
GC2	10,11,14,15	1	-	1	-

- Further merging of the combinations of min terms from adjacent groups is not possible, since they are differed in more than one-bit position. There are three rows in the above table. So, each row will give one prime implicant. Therefore, the **prime implicants** are YZ', WX' & WY.
- The **prime implicant table** is shown below.

Min terms / Prime Implicants	**2**	**6**	**8**	**9**	**10**	**11**	**14**	**15**
YZ'	1	1			1		1	
WX'			1	1	1	1		
WY					1	1	1	1

- The prime implicants are placed in row wise and min terms are placed in column wise. 1s are placed in the common cells of prime implicant rows and the corresponding min term columns.

- The min terms 2 and 6 are covered only by one prime implicant **YZ'**. So, it is an **essential prime implicant**. This will be part of simplified Boolean function. Now, remove this prime implicant row and the corresponding min term columns. The reduced prime implicant table is shown below.

Min terms / Prime Implicants	**8**	**9**	**11**	**15**
WX'	1	1	1	
WY			1	1

- The min terms 8 and 9 are covered only by one prime implicant **WX'**. So, it is an **essential prime implicant**. This will be part of simplified Boolean function. Now, remove this prime implicant row and the corresponding min term columns. The reduced prime implicant table is shown below.

Minterms / Prime Implicants	**15**
WY	1

- The min term 15 is covered only by one prime implicant **WY**. So, it is an **essential prime implicant**. This will be part of simplified Boolean function. In this example problem, we got three prime implicants and all the three are essential. Therefore, the **simplified Boolean function** is

$$\mathbf{F}(W,X,Y,Z) = \mathbf{YZ' + WX' + WY.}$$

Unit 3: Combinational Logic

1. Introduction
2. Combinational Circuits
3. Analysis Procedure
4. Design Procedure
5. Binary Adder–Subtractor
6. Decimal Adder
7. Binary Multiplier
8. Magnitude Comparator
9. Decoders
10. Encoders
11. Multiplexers

1. Introduction

- Logic circuits 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.
- The operation of combinational circuits can be specified logically by a set of Boolean functions.
- Sequential circuits contain *storage* elements in addition to logic gates.
- The outputs of sequential circuits are a function of the inputs and the state of the storage elements.
- Because the state of the storage elements is a function of previous inputs, the outputs of a sequential circuit depend not only on present values of inputs, but also on past inputs, and the circuit behavior must be specified by a time sequence of inputs and internal states.

2. Combinational Circuits

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

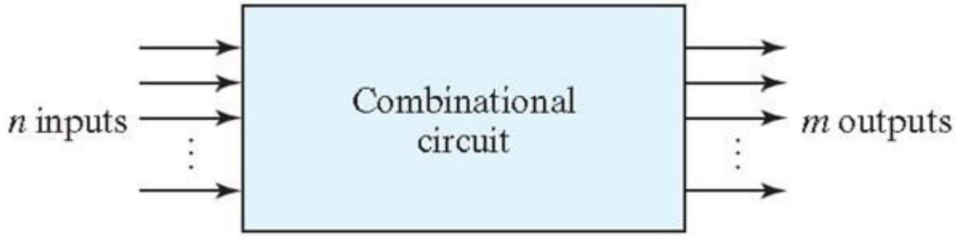


Figure 1. Combination circuit

- The *n* inputs come from an external source; the *m* outputs are produced by the combinational circuit and go to an external destination.
- Each input and output variable is an analog electrical signal whose values are interpreted to be a binary signal that represents logic 1 and logic 0.

- For n input variables, there are 2^n possible combinations.
- For each possible input combination, there is one possible value for each output.
- Thus, a combinational circuit can be specified with a truth table that lists the output values for each combination of input variables.
- A combinational circuit also can be described by m Boolean functions, one for each output variable.
- Each output function is expressed in terms of the n input variables.

- Several extensively used combinational circuits, such as adders, subtractors, multipliers, comparators, decoders, encoders, and multiplexers, are available in standard integrated circuit components and used as *standard cells* in complex very large scale integrated (VLSI) circuits.

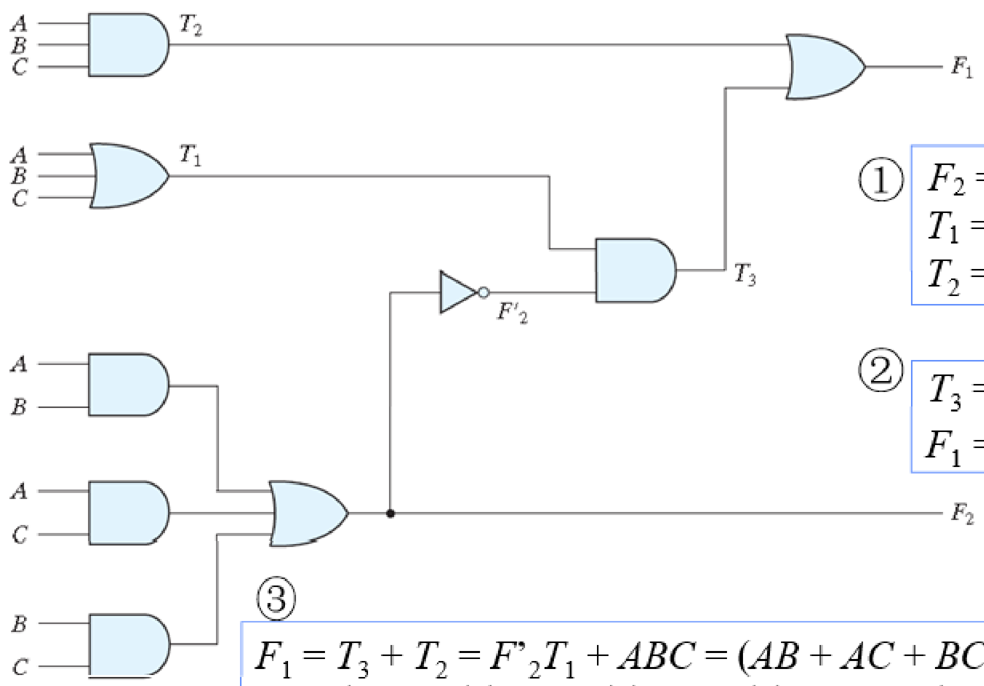
3. Analysis Procedure

- The analysis is to determine the function of an implemented circuit.
- This task starts with a given logic diagram and culminates with a set of Boolean functions, a truth table, or, possibly, an explanation of the circuit operation.
- The analysis can be performed manually by finding the Boolean functions or truth table or by using a computer simulation program.
- The first step in the analysis is to make sure that the given circuit is combinational and not sequential.
- **A combinational circuit has no feedback paths** or **memory elements**.
- A feedback path is a connection from the output of one gate to the input of a second gate whose output forms part of the input to the first gate.
- Feedback paths in a digital circuit define a sequential circuit.

Steps to Obtain Output Boolean Functions

- To obtain the output Boolean functions from a logic diagram, we proceed as follows:
  1. Label all gate outputs that are a function of input variables with arbitrary symbols—but with meaningful names. Determine the Boolean functions for each gate output.
  2. Label the gates that are a function of input variables and previously labeled gates with other arbitrary symbols. Find the Boolean functions for these gates.
  3. Repeat the process outlined in step 2 until the outputs of the circuit are obtained.
  4. By repeated substitution of previously defined functions, obtain the output Boolean functions in terms of input variables.

Example



①
$$F_2 = AB + AC + BC$$
$$T_1 = A + B + C$$
$$T_2 = ABC$$

②
$$T_3 = F_2 T_1$$
$$F_1 = T_3 + T_2$$

③
$$\begin{aligned}F_1 = T_3 + T_2 = F'_2 T_1 + ABC &= (AB + AC + BC)'(A + B + C) + ABC \\ &= (A' + B')(A' + C')(B' + C')(A + B + C) + ABC \\ &= (A' + B'C')(AB' + AC' + BC' + B'C) + ABC \\ &= A'BC' + A'B'C + AB'C' + ABC\end{aligned}$$

④
F_1 ➔ sum of a full-adder, F_2 ➔ carry of a full-adder

Figure 2. Full Adder Implementation

Steps to Obtain Truth Table

- Obtain the truth table directly from the logic diagram as follows:
  1. Determine the number of input variables. For *n* inputs, form the 2^n possible input combinations and list the binary numbers from 0 to $(2^n - 1)$ in a table.
  2. Label the outputs of selected gates with arbitrary symbols.
  3. Obtain the truth table for the outputs of those gates which are a function of the input variables only.

A	B	C	F_2	F'_2	T_1	T_2	T_3	F_1
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1

Figure 3. Truth Table

4. Proceed to obtain the truth table for the outputs of those gates which are a function of previously defined values until the columns for all outputs are determined.

4. Design Procedure

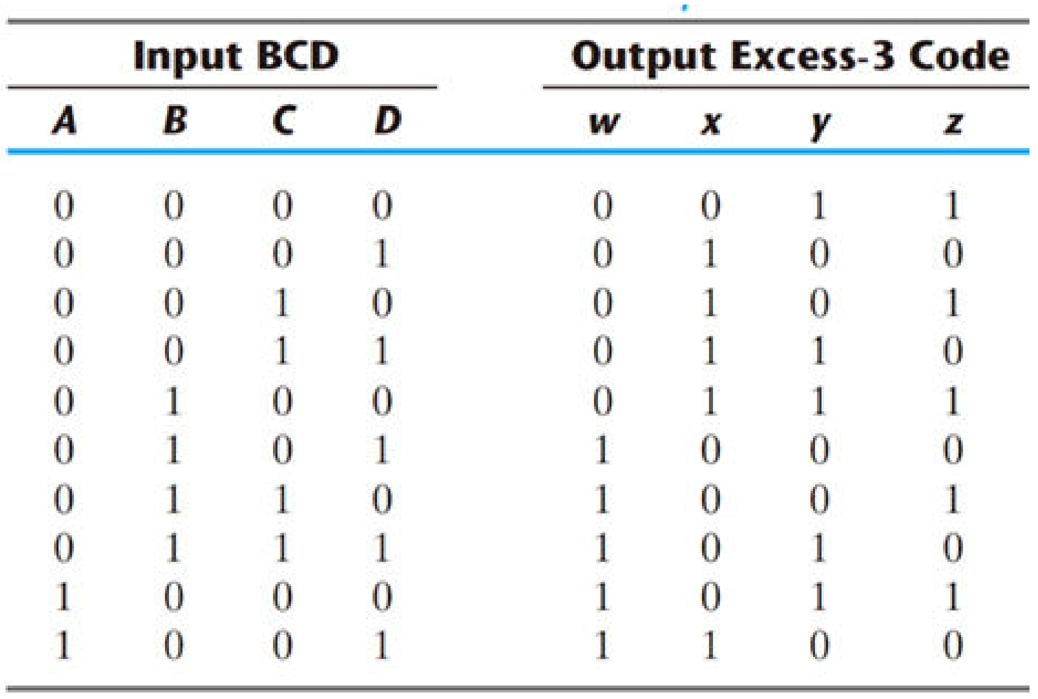
- The design is to derive a logic circuit or a set of Boolean functions from the specification of the design objective.
- The design procedure involves the following steps:
  1. From the specifications of the circuit, determine the required number of inputs and outputs and assign a symbol to each.
  2. Derive the truth table that defines the required relationship between inputs and outputs.
  3. Obtain the simplified Boolean functions for each output as a function of the input variables.
  4. Draw the logic diagram and verify the correctness of the design (manually or by simulation).

Design Exploration

- Truth table gives the exact definition of a combinational circuit.
- The output binary functions listed in the truth table are simplified by any available method, such as algebraic manipulation, the map method, or a computer-based simplification program.
- Practical design must consider such constraints as the number of gates, number of inputs to a gate, propagation time of the signal through the gates, number of interconnections, limitations of the driving capability of each gate (i.e., the number of gates to which the output of the circuit may be connected), and various other criteria that must be taken into consideration when designing integrated circuits.
- Since each constraint is dictated by the particular application, it is difficult to make a general statement about what constitutes an acceptable implementation.
- In most cases, the simplification begins by satisfying an elementary objective, such as producing the simplified Boolean functions in a standard form, then the simplification proceeds with further steps to meet other performance criteria.

Code Conversion Example

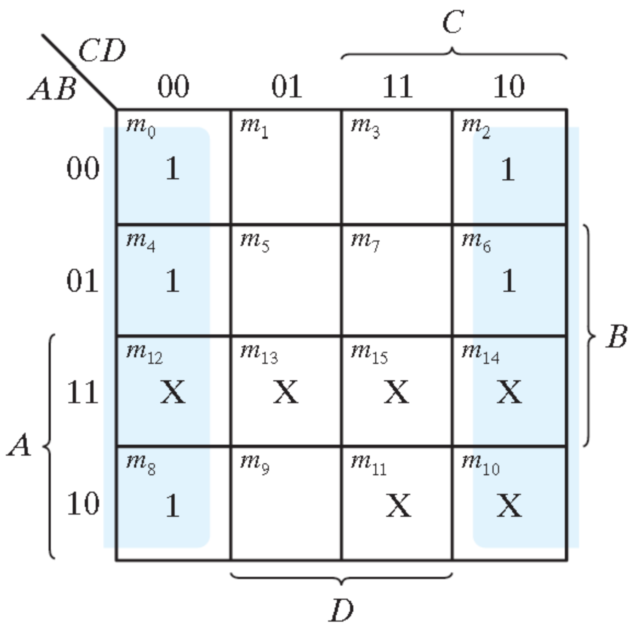
- BCD to excess-3 code
  - The truth table



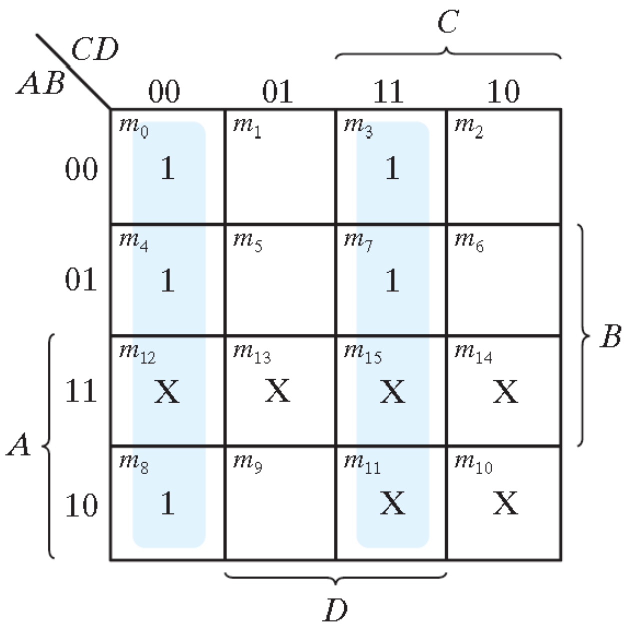
Input BCD				Output Excess-3 Code			
A	*B*	*C*	*D*	*w*	*x*	*y*	*z*
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

  -

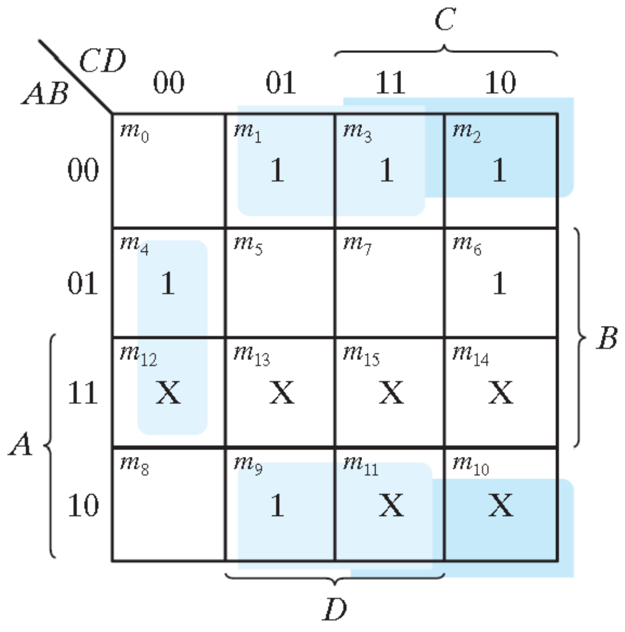
Figure 4. Truth Table for Conversion


CD
AB
C
00 01 11 10
00: m_0 1, m_1, m_3, m_2 1
01: m_4 1, m_5, m_7, m_6 1
11: m_{12} X, m_{13} X, m_{15} X, m_{14} X
10: m_8 1, m_9, m_{11} X, m_{10} X
A
B
D

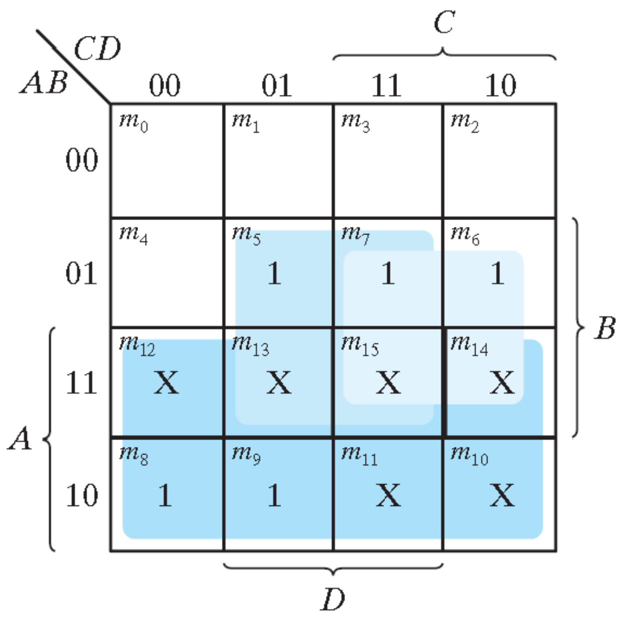
$z = D'$


CD
AB
C
00 01 11 10
00: m_0 1, m_1, m_3 1, m_2
01: m_4 1, m_5, m_7 1, m_6
11: m_{12} X, m_{13} X, m_{15} X, m_{14} X
10: m_8 1, m_9, m_{11} X, m_{10} X
A
B
D

$y = CD + C'D'$


CD
AB
C
00 01 11 10
00: m_0, m_1 1, m_3 1, m_2 1
01: m_4 1, m_5, m_7, m_6 1
11: m_{12} X, m_{13} X, m_{15} X, m_{14} X
10: m_8, m_9 1, m_{11} X, m_{10} X
A
B
D

$x = B'C + B'D + BC'D'$


CD
AB
C
00 01 11 10
00: m_0, m_1, m_3, m_2
01: m_4, m_5 1, m_7 1, m_6 1
11: m_{12} X, m_{13} X, m_{15} X, m_{14} X
10: m_8 1, m_9 1, m_{11} X, m_{10} X
A
B
D

$w = A + BC + BD$

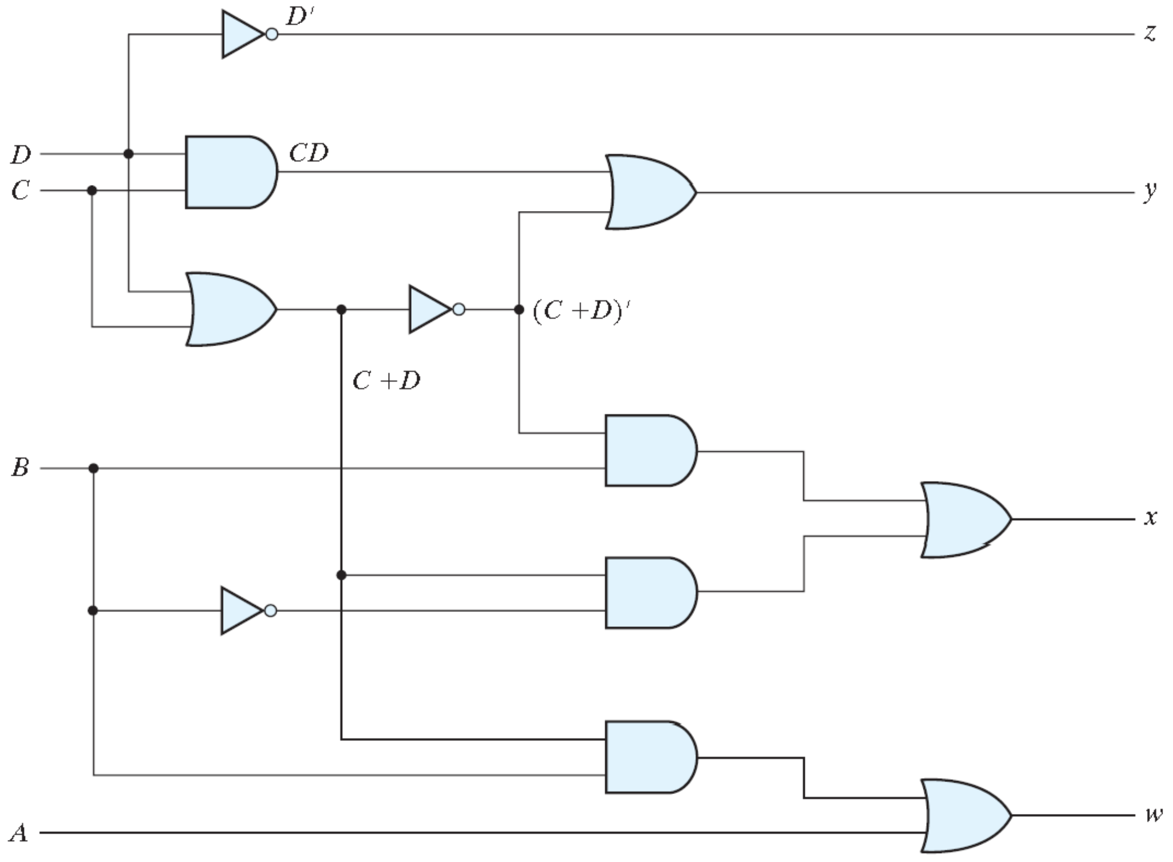
Simplified Functions and Logic Diagram

$$z = D'$$

$$y = CD + C'D' = CD + (C + D)'$$

$$x = B'C + B'D + BC'D' = B'(C + D) + BC'D'$$
$$= B'(C + D) + B(C + D)'$$

$$w = A + BC + BD = A + B(C + D)$$



5. Binary Adder-Subtractor

- Half adder
  - 0 + 0 = 0; 0 + 1 = 1; 1 + 0 = 1; 1 + 1 = 10
  - two input variables: *x*, *y*
  - two output variables: *C* (carry), *S* (sum)
  - truth table

Half Adder

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

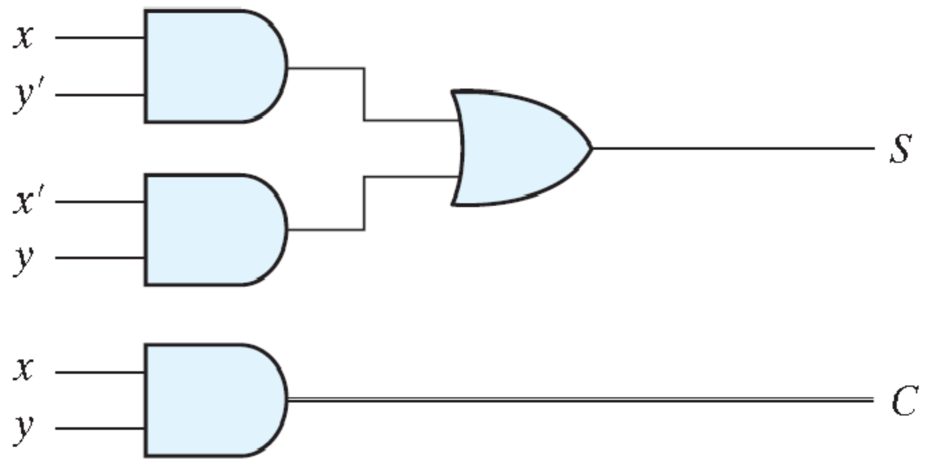
  - Boolean functions

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

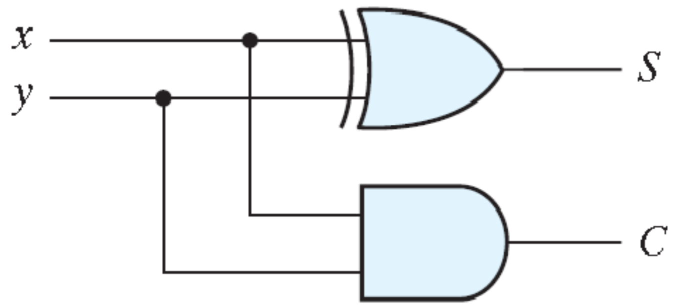
Implementation of Half Adder

- sum of products

exclusive-OR and AND



(a) $S = xy' + x'y$
$C = xy$



(b) $S = x \oplus y$
$C = xy$

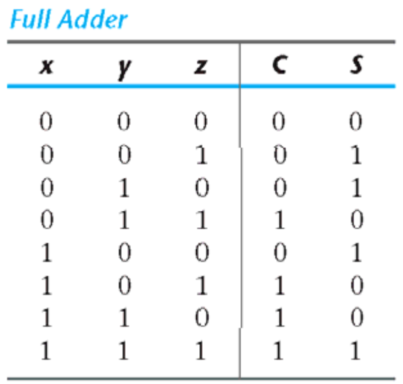
Full-Adder

- The arithmetic sum of three input bits
- Three input bits
  - *x*, *y*: two significant bits
  - *z*: the carry bit from the previous lower significant bit
- Two output bits: *C*, *S*
- Truth table

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

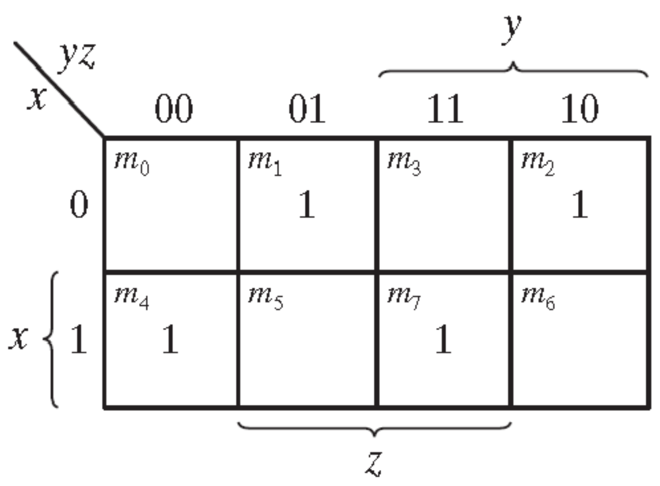
Boolean Functions of Full Adder



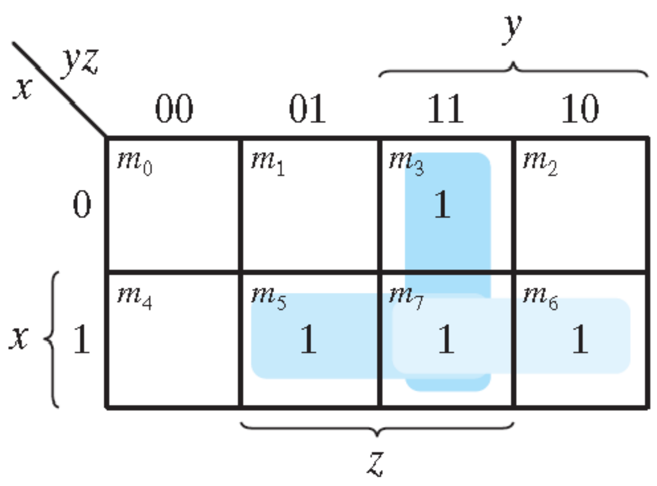
Full Adder

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

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



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

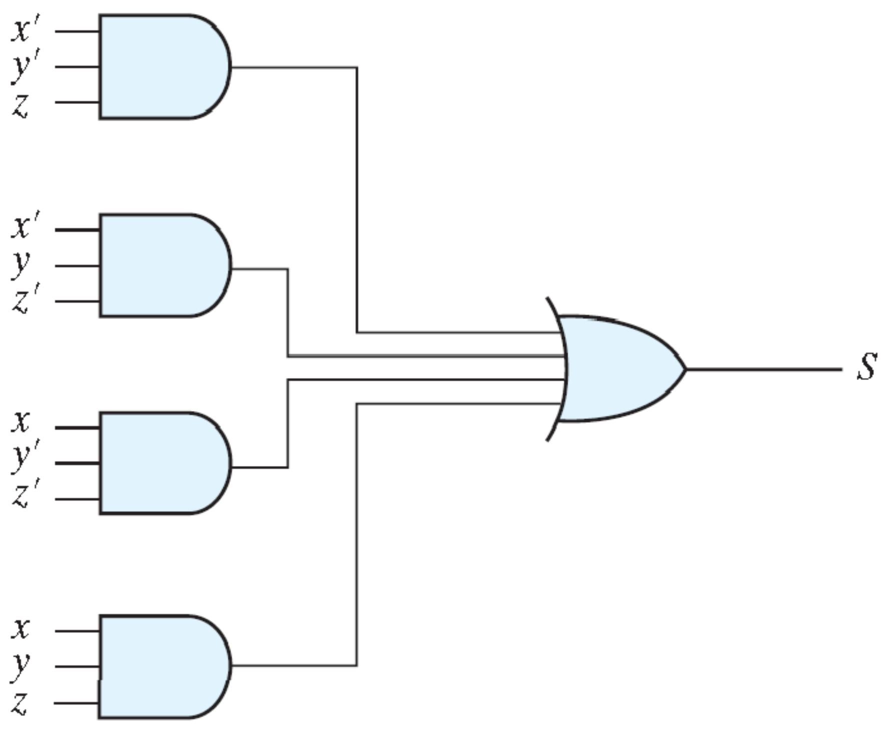


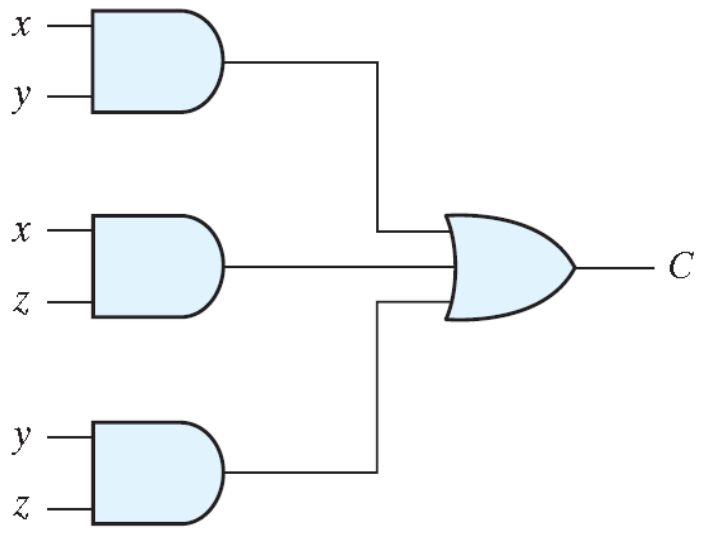
(b) $C = xy + xz + yz$

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

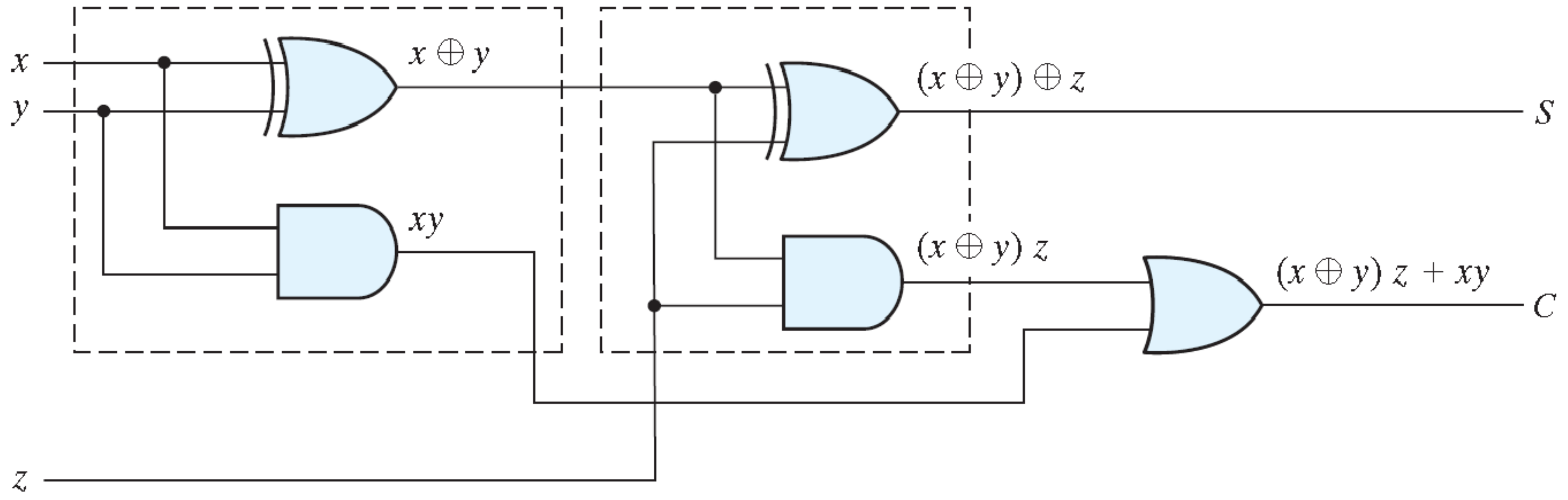
$$C = xy + (x + y)\, z$$

Sum-of-Product Implementation of Full Adder





Implementation of Full Adder with Half Adders



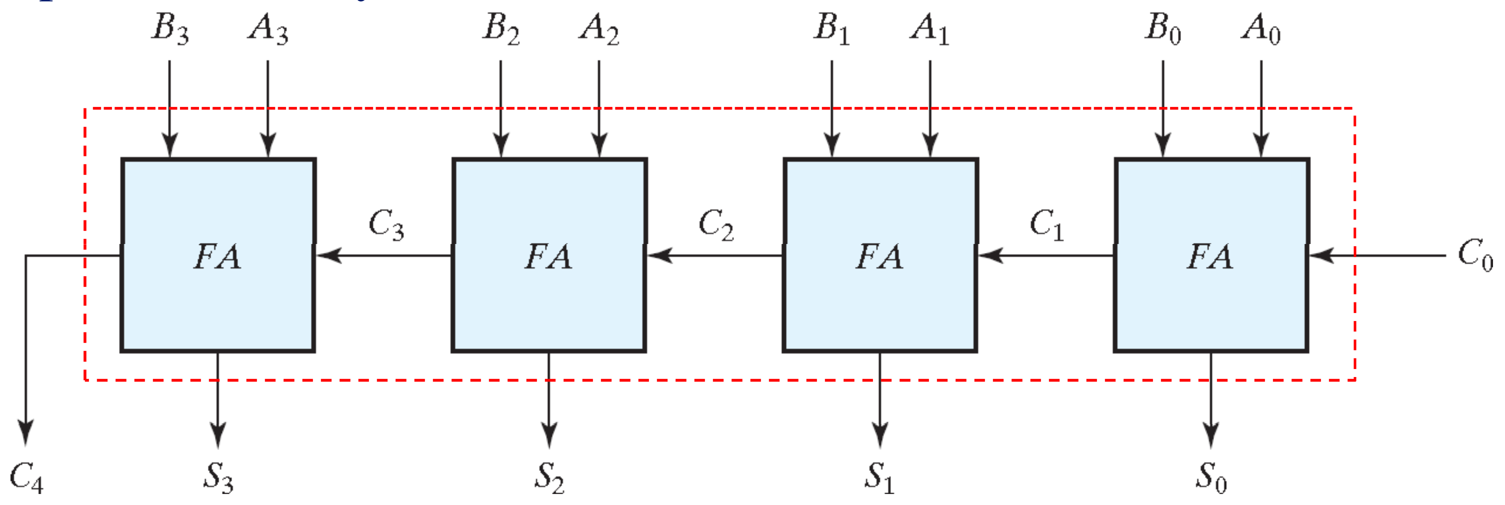
$$S = x \oplus y \oplus z$$

$$\begin{aligned} C &= xy + (x + y)\, z \\ &= xy + (x \oplus y)\, z \end{aligned}$$

Binary Adder

- A binary adder 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 (called *ripple-carry* adder).
- Example: 4-bit binary adder

Is it possible to design it with truth table?



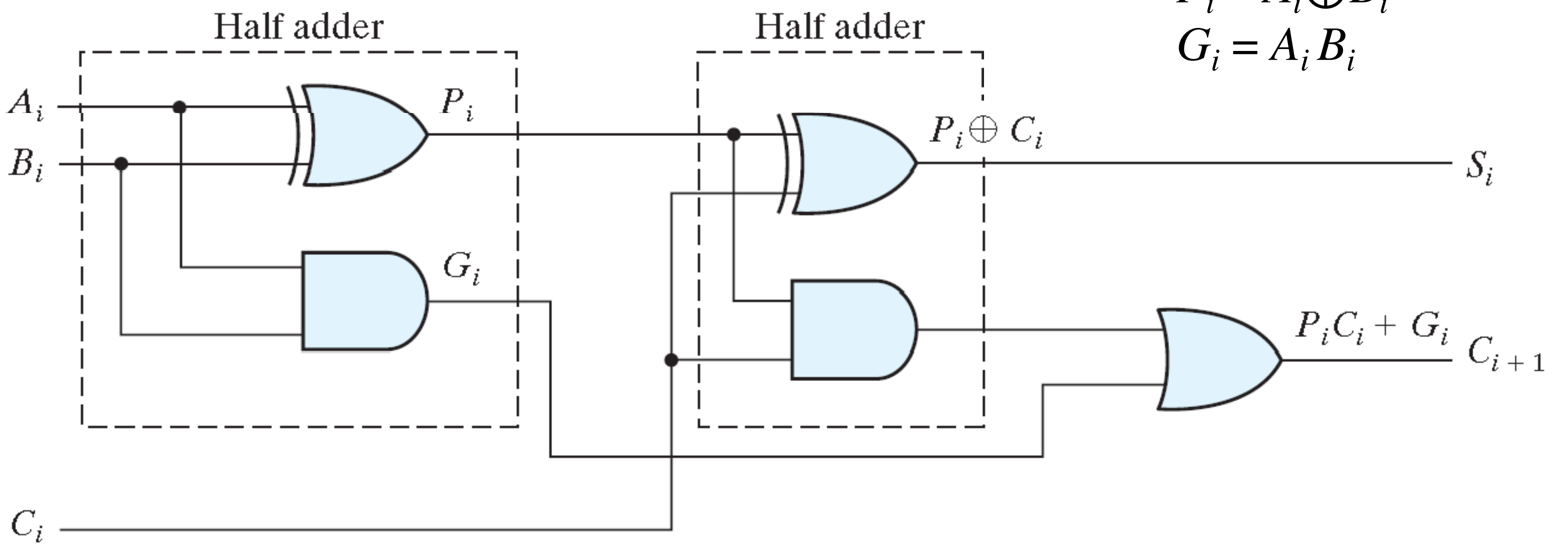
Carry Propagation

- The total propagation time is equal to the sum of the propagation delay of logic gates along a path from input to output.
- The longest propagation delay time in an adder is the time it takes the carry to propagate through the full adders. (called *critical path*)
- Each bit of the sum output, S_i, in the adder will be in its steady-state final value only after the input carry to that stage has been propagated.
- Consider output S_3 in the 4-bit adder, inputs A_3 and B_3 are available as soon as input signals are applied to the adder.
- However, input carry C_3 does not settle to its final value until C_2 is available from the previous stage.
- Similarly, C_2 has to wait for C_1 and so on down to C_0.
- Thus, only after the carry propagates and ripples through all stages will the last output S_3 and carry C_4 settle to their final correct value.
- This will be a serious problem if the adder has a long bit length.

Carry Propagation in Binary Adder

- Re-label the half-adder implementation with $P_i = A_i \oplus B_i$ and $G_i = A_i B_i$
- P_i and G_i settle to steady-state values after A_i and B_i propagate through their respective gates.
- The input carry C_i to the output carry C_{i+1} propagates through an AND gate and an OR gate. If there are four full adders in the adder, the output carry C_4 would have 2 * 4 = 8 gate levels from C_0 to C_4.
- For an n-bit adder, there are $2n$ gate levels for the carry to propagate from input to output.

$$P_i = A_i \oplus B_i$$
$$G_i = A_i B_i$$

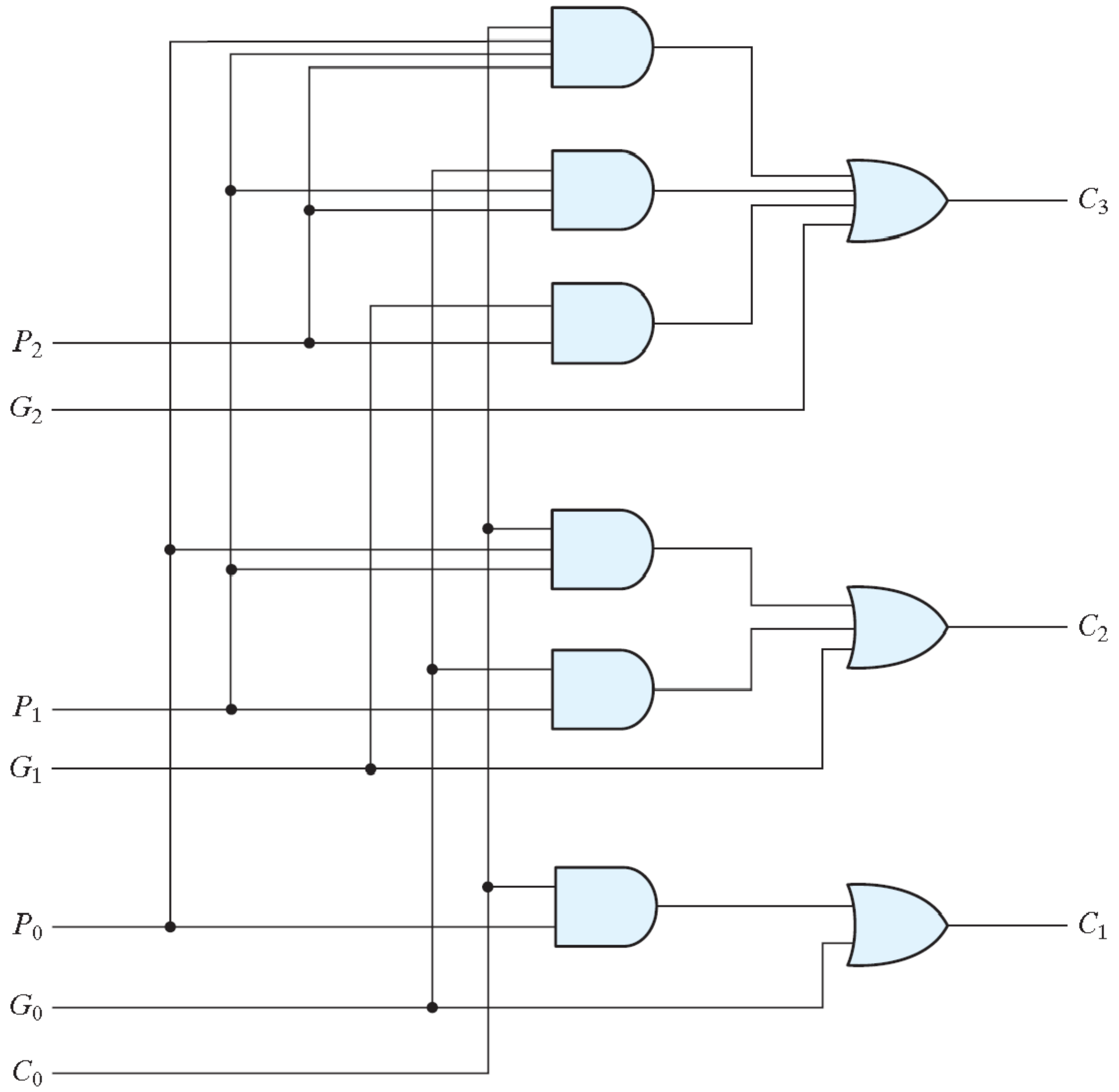


Carry Look-ahead Adder

- Reduce the carry propagation delay using look-ahead carry (more complex mechanism, yet faster)
- Two signals defined: *Carry Propagate*: $P_i = A_i \oplus B_i$ and *Carry Generate*: $G_i = A_i B_i$
- Sum and Carry are re-defined as: $S_i = P_i \oplus C_i$ and $C_{i+1} = G_i + P_i C_i$
- The carry signals of the adder become
  - C_0 = carry input
  - $C_1 = G_0 + P_0 C_0$
  - $C_2 = G_1 + P_1 C_1 = G_1 + P_1(G_0 + P_0 C_0) = G_1 + P_1 G_0 + P_1 P_0 C_0$
  - $C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$
- C_3 does not have to wait for C_2 and C_1 to propagate; in fact, C_3 is propagated at the same time as C_1 and C_2.

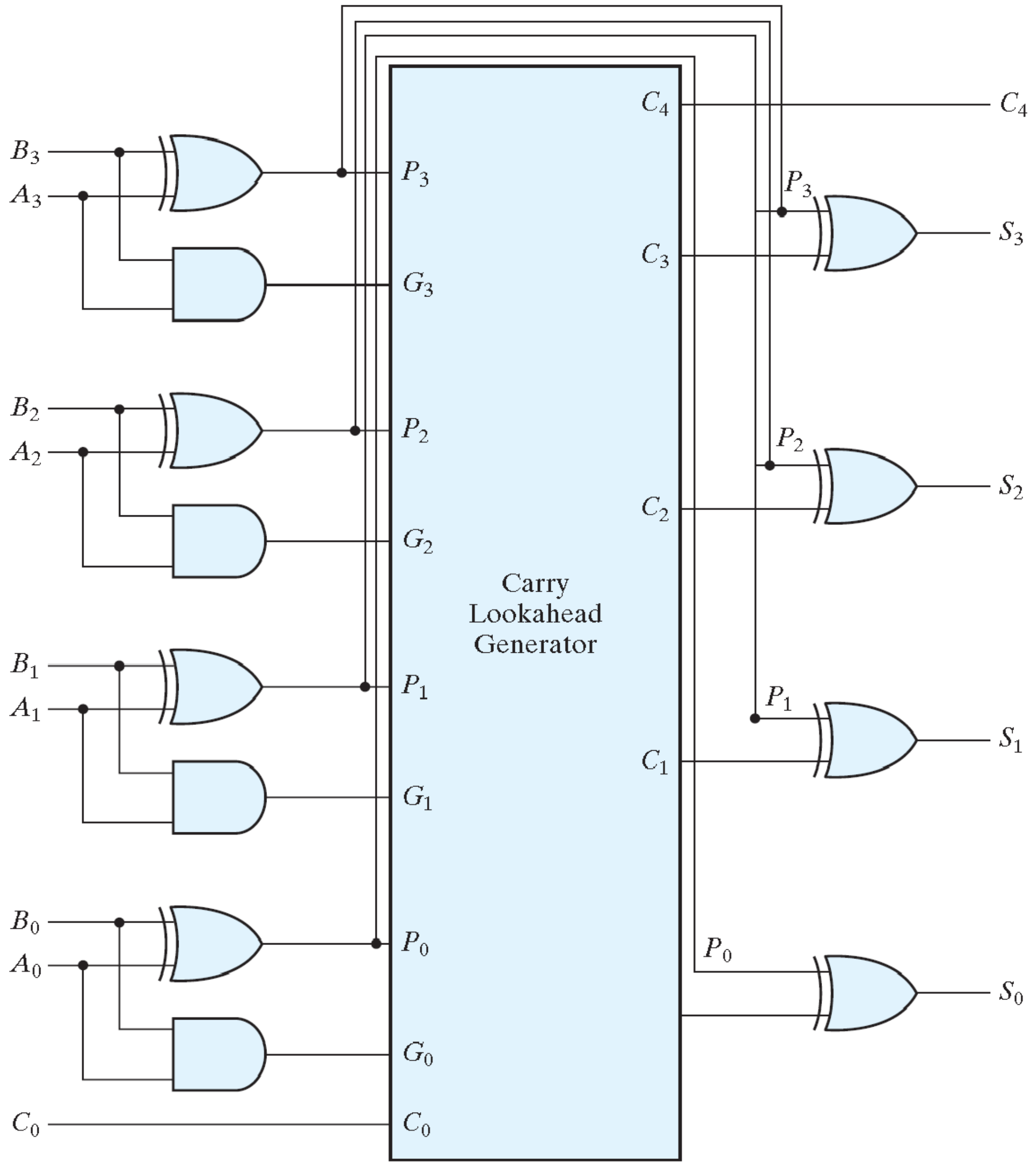
Carry Look-ahead Generator

$$P_i = A_i \oplus B_i$$
$$G_i = A_i B_i$$



Where is its longest propagation delay path?

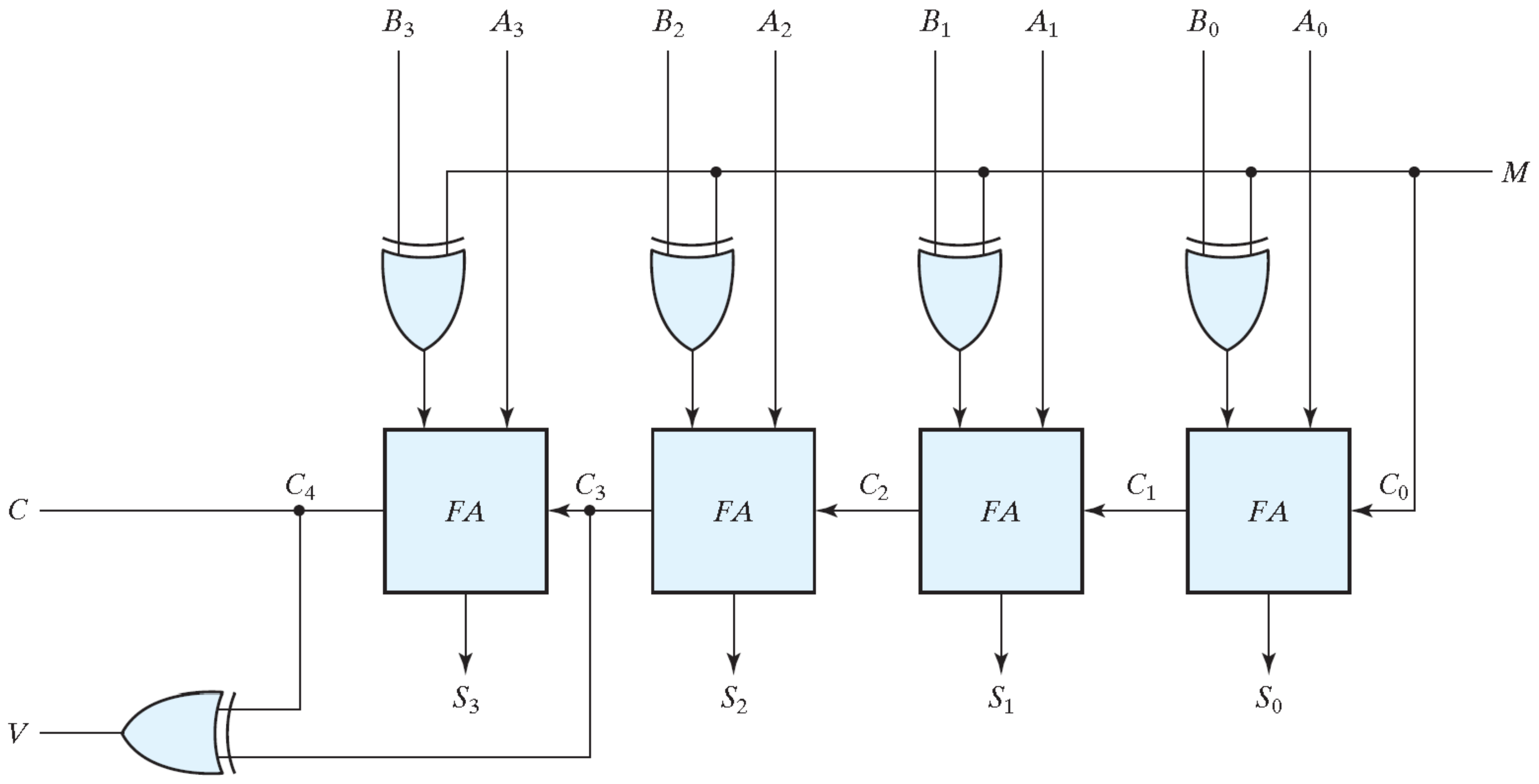
Four-bit Carry Look-ahead Adder



There exist many other faster adders and are not mentioned in this course.

Binary Adder-Subtractor

- $A - B = A + (2\text{'s complement of } B)$
- 4-bit adder-subtractor
  - $M = 0 \Rightarrow A + B$; $M = 1 \Rightarrow A + B' + 1$
- Output V is for detecting an *overflow*.



Overflow

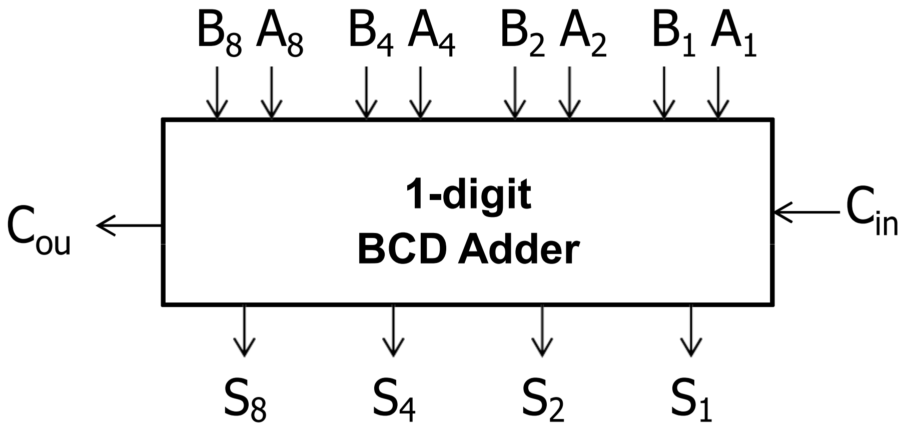
- Overflow: two *n*-digit numbers are added and the sum becomes (*n*+1)-digit.
- Overflow is a problem in computers because the number of bits that hold the number is finite and a (*n*+1)-bit result cannot be stored in an *n*-bit word.
- Many computers detect the occurrence of an overflow, a corresponding flip-flop (1-bit memory) is set that can then be checked by the user (or program).
- Add two positive numbers and obtain a negative number
- Add two negative numbers and obtain a positive number
- An overflow condition can be detected by observing the carry into the sign bit position and the carry out of the sign bit position.
- $V = 0$ → no overflow; $V = 1$→ overflow (see previous page)

Example: 8-bit signed addition, 2's complement, ranges -128 ~ +127

$$\begin{array}{lrr} \text{carries:} & 0\ \ 1 & \\ & +70 & 0\ \ 1000110 \\ & +80 & 0\ \ 1010000 \\ \hline & +150 & 1\ \ 0010110 \end{array} \qquad \begin{array}{lrr} \text{carries:} & 1\ \ 0 & \\ & -70 & 1\ \ 0111010 \\ & -80 & 1\ \ 0110000 \\ \hline & -150 & 0\ \ 1101010 \end{array}$$

6. Decimal Adder

- Add two BCD's
  - 9 inputs: two BCD's and one carry-in
  - 5 outputs: one BCD and one carry-out



- Design approaches
  - A truth table with 2^9 entries
  - use binary full Adders
    - » the decimal sum must be not larger than 19 (= 9 + 9 + 1)
    - » the BCD sum is no larger than 9; $(S_8S_4S_2S_1) \leqq (1001)$

The Sum of a BCD Adder

Binary Sum					BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

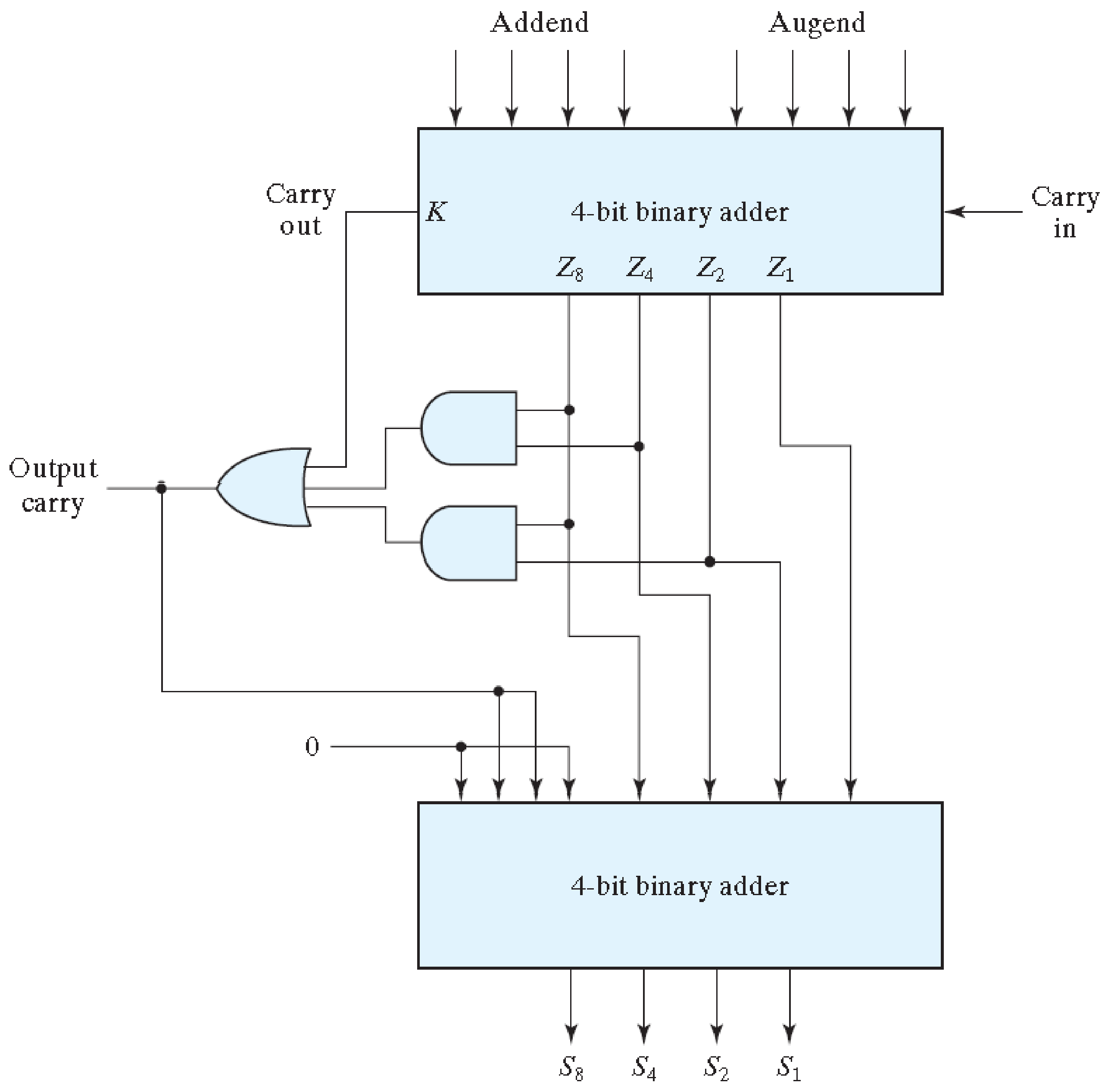
BCD Adjustment

- When the binary sum is equal to or less than 1001, the corresponding BCD number is identical, no conversion is needed.
- When the binary sum is greater than 1001, an addition of 6 (0110) converts it to the correct BCD representation and also produces an output carry as required.
- Modifications are needed if the sum > 9 (1001)
  - C must be set to 1, if
    - » $K = 1$, or
    - » $Z_8Z_4 = 1$, or
    - » $Z_8Z_2 = 1$

$$\begin{aligned} C &= K + Z_8Z_4 + Z_8Z_2 \\ &= K + Z_8(Z_4 + Z_2) \end{aligned}$$

- When $C = 1$, add 0110 to the binary sum.

BCD Adder


Addend
Augend
Carry out
K
4-bit binary adder
Carry in
Z8 Z4 Z2 Z1
Output carry
0
4-bit binary adder
S8 S4 S2 S1

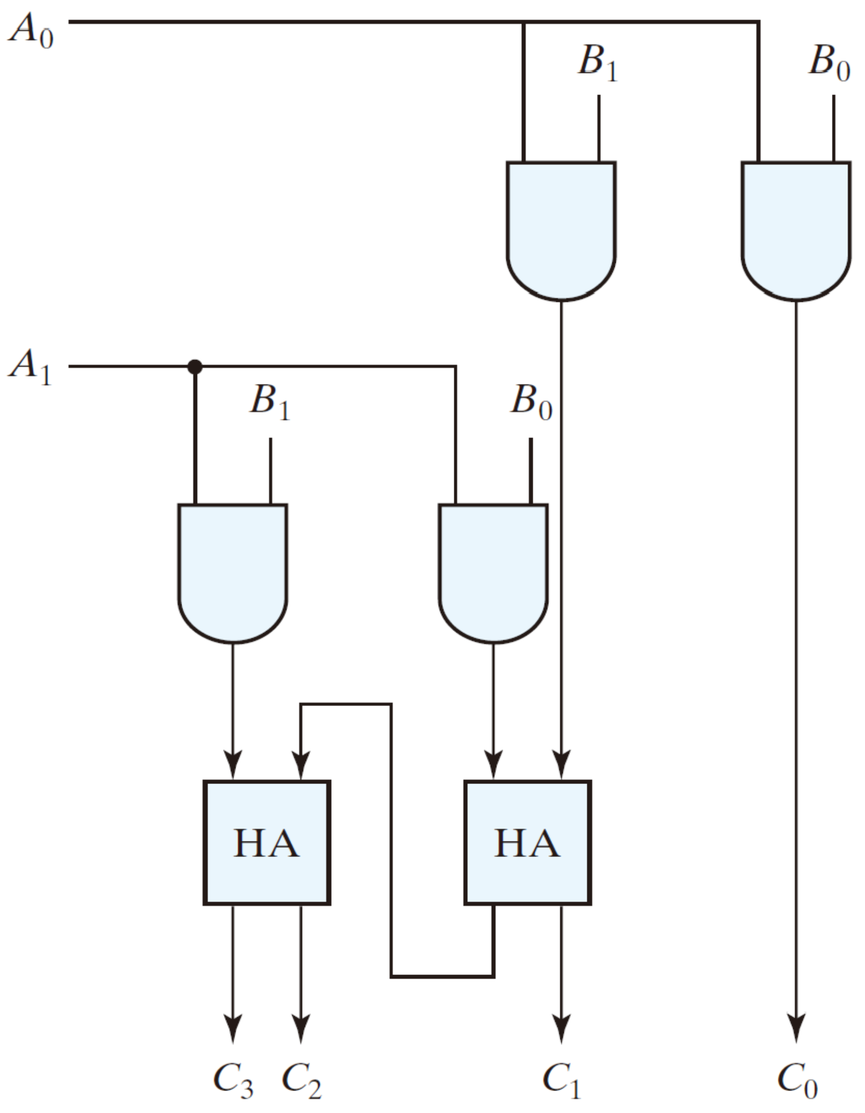
Binary Multiplier

- Performed in the same way as multiplication of decimal numbers.
- Partial products: AND operations.
- 2-bit x 2-bit ➔ 4-bit

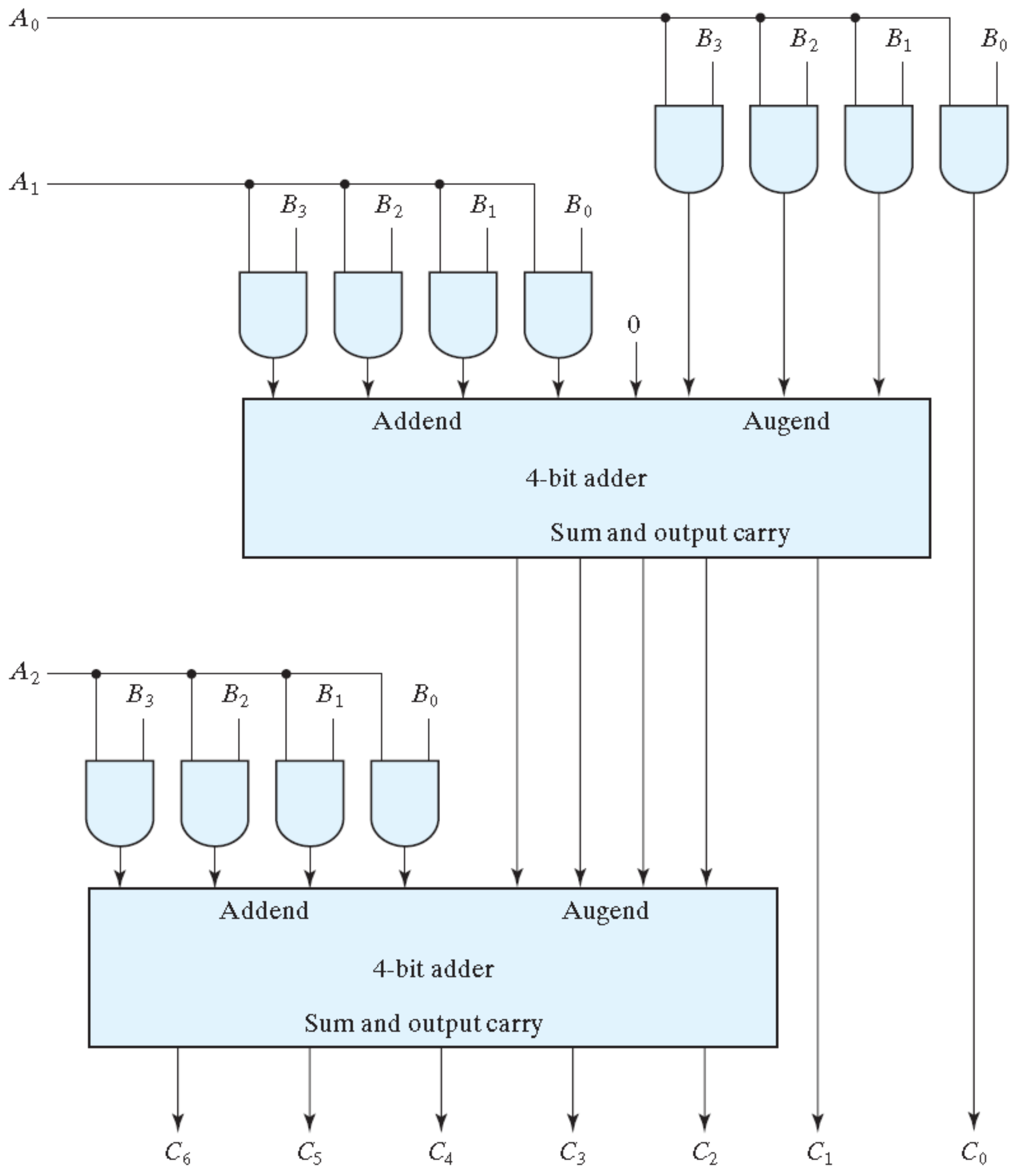
$$\begin{array}{cccc} & & B_1 & B_0 \\ & \times & A_1 & A_0 \\ \hline & & A_0B_1 & A_0B_0 \\ + & A_1B_1 & A_1B_0 & \\ \hline C_3 & C_2 & C_1 & C_0 \end{array}$$

1-bit multiplication ➔ AND

0 x 0 = 0
0 x 1 = 0
1 x 0 = 0
1 x 1 = 1



- For *J* multiplier and *K* multiplicand bits, we need (*J* * *K*) AND gates and (*J* - 1) *K*-bit adders to produce a product of (*J* + *K*) bits.
- *K* = 4 and *J* = 3:
  - 12 AND gates and
  - two 4-bit adders
  - produce a 7-bit product.



There are a lot of multipliers has been presented for high speed applications.

8. Magnitude Comparator

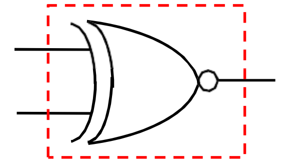
- A *magnitude comparator* compares two numbers *A* and *B* and determines their relative magnitudes.
- The results of comparison between two numbers are:
  - $A > B$, $A = B$, $A < B$
- Design Approaches
  - the truth table for two n-bit numbers comparison
    - » 2^{2n} entries - too cumbersome for large *n*
  - use inherent regularity of the problem (algorithm approach)
    - » algorithm — a procedure which specifies a finite set of steps
    - » reduce design efforts
    - » reduce human errors

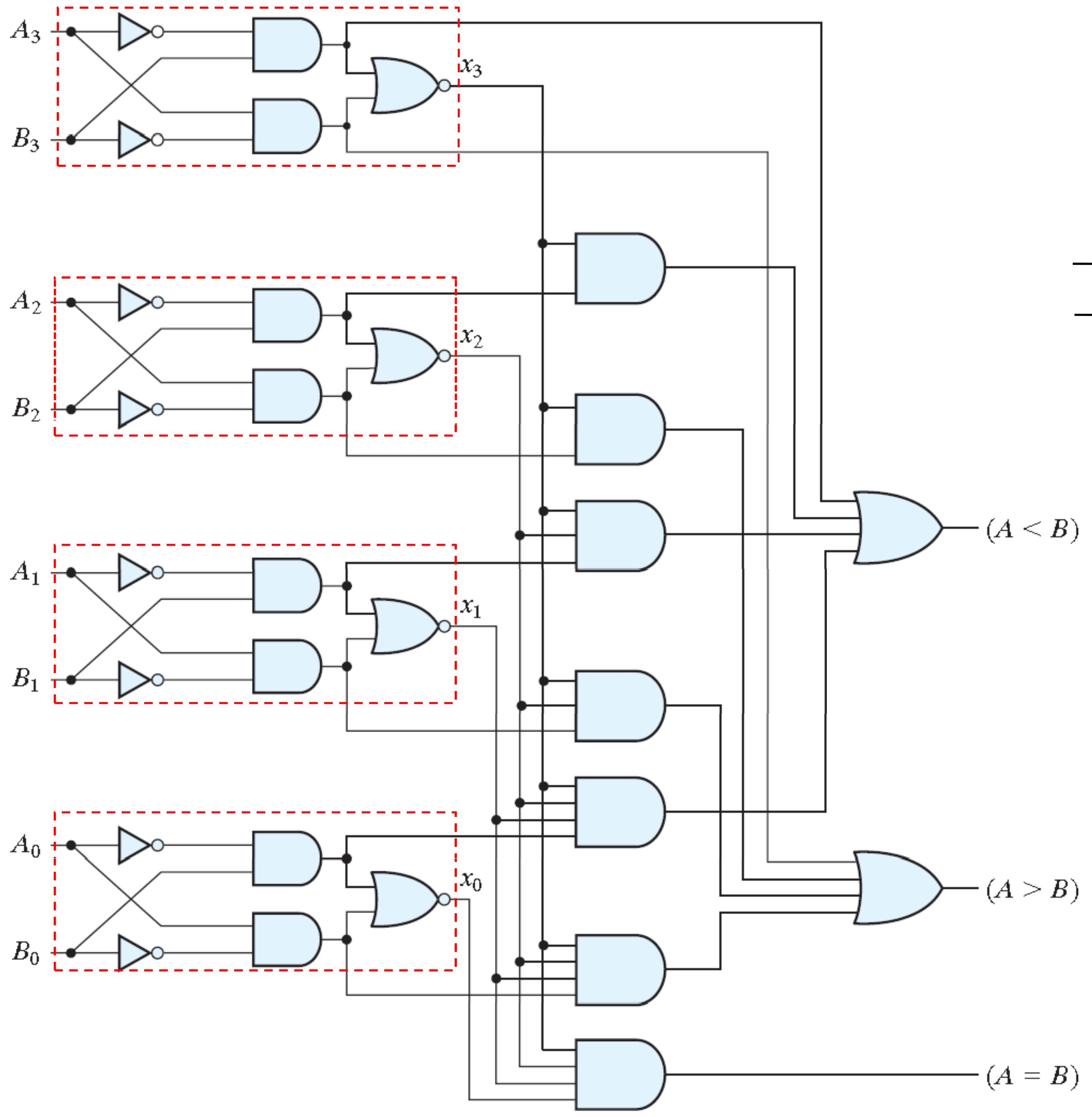
Comparison Algorithm

- Consider two 4-bit numbers, $A = A_3A_2A_1A_0$, $B = B_3B_2B_1B_0$
- A and B are equal ($A = B$) if $A_3 = B_3$, $A_2 = B_2$, $A_1 = B_1$, and $A_0 = B_0$.
- The equality of each pair of bits can be expressed with an exclusive-NOR function as: $x_i = A_iB_i + A_i'B_i'$ for $i = 0, 1, 2, 3$
- $x_i = 1$ only if the pair of bits in position i are equal (both are 1 or both are 0).
- For equality to exist ($A = B$), all x_i variables must be equal to 1: $(A = B) = x_3x_2x_1x_0$

- To determine whether ($A > B$) or ($A < B$), starting from the MSB, if the two bits are equal, then compare the next lower significant pair of bits until a pair of unequal bits is reached.
- If the corresponding bit of A is 1 and that of B is 0, we conclude that $A > B$.
- If the corresponding digit of A is 0 and that of B is 1, we have $A < B$.
- The sequential comparison can be expressed by the two Boolean functions
  - $(A > B) = A_3B_3' + x_3A_2B_2' + x_3x_2A_1B_1' + x_3x_2x_1A_0B_0'$
  - $(A < B) = A_3'B_3 + x_3A_2'B_2 + x_3x_2A_1'B_1 + x_3x_2x_1A_0'B_0$

XNOR





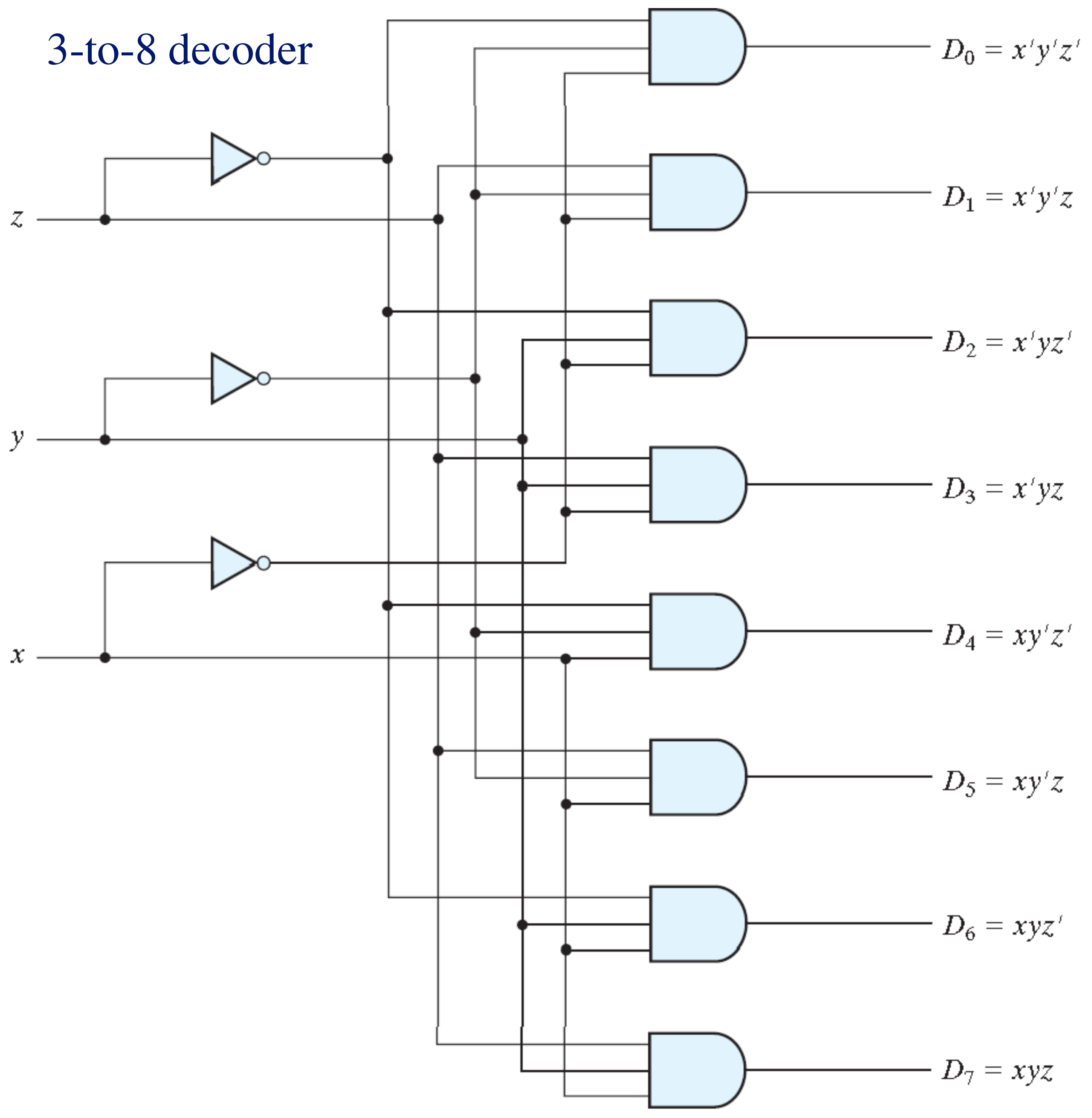
9. Decoders

- A *decoder* converts binary information from *n* input lines to a maximum of 2^n unique output lines.
- A *n*-to-*m* decoder ($m \leqq 2^n$)
  - a binary code of *n* bits has 2^n distinct information
  - *n* input variables; up to 2^n output lines
  - only one output can be active (high) at any time

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

3-to-8 decoder

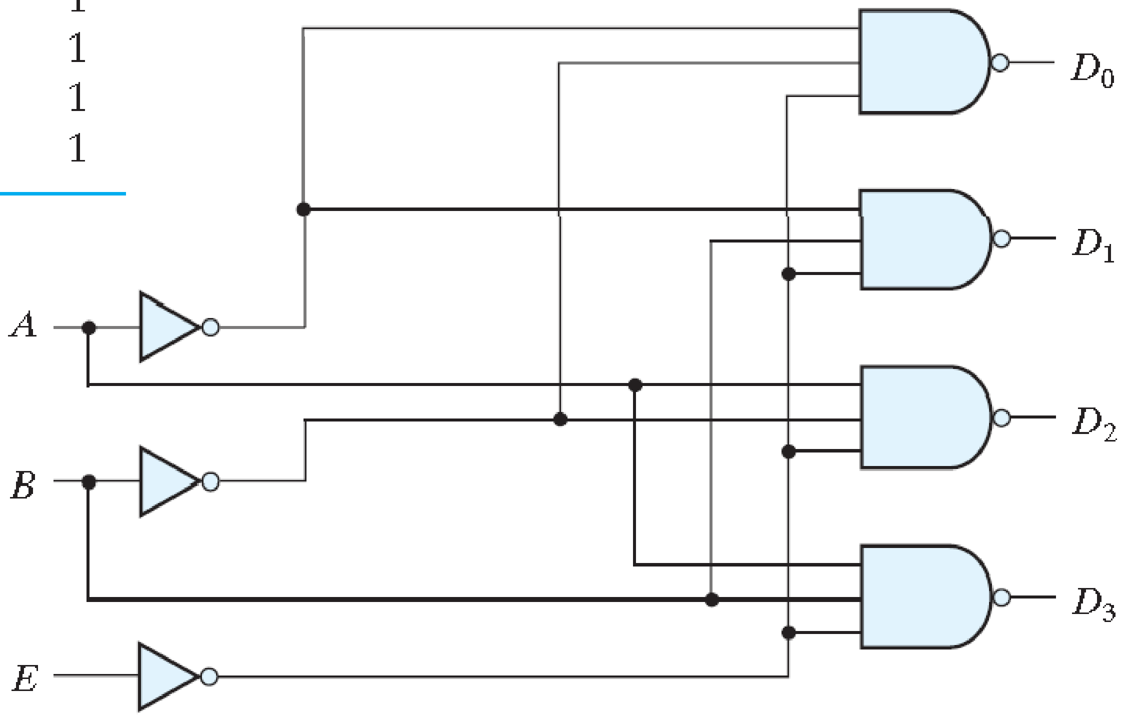


each output = a minterm

Two-to-four Decoder with Enable

- *Enable* input is added to control the circuit operation.

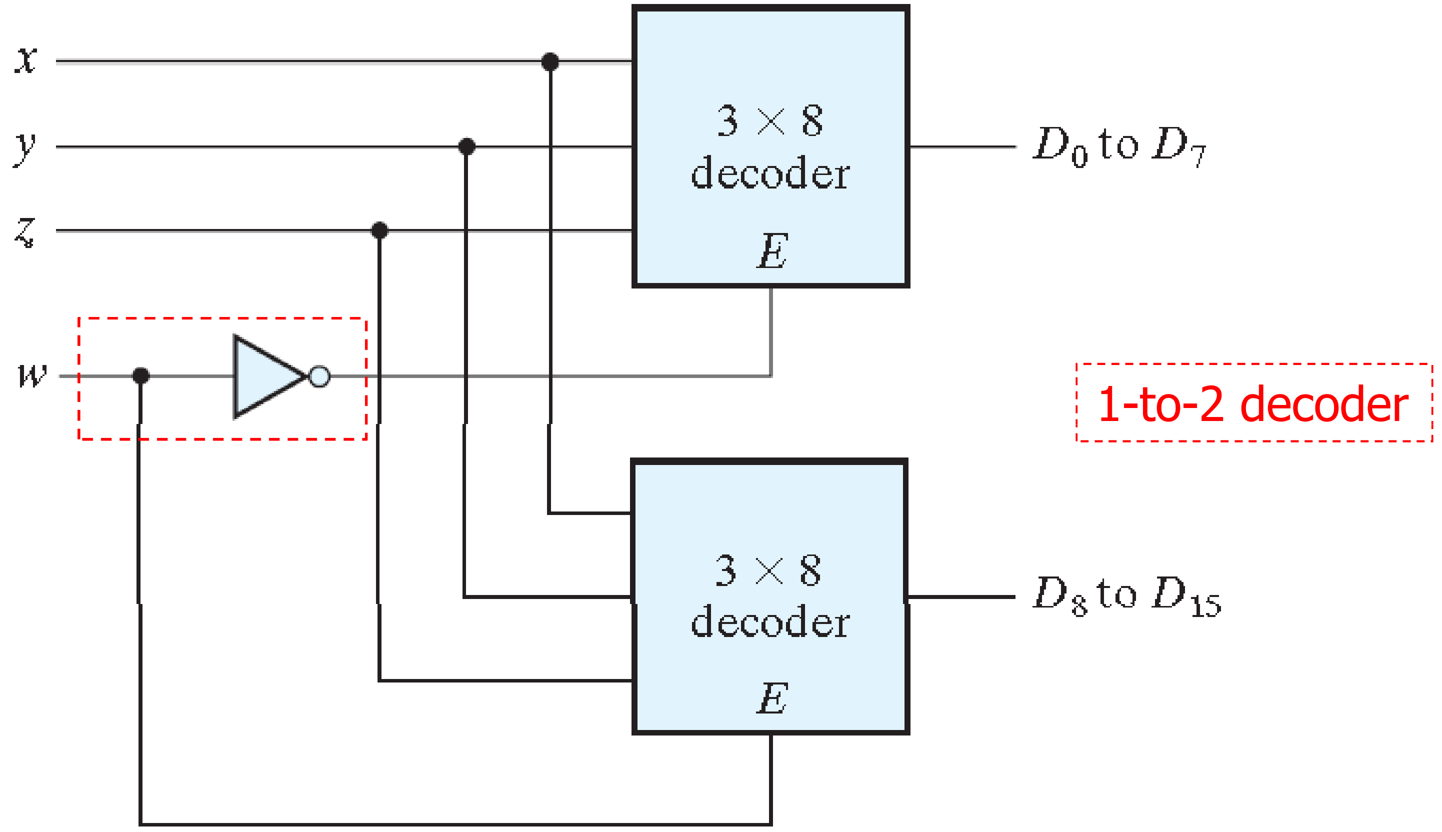
E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0



The demultiplexer described in textbook is questionable.
It will be talked later.

Decoder Expansion

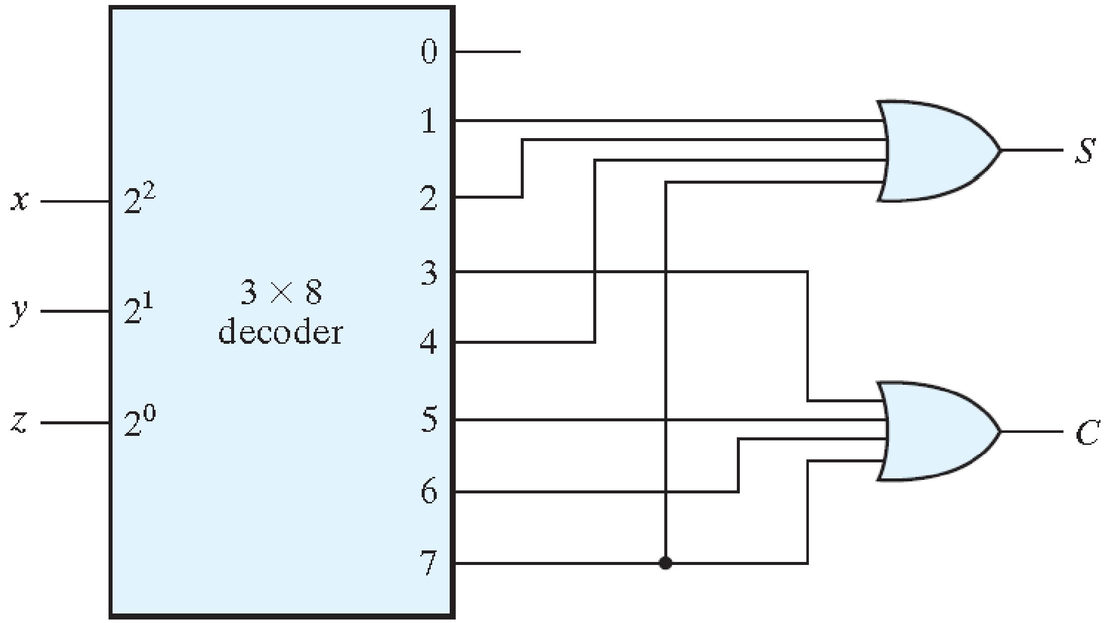
- Expand two 3-to-8 decoder to a 4-to-16 decoder



- How about a 5-to-32 decoder?

Universal Combinational Logic Implementation

- A decoder provides the 2^n minterms of n input variables.
- A decoder and an external OR gate can implement any Boolean function of n input variables in sum-of-minterm form.
- For example, see Table 4.4, a full-adder has its sum $S(x,y,z) = \Sigma(1,2,4,7)$ and carry $C(x,y,z) = \Sigma(3,5,6,7)$.



- Two possible approaches using decoder
  - OR(minterms of F): k inputs
  - NOR(minterms of F'): $2^n - k$ inputs

- The inverse function of a decoder
- 2^n (or fewer) input lines and n output lines
- The output lines generate the binary code corresponding to the input value.
- Example:

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

$z = D_1 + D_3 + D_5 + D_7$ can be implemented with OR gates

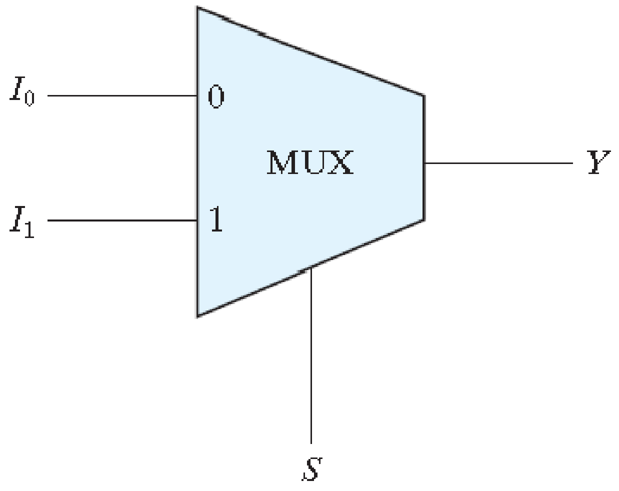
$y = D_2 + D_3 + D_6 + D_7$

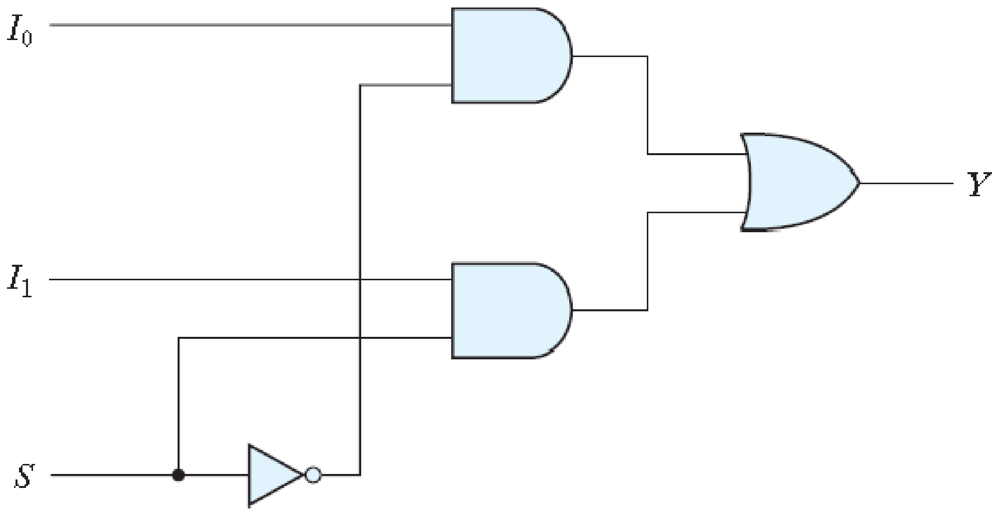
$x = D_4 + D_5 + D_6 + D_7$

10. Multiplexers

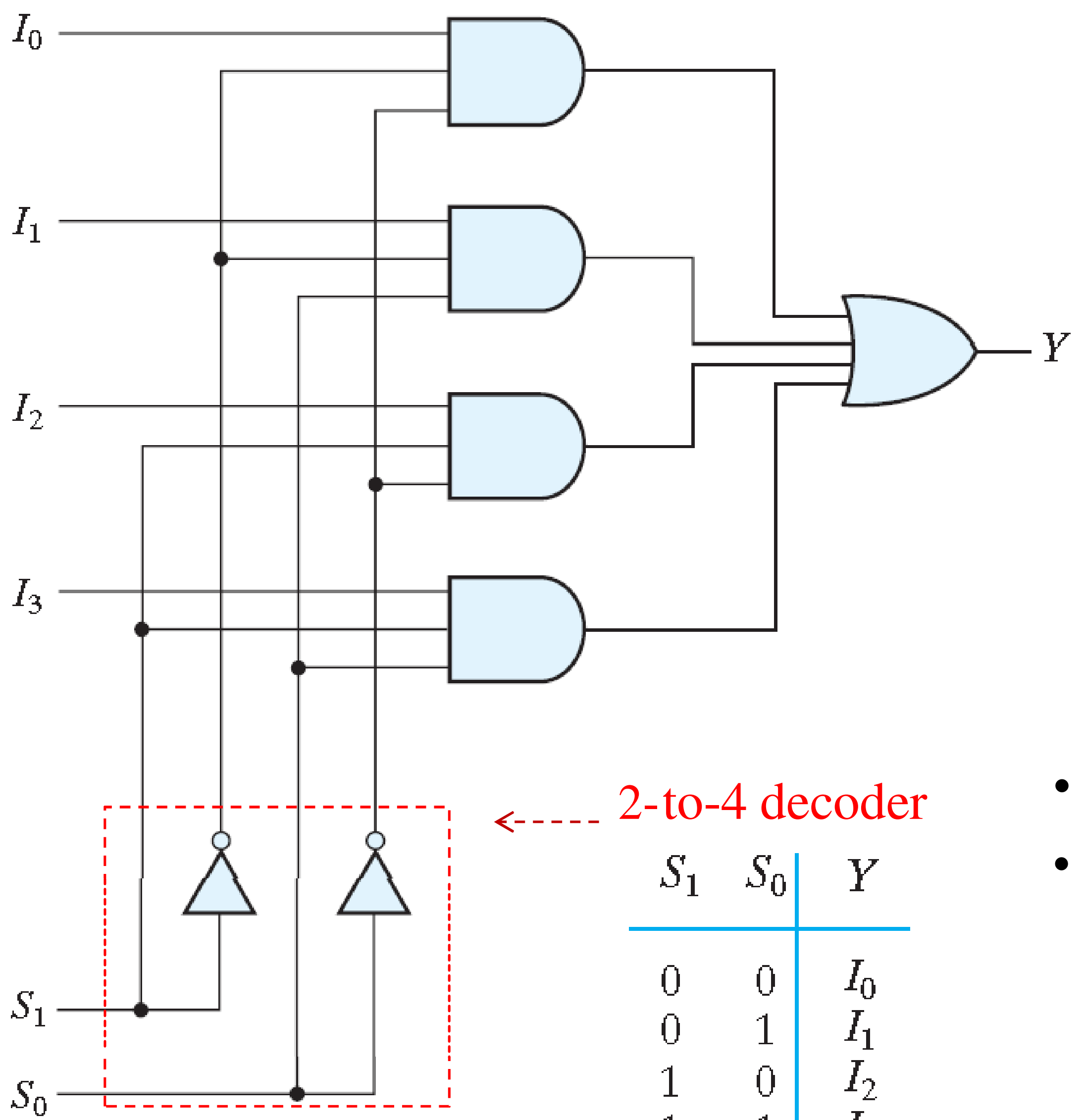
- Select from one of many inputs and directs it to a single output, controlled by a set of selection lines. A multiplexer is also called a *data selector*.
- Normally, there are 2^n inputs and n selection lines whose bit combinations determine which input is selected.
- Example: (two-to-one multiplexer)
  - $Y = I_0$ if $S = 0$, and $Y = I_1$ if $S = 1$.
  - $Y = S'I_0 + SY_1$

S	I_0	I_1	Y
0	0	X	0
0	1	X	1
1	X	0	0
1	X	1	1





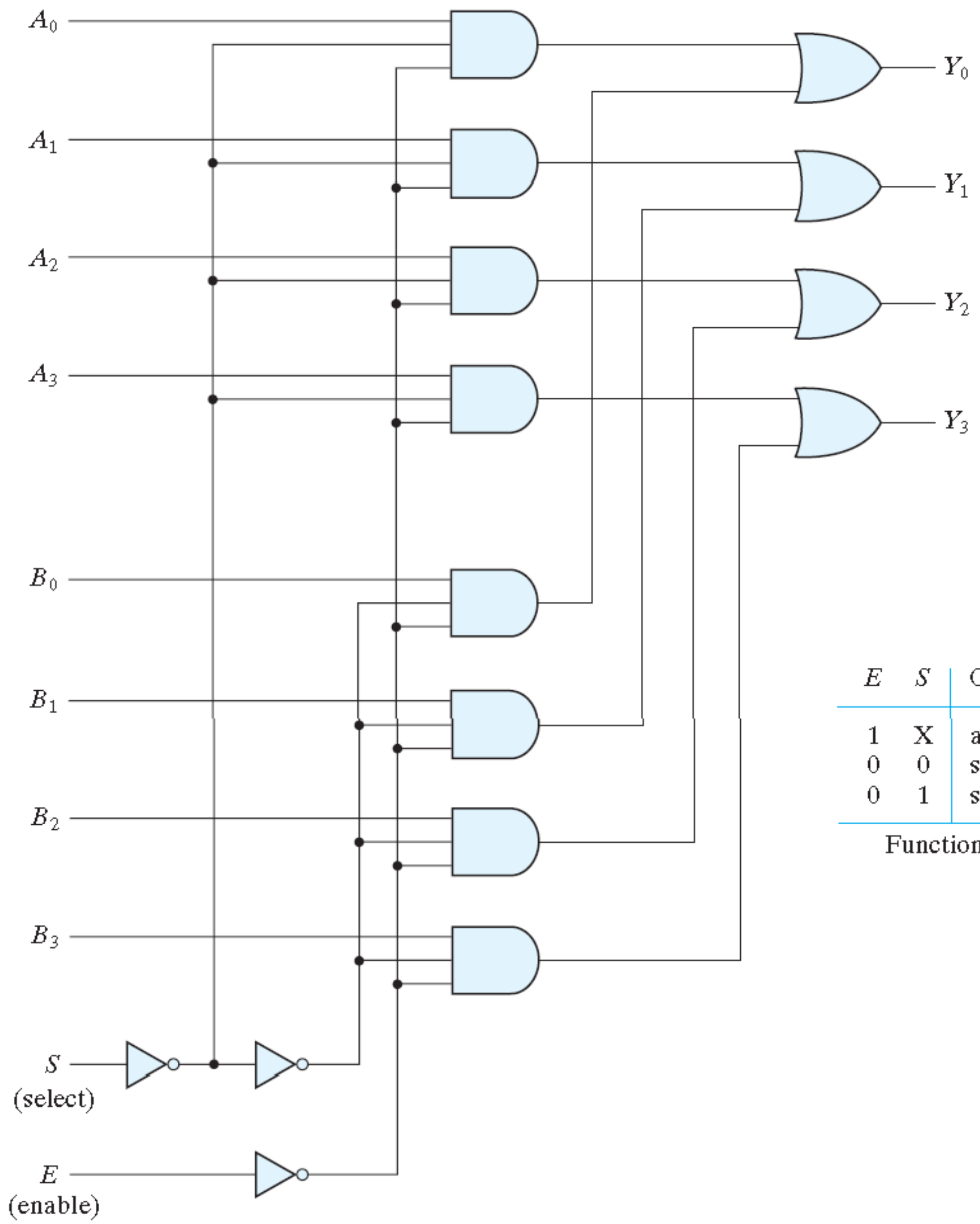
4-to-1 Multiplexer



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

- AND gates act as a switch
- OR gate will face a large fan-in if the input number grow

Quadruple 2-to-1 Multiplexer



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

Function table

$$A = A_3A_2A_1A_0$$
$$B = B_3B_2B_1B_0$$
$$Y = Y_3Y_2Y_1Y_0$$

$Y = A$ means

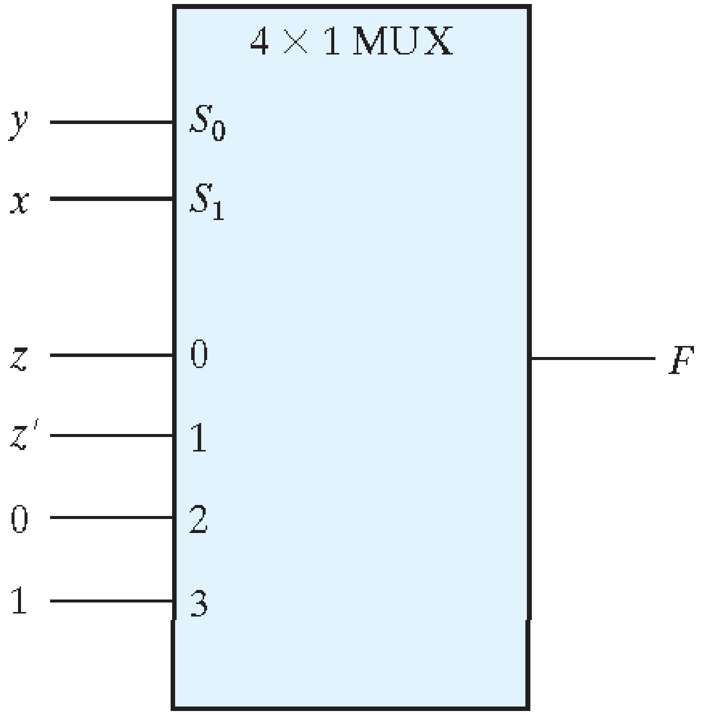
$$Y_3 = A_3,\ Y_2 = A_2$$
$$Y_1 = A_1,\ Y_0 = A_0$$

Boolean Function Implementation

- MUX has a structure composed of a decoder and an OR gate
- 2^n-to-1 MUX can implement any Boolean function of *n*+1 input variables
- *n* of these input variables are used as the selection lines
- The remaining single variable is used for the data inputs.
- If the single variable is denoted by *z*, each data input of the multiplexer will be *z*, *z'*, 1, or 0.
- Example: $F(x, y, z) = \Sigma(1, 2, 6, 7)$

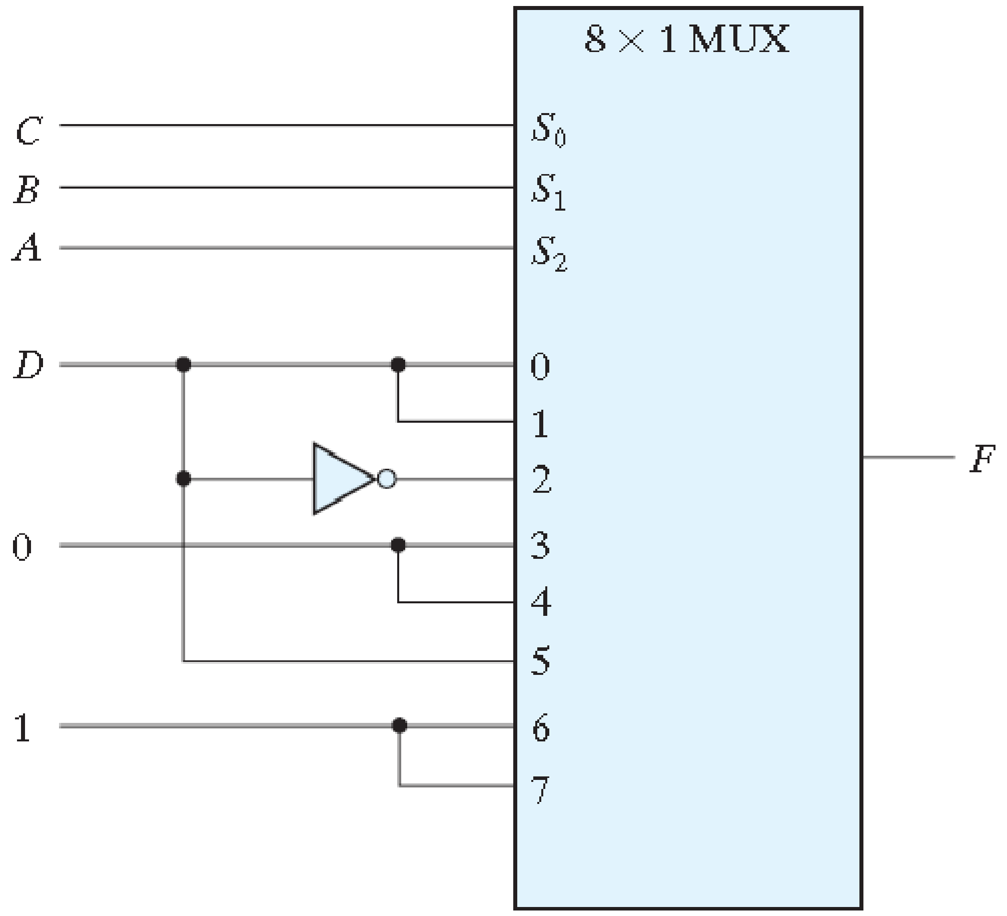
x	y	z	F	
0	0	0	0	$F = z$
0	0	1	1	
0	1	0	1	$F = z'$
0	1	1	0	
1	0	0	0	$F = 0$
1	0	1	0	
1	1	0	1	$F = 1$
1	1	1	1	

(a) Truth table



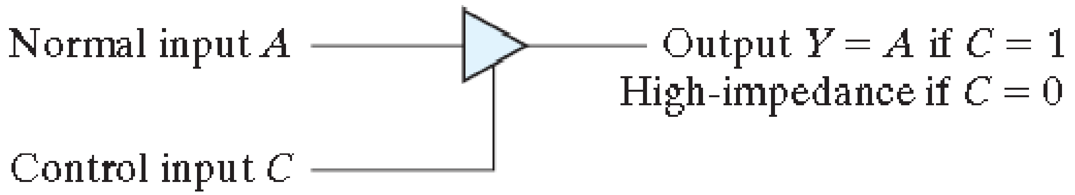
Another example: $F(A, B, C, D) = \Sigma\,(1, 3, 4, 11, 12, 13, 14, 15)$

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	



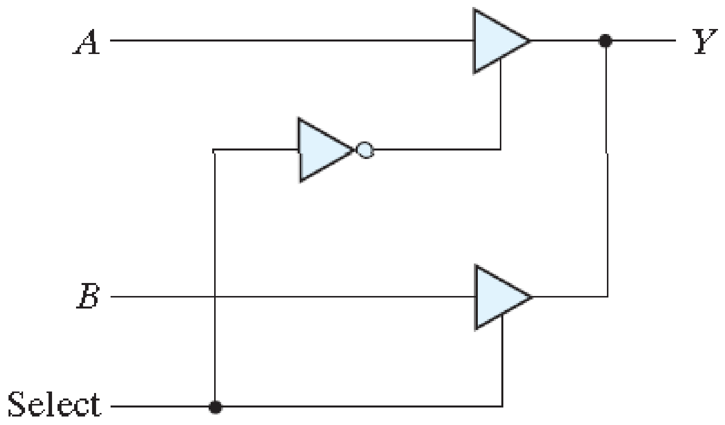
Three-state Gate

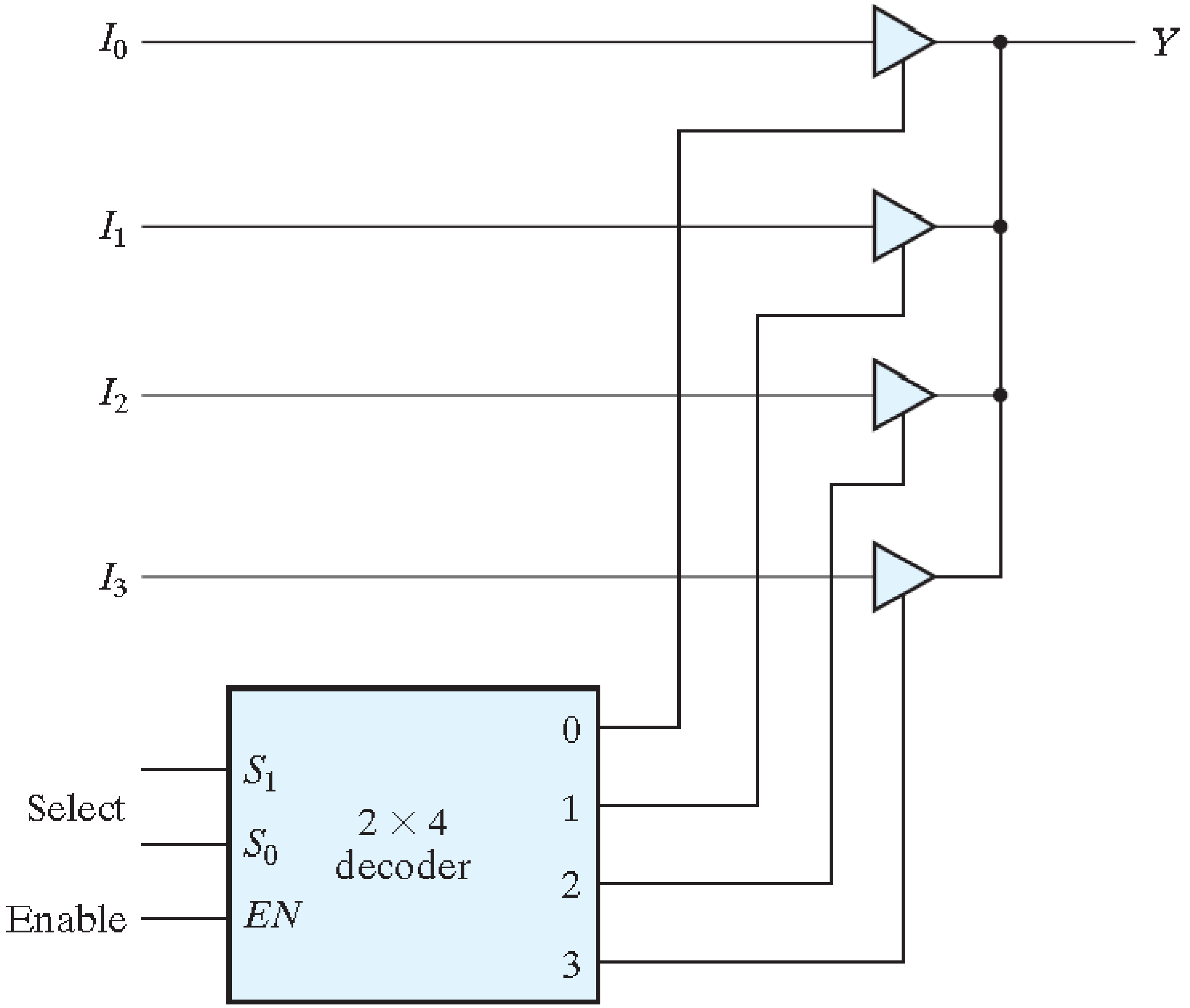
- The third state is a *high-impedance* state in which
  (1) the logic behaves like an open circuit, which means that the output appears to be disconnected,
  (2) the circuit has no logic significance, and
  (3) the circuit connected to the output of the three-state gate is not affected by the inputs to the gate.
- The most commonly used is the three-state buffer gate. (*a.k.a.* tri-state buffer)



- A multiplexer can be constructed with three-state gates.
- Example: 2-to-1 multiplexer

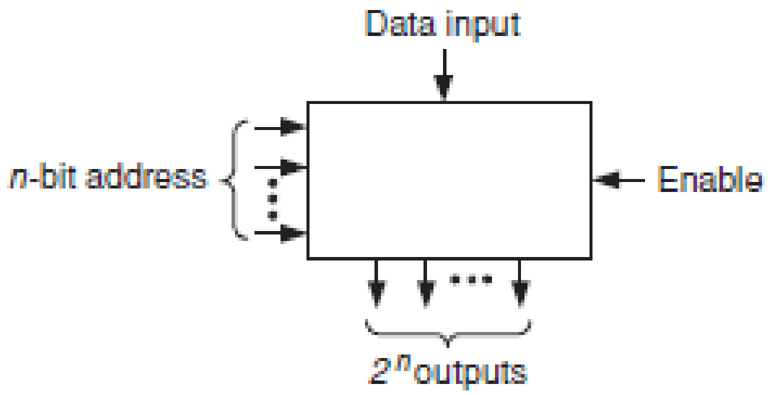
  $Y = A$ if Select = 0, $Y = B$ if Select = 1




I_0
I_1
I_2
I_3
Y
Select
S_1
S_0
Enable
EN
2×4 decoder
0
1
2
3

11.Demultiplexer

A decoder with one data input and *n* address inputs is called a *demultiplexer*. It directs the input data to any one of the 2*n* outputs, as specified by the *n*-bit input address. A block diagram for a demultiplexer is shown in Fig. When larger-size decoders are needed, they can usually be formed by interconnecting several smaller decoders with some additional logic.



Unit IV Synchronous Sequential Logic

1. SEQUENTIAL CIRCUITS

- Although every digital system is likely to have some **combinational circuits,** most systems encountered in practice also include storage elements which require that the system can be described in terms of sequential logic.
- **Sequential Logic** consists of combinational circuit to which storage elements are connected to form a **feedback path.**
- Storage elements are devices capable of storing binary information. This information that is stored at any time defines the state of the sequential circuit at that time.
- The sequential circuit receives binary valued inputs from external sources, this together with the present state of the storage elements determines the binary value of the outputs.
- These **external inputs** also determine the condition for changing the state in the storage elements.

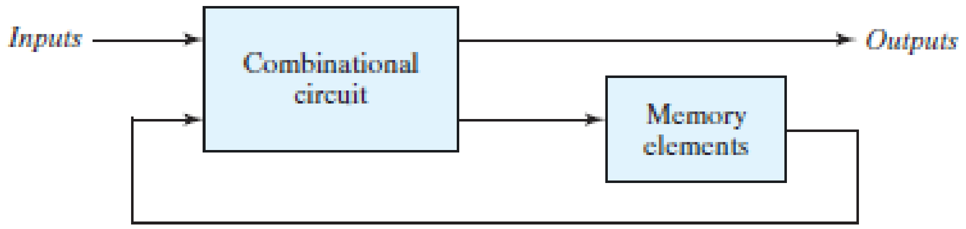


Figure 1. Block diagram of sequential circuit

- The block diagram demonstrates that the **outputs in a sequential circuit** are a function not only of the inputs, but also of the present state of the storage elements.

- The next state of the storage elements is also a function of external inputs and the present state. Thus, a sequential circuit is specified by a **time sequence of inputs, outputs, and internal states.**
- There are two main types of sequential circuits, and their classification is a function of the timing of their signals.

  - **Asynchronous sequential circuit:** A system whose behavior depends upon the input signals at any instant of time and the order in which the inputs change, The storage elements commonly used in asynchronous sequential circuits are **time-delay devices.**

  - **Synchronous sequential circuit:** A system whose behavior can be defined from the knowledge of its signals at discrete instants of time. They use a clock pulse for controlling the output.

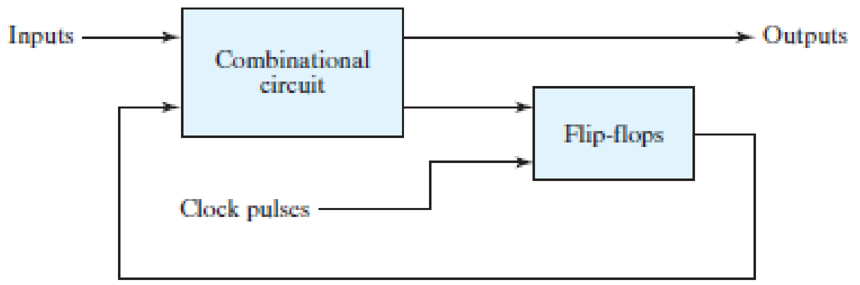
- Synchronous sequential circuit employs signals that affect the storage elements at only **discrete instants of time.** Synchronization is achieved by a timing device called a 'a clock generator', which provides a clock signal having the form of a periodic train of 'clock pulses'.
- The clock pulses are **distributed throughout the system** in such a way that storage elements are affected only with the arrival of each pulse.
- The clock pulses determine when computational activity will occur within the circuit, and other signals **(external inputs and otherwise)** determine what changes will take place affecting the storage elements and the outputs.
- **Example,** a circuit that is to add and store two binary numbers would compute their sum from the values of the numbers and store the sum at the occurrence of a clock pulse.
- Synchronous sequential circuits that use **clock pulses** to control storage elements are called clocked sequential circuits and are the type most frequently encountered in practice.
- They are called **synchronous circuits** because the activity within the circuit and the resulting updating of stored values is synchronized to the occurrence of clock pulses.

Advantage of Synchronous circuits:

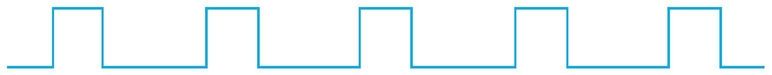
- The design of synchronous circuits is feasible because they seldom manifest **instability problems**
- Their timing is easily broken down into **independent discrete steps,** each of which can be considered separately.

Block diagram - Synchronous clocked sequential circuit

- The storage elements (memory) used in clocked sequential circuits are called **flip-flops.** A flip-flop is a binary storage device capable of storing one bit of information. In a stable state, the output of a flip-flop is either 0 or 1.
- The outputs are formed by a **combinational logic** function of the inputs to the circuit or the values stored in the flip-flops (or both).
- The value that is stored in a flip-flop when the clock pulse occurs is also determined by the inputs to the circuit or the values presently stored in the flip-flop or both.
- The new value is stored **(i.e the flip-flop is updated)** when a pulse of the clock signal occurs. Prior to the occurrence of the clock pulse the combinational logic forming the next value of the flop-flop must have reached a stable value.
- The transition from one state to the next occurs only at predetermined intervals dictated by the clock pulse - value of the clock signals changes from **0 to 1 or 1 to 0.**



(a) Block diagram


(b) Timing diagram of clock pulses

STORAGE ELEMENTS: LATCHES

- A storage element in a digital circuit can maintain a binary state **indefinitely** (as long as power is delivered to the circuit), until directed by an input signal to switch states.
- The major differences among various types of storage elements are in the number of inputs they possess and in the manner in which the inputs affect the binary state.
- Storage elements that operate at **signal levels** are referred to as latches; Those controlled by **clock transitions** are called flipflops.

- Latches are **level sensitive** and Flipflops are **edge sensitive.**
- Although latches are useful for storing binary information and for the design of **asynchronous sequential circuits** but they are not practical for use in **synchronous sequential circuits.**
- They are however the **building blocks of flip-flops**

SR Latch

- The SR latch is a circuit with two cross-coupled NOR gates or two **cross-coupled NAND gates** and two inputs labeled S for set and R for reset.
- The latch has two useful states:
    - When output **Q = 1 and Q' = 0** the latch is said to be in the set state
    - When **Q = 0 and Q' = 1** it is in the reset state
- **Outputs Q and Q'** are normally the complement of each other. However. when both inputs are equal to 1 at the same time, a condition in which both outputs are equal to 0 (rather than be mutually complementary) occurs.
- If both inputs are then **switched to 0** simultaneously, the device will enter an unpredictable or undefined state or a metastable state. Consequently, in practical applications setting both inputs to 1 is forbidden .
- Under normal conditions both inputs of the latch **remain at 0** unless the state has to be changed

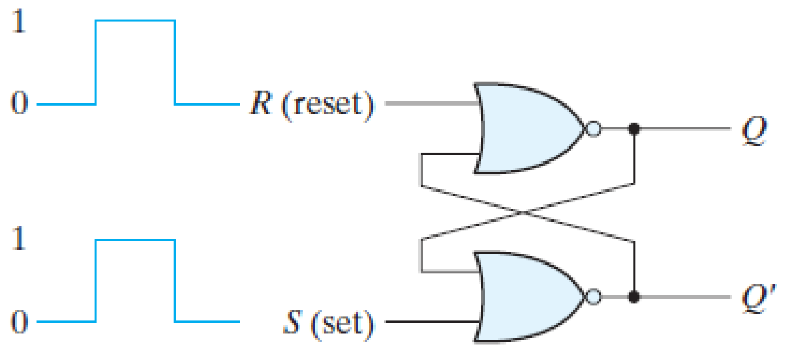
SR Latch using Cross coupled NOR Gates

- Both inputs of the **latch remain at 0** unless the state has to be changed. The application of a momentary 1 to the S input causes the latch to go to the set state.
- The S input must **go back to 0** before any other changes take place , in order to avoid the occurrence of an undefined next state that results from the forbidden input condition.
- Now, two input conditions cause the circuit to be in the set state.

- The first condition **(S = 1, R = 0)** is the action that must be taken by input S to bring the circuit to the set state. Removing the active input from S leaves the circuit in the same state.

- After both **inputs return to 0,** it is then possible to shift to the reset state by momentary applying a 1 to the R input. The 1 can then be removed from R, whereupon the circuit remains in the reset state.

- Thus, when both **inputs S and R are equal to 0,** the latch can be in either the set or the reset state. depending on which input was most recently a 1.
- If a 1 is applied to both the S and R inputs of the latch both outputs **go to 0.** This action produces an undefined next state because the state that results from the input transitions depends on the order in which they return to 0.
- It also violates the **requirement** that outputs be the complement of each other. In normal operation this condition is avoided by making sure that 1' s are not applied to both inputs simultaneously.



(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)

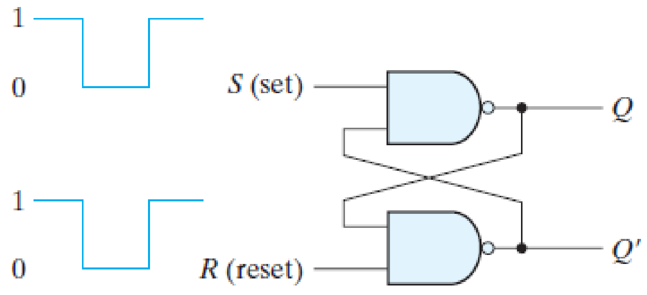
(b) Function table

Figure 2. SR latch with NOR gates

SR latch with two cross-coupled NAND gates

- It operates with both **inputs normally at 1** unless the state of the latch has to be changed

- The application of 0 to the S input causes output Q to **go to 1** putting, the latch in the Set state. When the S input goes back to 1 the circuit remains in the set state.
- After both inputs **go back to 1** we are allowed to change the state of tbe latch by placing a 0 in the R input.
- This action causes the circuit to go to the reset state and stay there even after both inputs **return to 1.**
- The condition that is **forbidden** for the NAND latch is both inputs being equal to 0 at the same time, an input combination that should be avoided.
- In comparing the NAND with the NOR latch, note that the input signals for the NAND require the complement of those values used for the NOR latch.
- Hence a NAND latch is also called a **S'R' latch.**



(a) Logic diagram

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

(b) Function table

Figure 3. SR latch with NAND gates

- In an SR Latch an **additional input called control input** can be provided to decide when the state of the latch can be changed.
- It consists of the basic SR latch and two additional NAND gates

SR Latch with control inputs

- The **control input En** acts as a enable for the other two inputs.
- The **output of the NAND gates** stay at logic 1 level as long as the enable signal remains at 0.
- When the enable signal goes to 1, the **S and R are allowed affect** the latch.

- The set state is reached when **S = 1, R = 0 and En = 1**
- The reset state is reached when **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.
- An indeterminate condition occurs when **all three inputs are equal to 1.** This condition places 0's on both inputs of the basic SR latch, which puts it in the **undefined state.**
- When the enable input goes back to 0, one cannot **conclusively** determine the next state because it depends on whether the S or R input goes to 0 first.
- This indeterminate condition makes this circuit difficult to manage and it is **seldom used in practice.**

Flip-Flops

- A flip-flop in digital electronics is a circuit with two stable states that can be used to store binary data. The stored data can be changed by applying varying inputs. Flip-flops and latches are fundamental building blocks of digital electronics systems used in computers, communications, and many other types of systems. Both are used as data storage elements.
- It is the basic storage element in sequential logic. But first, let’s clarify the difference between a latch and flip-flops.

Types

There are basically 4 types of flip-flops:

- SR Flip-Flop
- JK Flip-Flop
- D Flip-Flop
- T Flip-Flop

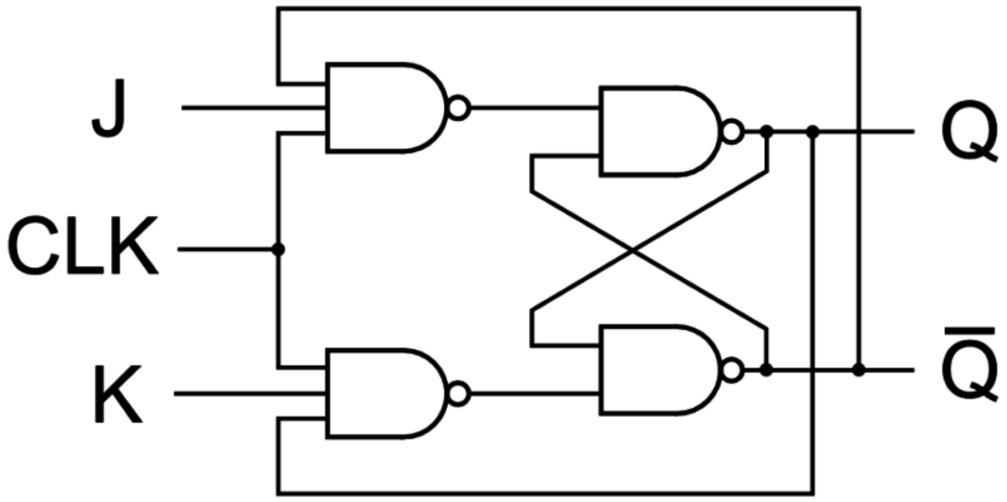
SR Flip Flop

- This is the most common flip-flop among all. This simple flip-flop circuit has a set input (S) and a reset input (R). In this system, when you Set "S" as active, the output "Q" would be high, and "Q'" would be low. Once the outputs are established, the wiring of the circuit is maintained until "S" or "R" go high, or power is turned off.
- As shown above, it is the simplest and easiest to understand. The two outputs, as shown above, are the inverse of each other. The **truth table of SR Flip-Flop** is highlighted below.
-

S	**R**	**Q**	**Q′**
0	0	0	1
0	1	0	1
1	0	1	0
1	1	∞	∞

JK Flip-Flop

- Due to the undefined state in the SR flip-flops, another flip-flop is required in electronics. The JK flip-flop is an improvement on the SR flip-flop where S=R=1 is not a problem.



•

Figure 4• JK Flip Flop Circuit

- The input condition of J=K=1 gives an output inverting the output state. However, the outputs are the same when one tests the circuit practically.
- In simple words, If J and K data input are different (i.e. high and low), then the output Q takes the value of J at the next clock edge. If J and K are both low, then no change occurs. If J and K are both high at the clock edge, then the output will toggle from one state to the other. JK Flip-Flops can function as Set or Reset Flip-flops.

Truth Table:

J	**K**	**Q**	**Q’**
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1
0	0	1	1
0	1	1	0
1	0	1	1
1	1	1	0

D Flip-Flop

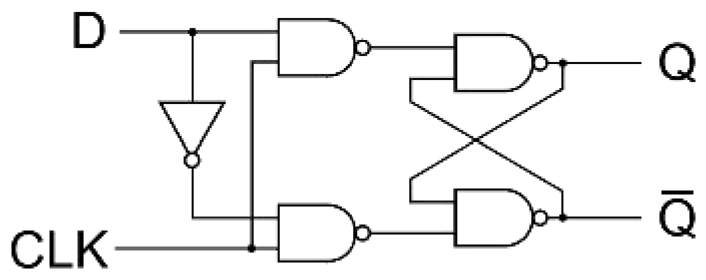
- D flip-flop is a better alternative that is very popular with digital electronics. They are commonly used for counters and shift registers and input synchronization.
- 

Figure 5. D Flip-Flop

- In the D flip-flops, the output can only be changed at the clock edge, and if the input changes at other times, the output will be unaffected.
  Truth Table:

Clock	D	Q	Q’
↓ » 0	0	0	1
↑ » 1	0	0	1
↓ » 0	1	0	1
↑ » 1	1	1	0

-
- The change of state of the output is dependent on the rising edge of the clock. The output (Q) is the same as the input and can only change at the rising edge of the clock.

T Flip-Flop

- A T flip-flop is like a JK flip-flop. These are basically single-input versions of JK flip-flops. This modified form of the JK is obtained by connecting inputs J and K together. It has only one input along with the clock input.

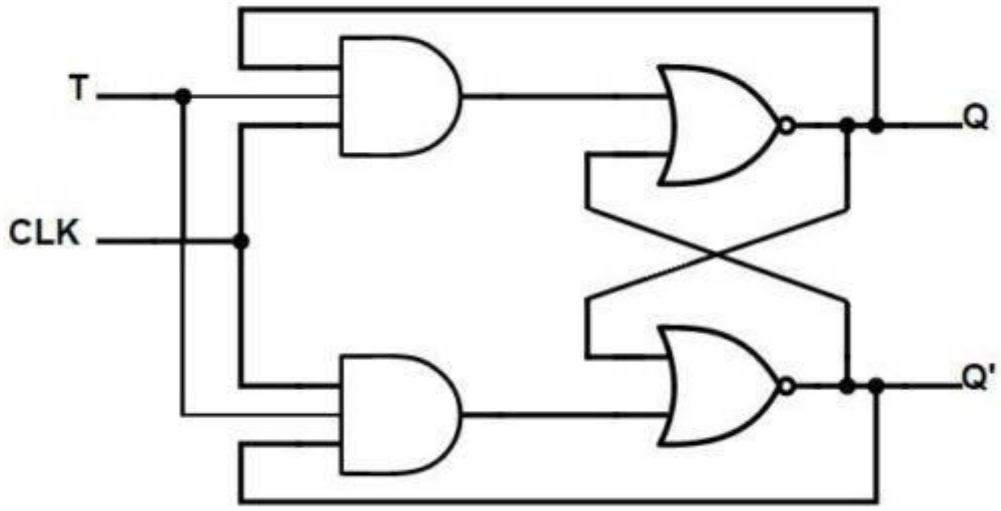


Figure 6. T flip flop

- These flip-flops are called T flip-flops because of their ability to complement their state i.e. Toggle, hence they are named Toggle flip-flops.

Truth Table:

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

ANALYSIS OF CLOCKED SEQUENTIAL CIRCUITS

- Analysis describes what a given circuit will do under certain operating conditions. The behavior of a clocked sequential circuit is determined from the inputs, the outputs, and the state of its flip-flops.
- The outputs and the next state are both a function of the inputs and the present state. The analysis of a sequential circuit consists of obtaining a table or a diagram for the time sequence of inputs, outputs, and internal states. It is also possible to write Boolean expressions that describe the behavior of the sequential circuit. These expressions must include the necessary time sequence, either directly or indirectly. A logic diagram is recognized as a clocked sequential circuit if it includes flip-flops with clock inputs. The flip-flops may be of any type, and the logic diagram may or may not include combinational logic gates. In this section, we introduce an algebraic representation for specifying the next-state condition in terms of the present state and inputs. A state table and state diagram are then presented to describe the behavior of the sequential circuit.

- Another algebraic representation is introduced for specifying the logic diagram of sequential circuits. Examples are used to illustrate the various procedures.

State Equations

- The behavior of a clocked sequential circuit can be described algebraically by means of state equations. A state equation (also called a transition equation) specifies the next state as a function of the present state and inputs.

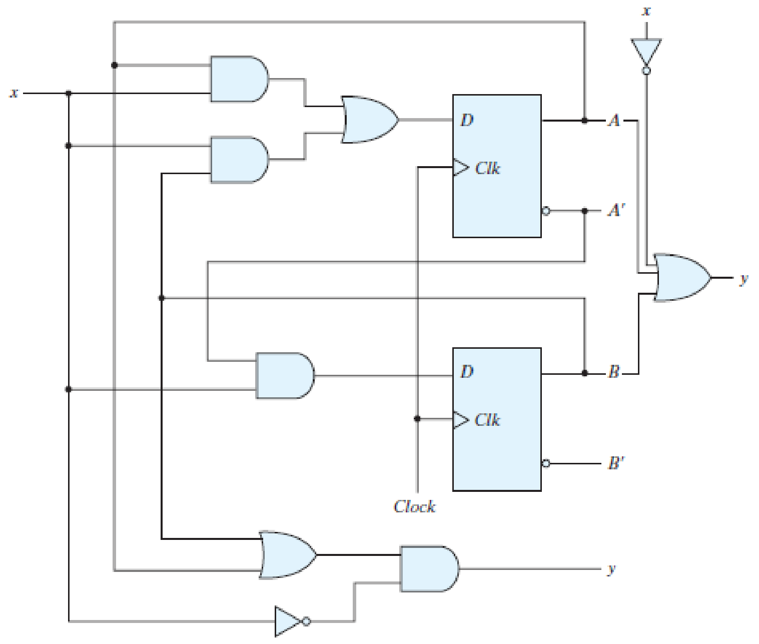


Figure 7. Example of sequential circuit

- The above circuit consists of two D flip-flops A and B, an input x and an output y . Since the D input of a flip-flop determines the value of the next state (i.e., the state reached after the clock transition), it is possible to write a set of state equations for the circuit:

$$A(t+1) = A(t)x(t) + B(t)x(t)$$
$$B(t+1) = A'(t)x(t)$$

- A state equation is an algebraic expression that specifies the condition for a flip-flop state transition, for convenience and can express the state equations in the more compact form.

$$A(t+1) - Ax + Bx$$
$$B(t+1) - A'x$$

By removing the symbol *(t)* for the present state, we obtain the output Boolean equation:

$$y = (A + B)x'$$

State Table

- The time sequence of inputs, outputs, and flip-flop states can be enumerated in a state table (sometimes called a transition table).

Present State		Input	Next State		Output
A	*B*	*x*	*A*	*B*	*y*
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

State Diagram

- The information available in a state table can be represented graphically in the form of a state diagram. In this type of diagram, a state is represented by a circle, and the (clock-triggered) transitions between states are indicated by directed lines connecting the circles.

Second Form of the State Table

Present State		Next State				Output	
		x = 0		x = 1		x = 0	x = 1
A	*B*	*A*	*B*	*A*	*B*	*y*	*y*
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0

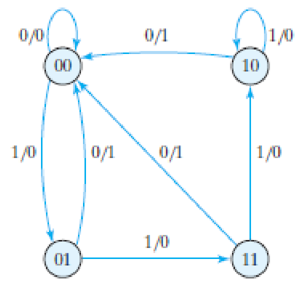


Figure 8. State diagram of the circuit

Registers

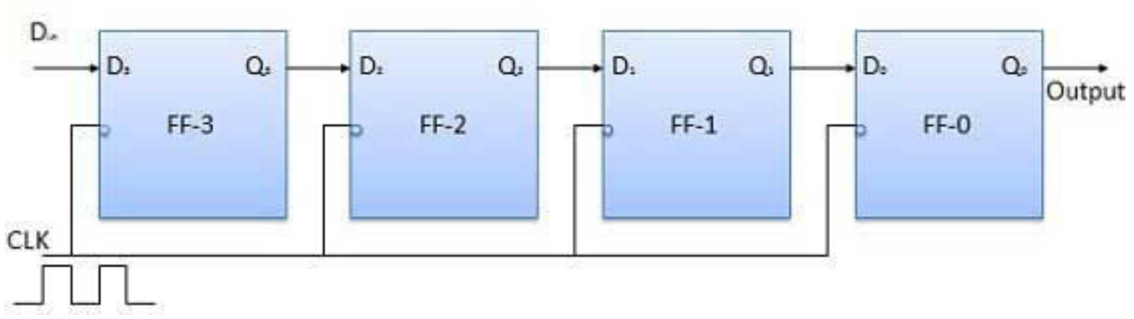
- Flip-flop is a 1 bit memory cell which can be used for storing the digital data. To increase the storage capacity in terms of number of bits, we have to use a group of flip-flop. Such a group of flip-flop is known as a Register. The n-bit register will consist of n number of flip-flop and it is capable of storing an n-bit word.
- The binary data in a register can be moved within the register from one flip-flop to another. The registers that allow such data transfers are called as shift registers. There are four mode of operations of a shift register.
- Serial Input Serial Output
- Serial Input Parallel Output
- Parallel Input Serial Output
- Parallel Input Parallel Output

Shift Registers

Serial Input Serial Output

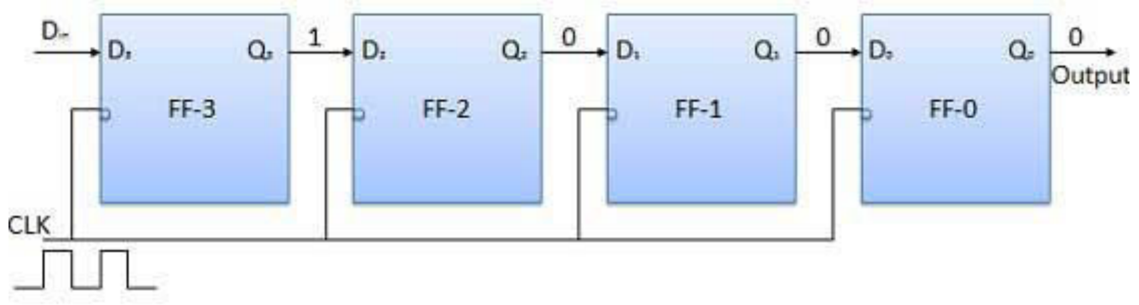
- Let all the flip-flop be initially in the reset condition i.e. Q3 = Q2 = Q1 = Q0 = 0. If an entry of a four bit binary number 1 1 1 1 is made into the register, this number should be applied to Din bit with the LSB bit applied first. The D input of FF-3 i.e. D3 is connected to serial data input Din. Output of FF-3 i.e. Q3 is connected to the input of the next flip-flop i.e. D2 and so on.

Block Diagram

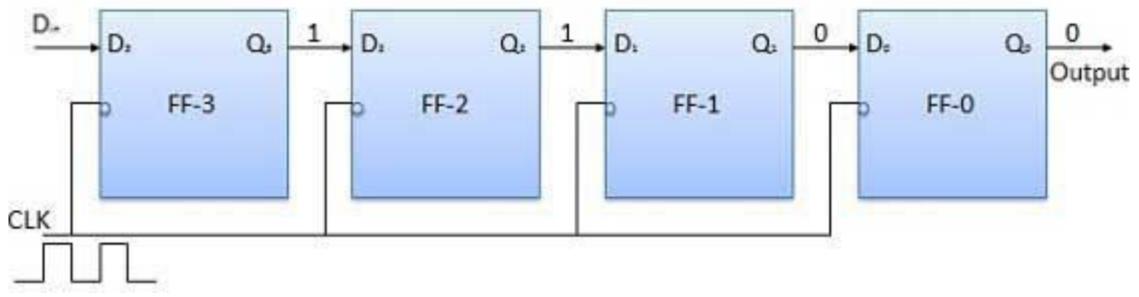


Operation

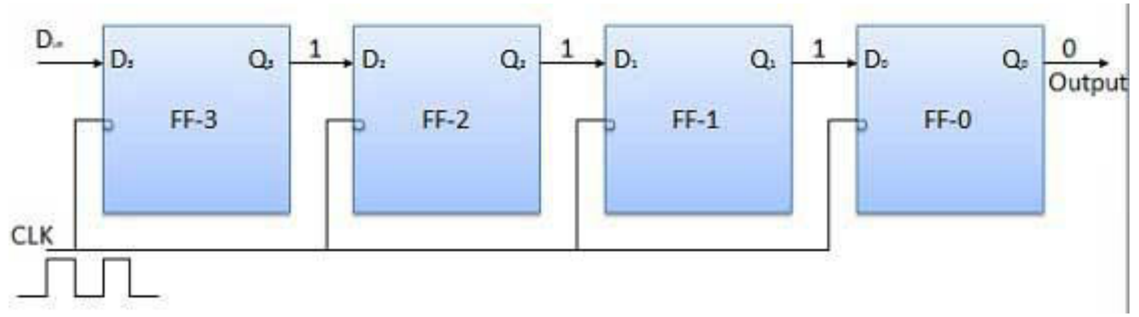
- Before application of clock signal, let Q3 Q2 Q1 Q0 = 0000 and apply LSB bit of the number to be entered to Din. So Din = D3 = 1. Apply the clock. On the first falling edge of clock, the FF-3 is set, and stored word in the register is Q3 Q2 Q1 Q0 = 1000.



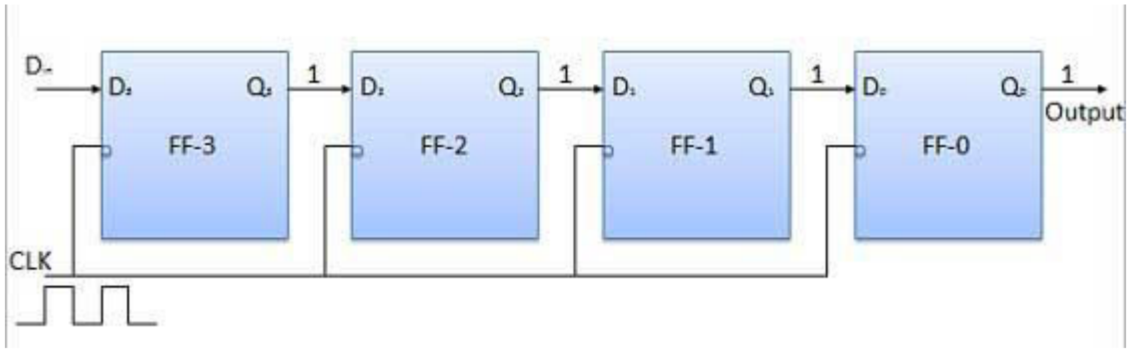
- Apply the next bit to Din. So Din = 1. As soon as the next negative edge of the clock hits, FF-2 will set and the stored word change to Q3 Q2 Q1 Q0 = 1100.



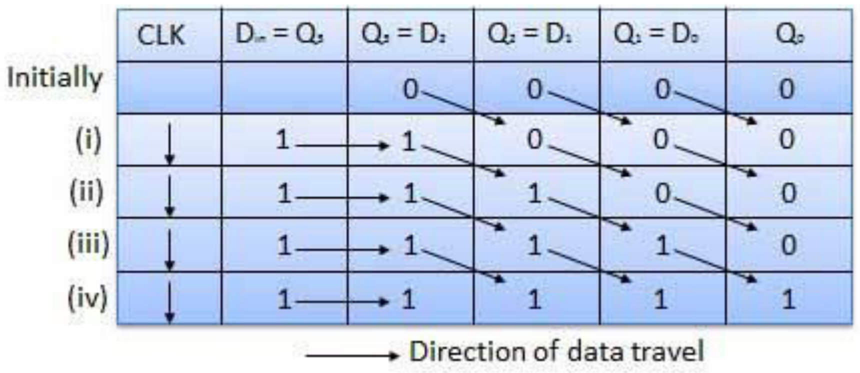
- Apply the next bit to be stored i.e. 1 to Din. Apply the clock pulse. As soon as the third negative clock edge hits, FF-1 will be set and output will be modified to Q3 Q2 Q1 Q0 = 1110.



- Similarly with Din = 1 and with the fourth negative clock edge arriving, the stored word in the register is Q3 Q2 Q1 Q0 = 1111.

- 

- Truth Table

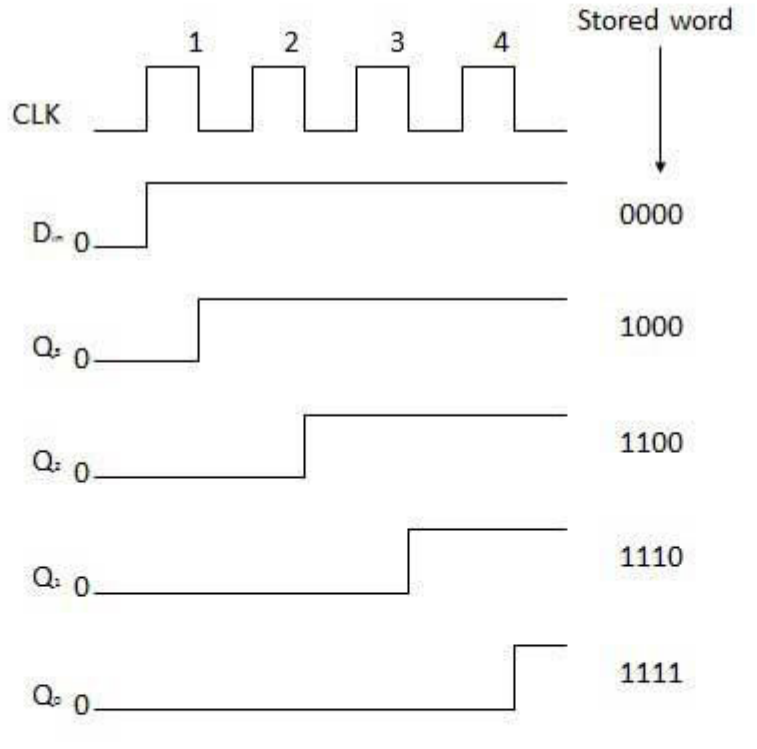


	CLK	$D_{in} = Q_3$	$Q_3 = D_2$	$Q_2 = D_1$	$Q_1 = D_0$	Q_0
Initially			0	0	0	0
(i)	↓	1	1	0	0	0
(ii)	↓	1	1	1	0	0
(iii)	↓	1	1	1	1	0
(iv)	↓	1	1	1	1	1

$\longrightarrow$ Direction of data travel

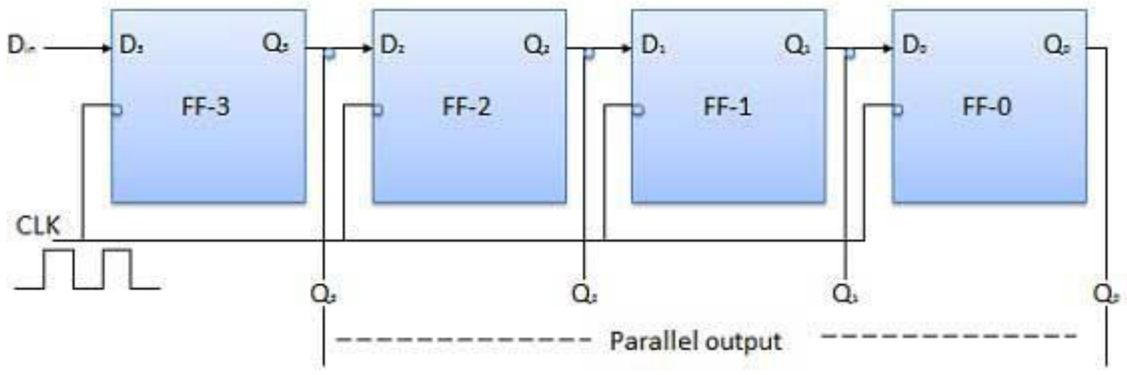
-

- Waveforms



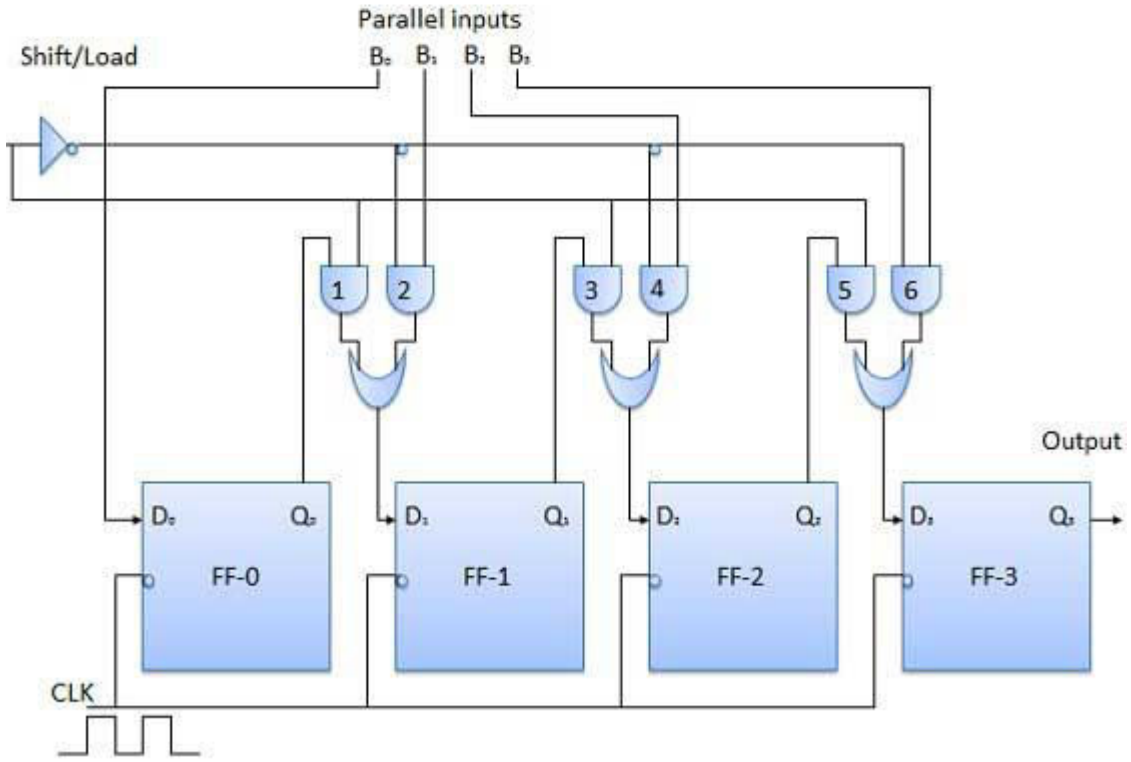
Serial Input Parallel Output

- In such types of operations, the data is entered serially and taken out in parallel fashion. Data is loaded bit by bit. The outputs are disabled as long as the data is loading. As soon as the data loading gets completed, all the flip-flops contain their required data, the outputs are enabled so that all the loaded data is made available over all the output lines at the same time.
- 4 clock cycles are required to load a four bit word. Hence the speed of operation of SIPO mode is same as that of SISO mode.



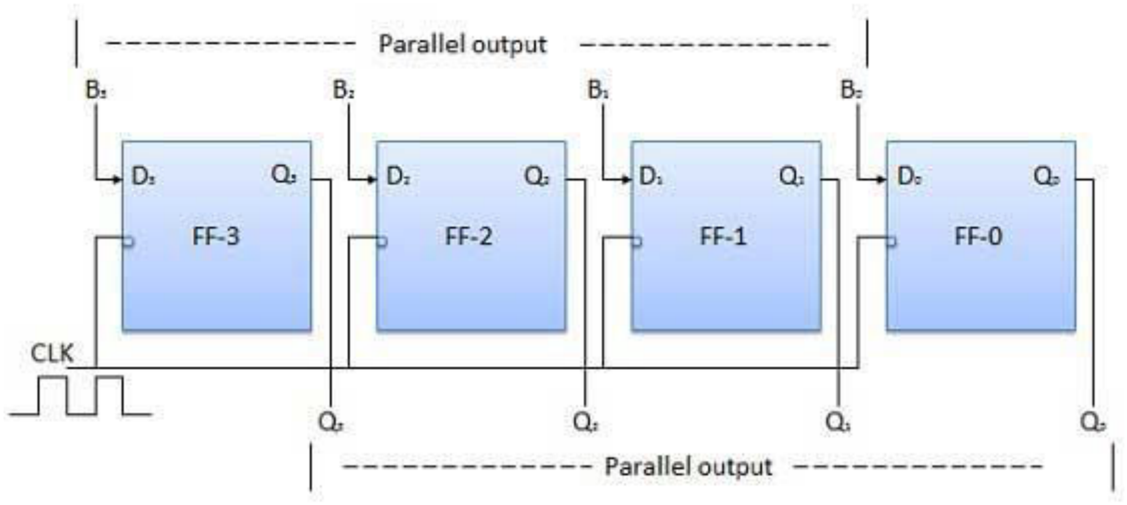
Parallel Input Serial Output (PISO)

- Data bits are entered in parallel fashion.
- The circuit shown below is a four bit parallel input serial output register.
- Output of previous Flip Flop is connected to the input of the next one via a combinational circuit.
- The binary input word B0, B1, B2, B3 is applied though the same combinational circuit.
- There are two modes in which this circuit can work namely - shift mode or load mode.
- Load mode
- When the shift/load bar line is low (0), the AND gate 2, 4 and 6 become active they will pass B1, B2, B3 bits to the corresponding flip-flops. On the low going edge of clock, the binary input B0, B1, B2, B3 will get loaded into the corresponding flip-flops. Thus parallel loading takes place.
- Shift mode
- When the shift/load bar line is low (1), the AND gate 2, 4 and 6 become inactive. Hence the parallel loading of the data becomes impossible. But the AND gate 1,3 and 5 become active. Therefore the shifting of data from left to right bit by bit on application of clock pulses. Thus the parallel in serial out operation takes place.



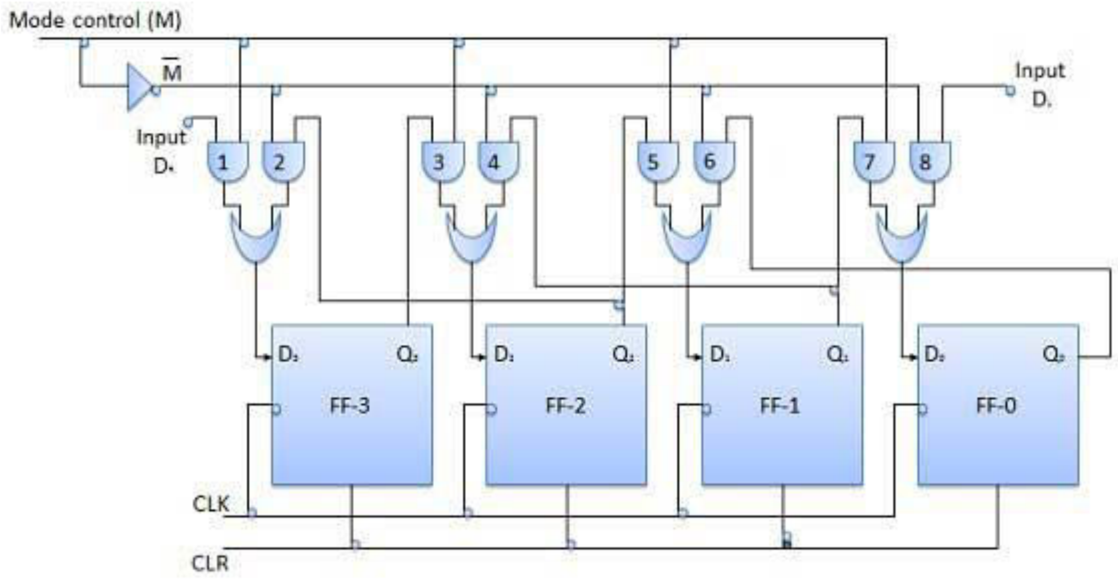
Parallel Input Parallel Output (PIPO)

- In this mode, the 4 bit binary input B0, B1, B2, B3 is applied to the data inputs D0, D1, D2, D3 respectively of the four flip-flops. As soon as a negative clock edge is applied, the input binary bits will be loaded into the flip-flops simultaneously. The loaded bits will appear simultaneously to the output side. Only clock pulse is essential to load all the bits.



Bidirectional Shift Register

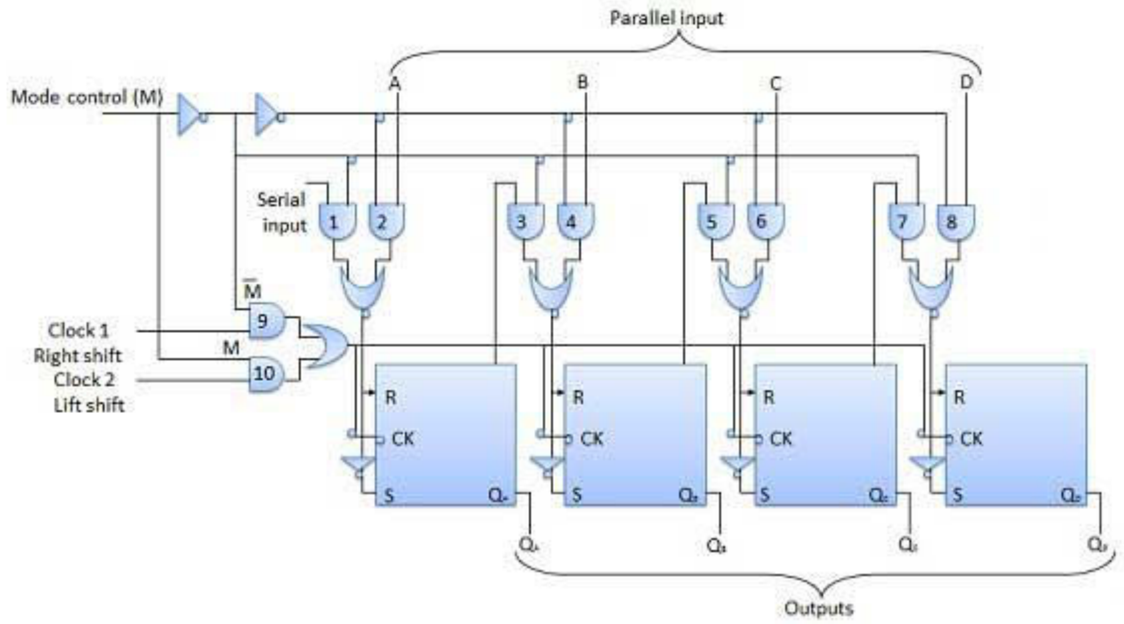
- If a binary number is shifted left by one position then it is equivalent to multiplying the original number by 2. Similarly if a binary number is shifted right by one position then it is equivalent to dividing the original number by 2.
- Hence if we want to use the shift register to multiply and divide the given binary number, then we should be able to move the data in either left or right direction.
- Such a register is called bi-directional register. A four bit bi-directional shift register is shown in fig.
- There are two serial inputs namely the serial right shift data input DR, and the serial left shift data input DL along with a mode select input (M).



S.N.	Condition	Operation
1	With M = 1 − Shift right operation	If M = 1, then the AND gates 1, 3, 5 and 7 are enabled whereas the remaining AND gates 2, 4, 6 and 8 will be disabled. The data at DR is shifted to right bit by bit from FF-3 to FF-0 on the application of clock pulses. Thus with M = 1 we get the serial right shift operation.
2	With M = 0 − Shift left operation	When the mode control M is connected to 0 then the AND gates 2, 4, 6 and 8 are enabled while 1, 3, 5 and 7 are disabled. The data at DL is shifted left bit by bit from FF-0 to FF-3 on the application of clock pulses. Thus with M = 0 we get the serial right shift operation.

Universal Shift Register

- A shift register which can shift the data in only one direction is called a uni-directional shift register. A shift register which can shift the data in both directions is called a bi-directional shift register. Applying the same logic, a shift register which can shift the data in both directions as well as load it parallely, is known as a universal shift register. The shift register is capable of performing the following operation −
- Parallel loading
- Left Shifting
- Right shifting
- The mode control input is connected to logic 1 for parallel loading operation whereas it is connected to 0 for serial shifting. With mode control pin connected to ground, the universal shift register acts as a bi-directional register. For serial left operation, the input is applied to the serial input which goes to AND gate-1 shown in figure. Whereas for the shift right operation, the serial input is applied to D input.
- Block Diagram



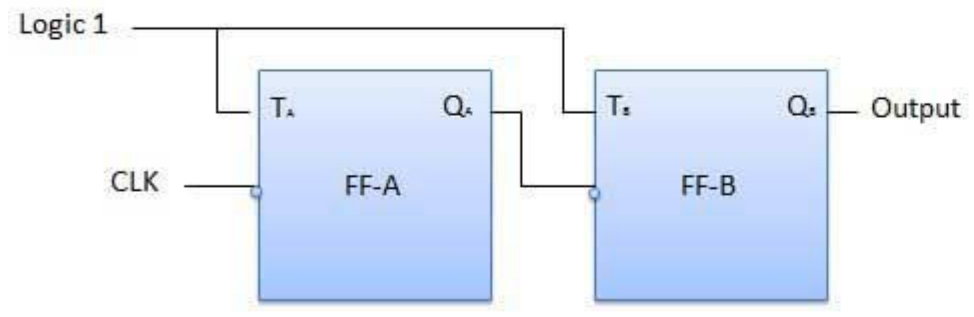
Counter is a sequential circuit. A digital circuit which is used for a counting pulses is known counter. Counter is the widest application of flip-flops. It is a group of flip-flops with a clock signal applied. Counters are of two types.

- Asynchronous or ripple counters.
- Synchronous counters.

Asynchronous or ripple counters

The logic diagram of a 2-bit ripple up counter is shown in figure. The toggle (T) flip-flop are being used. But we can use the JK flip-flop also with J and K connected permanently to logic 1. External clock is applied to the clock input of flip-flop A and Q_A output is applied to the clock input of the next flip-flop i.e. FF-B.

Logical Diagram



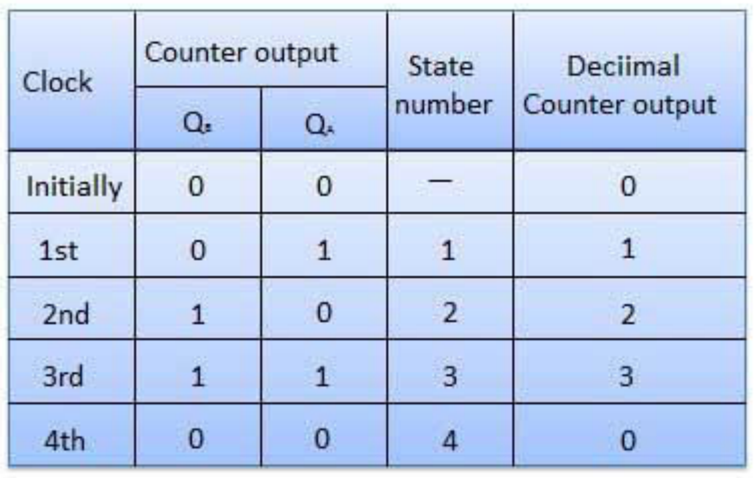
Operation

S.N.	Condition	Operation

1	**Initially let both the FFs be in the reset state**	Q_BQ_A = 00 initially
2	**After 1st negative clock edge**	As soon as the first negative clock edge is applied, FF-A will toggle and Q_A will be equal to 1. Q_A is connected to clock input of FF-B. Since Q_A has changed from 0 to 1, it is treated as the positive clock edge by FF-B. There is no change in Q_B because FF-B is a negative edge triggered FF. Q_BQ_A = 01 after the first clock pulse.
3	**After 2nd negative clock edge**	On the arrival of second negative clock edge, FF-A toggles again and Q_A = 0. The change in Q_A acts as a negative clock edge for FF-B. So it will also toggle, and Q_B will be 1. Q_BQ_A = 10 after the second clock pulse.
4	**After 3rd negative clock edge**	On the arrival of 3rd negative clock edge, FF-A toggles again and Q_A become 1 from 0. Since this is a positive going change, FF-B does not respond to it and remains inactive. So Q_B does not change and continues to be equal to 1. Q_BQ_A = 11 after the third clock pulse.
5	**After 4th negative clock edge**	On the arrival of 4th negative clock edge, FF-A toggles again and Q_A becomes 1 from 0. This negative change in Q_A acts as clock pulse for FF-B. Hence it toggles to change

		Q_B from 1 to 0. Q_BQ_A = 00 after the fourth clock pulse.

Truth Table



Clock	Counter output		State number	Deciimal Counter output
	Q_B	Q_A		
Initially	0	0	—	0
1st	0	1	1	1
2nd	1	0	2	2
3rd	1	1	3	3
4th	0	0	4	0

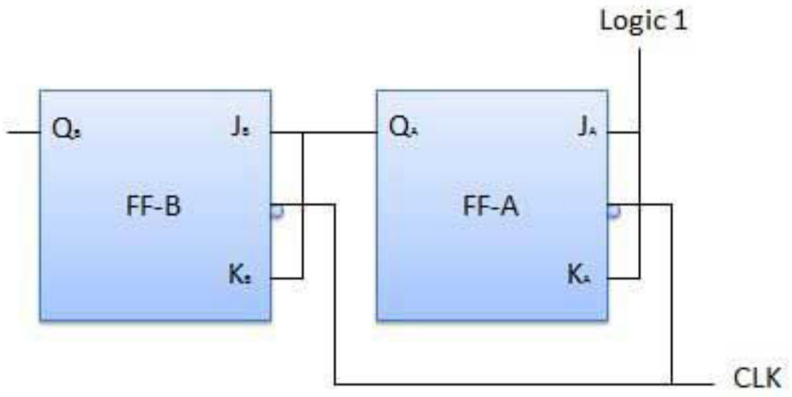
Synchronous counters

If the "clock" pulses are applied to all the flip-flops in a counter simultaneously, then such a counter is called as synchronous counter.

2-bit Synchronous up counter

The J_A and K_A inputs of FF-A are tied to logic 1. So FF-A will work as a toggle flip-flop. The J_B and K_B inputs are connected to Q_A.

Logical Diagram



Operation

S.N.	Condition	Operation
1	**Initially let both the FFs be in the reset state**	Q_BQ_A = 00 initially.
2	**After 1st negative clock edge**	As soon as the first negative clock edge is applied, FF-A will toggle and Q_A will

		change from 0 to 1. But at the instant of application of negative clock edge, Q_A , $J_B = K_B = 0$. Hence FF-B will not change its state. So Q_B will remain 0. Q_BQ_A = 01 after the first clock pulse.
3	**After 2nd negative clock edge**	On the arrival of second negative clock edge, FF-A toggles again and Q_A changes from 1 to 0. But at this instant Q_A was 1. So $J_B = K_B= 1$ and FF-B will toggle. Hence Q_B changes from 0 to 1. Q_BQ_A = 10 after the second clock pulse.
4	**After 3rd negative clock edge**	On application of the third falling clock edge, FF-A will toggle from 0 to 1 but there is no change of state for FF-B. Q_BQ_A = 11 after the third clock pulse.
5	**After 4th negative clock edge**	On application of the next clock pulse, Q_A will change from 1 to 0 as Q_B will also change from 1 to 0. Q_BQ_A = 00 after the fourth clock pulse.

Classification of counters

Depending on the way in which the counting progresses, the synchronous or asynchronous counters are classified as follows −

- Up counters
- Down counters

- Up/Down counters

UP/DOWN Counter

Up counter and down counter is combined together to obtain an UP/DOWN counter. A mode control (M) input is also provided to select either up or down mode. A combinational circuit is required to be designed and used between each pair of flip-flop in order to achieve the up/down operation.

- Type of up/down counters
- UP/DOWN ripple counters
- UP/DOWN synchronous counter

UP/DOWN Ripple Counters

In the UP/DOWN ripple counter all the FFs operate in the toggle mode. So either T flip-flops or JK flip-flops are to be used. The LSB flip-flop receives clock directly. But the clock to every other FF is obtained from (Q = Q bar) output of the previous FF.

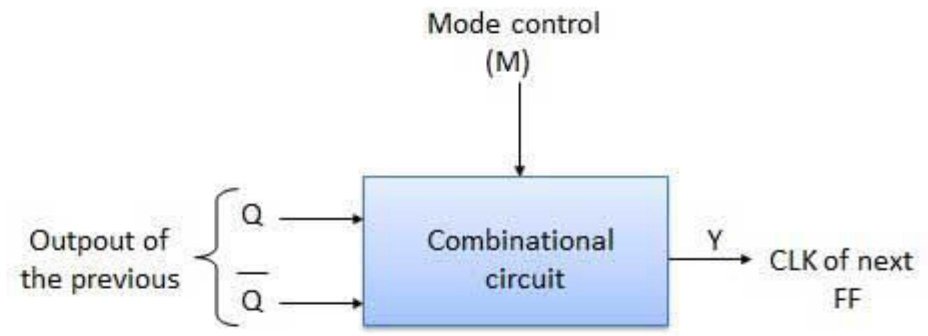
- **UP counting mode (M=0)** − The Q output of the preceding FF is connected to the clock of the next stage if up counting is to be achieved. For this mode, the mode select input M is at logic 0 (M=0).
- **DOWN counting mode (M=1)** − If M = 1, then the Q bar output of the preceding FF is connected to the next FF. This will operate the counter in the counting mode.

Example

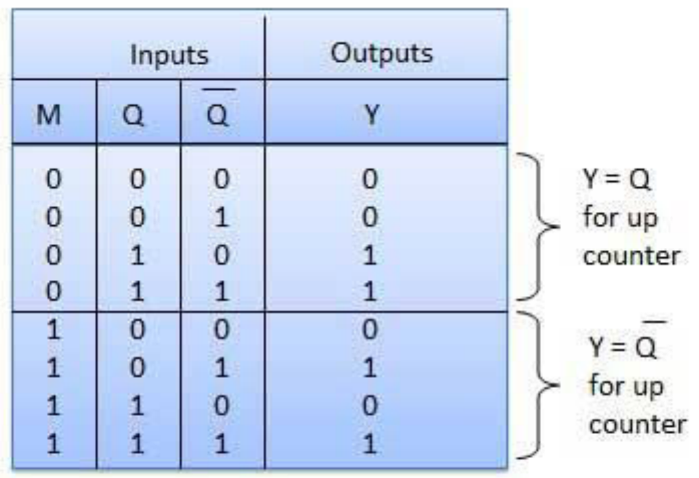
3-bit binary up/down ripple counter.

- 3-bit − hence three FFs are required.
- UP/DOWN − So a mode control input is essential.
- For a ripple up counter, the Q output of preceding FF is connected to the clock input of the next one.
- For a ripple up counter, the Q output of preceding FF is connected to the clock input of the next one.
- For a ripple down counter, the Q bar output of preceding FF is connected to the clock input of the next one.
- Let the selection of Q and Q bar output of the preceding FF be controlled by the mode control input M such that, If M = 0, UP counting. So connect Q to CLK. If M = 1, DOWN counting. So connect Q bar to CLK.

Block Diagram



Truth Table



Inputs			Outputs
M	Q	$\overline{Q}$	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

$Y = Q$ for up counter

$Y = \overline{Q}$ for up counter

Operation

S.N.	Condition	Operation

1	**Case 1 − With M = 0 (Up counting mode)**	If M = 0 and M bar = 1, then the AND gates 1 and 3 in fig. will be enabled whereas the AND gates 2 and 4 will be disabled. Hence Q_A gets connected to the clock input of FF-B and Q_B gets connected to the clock input of FF-C. These connections are same as those for the normal up counter. Thus with M = 0 the circuit work as an up counter.
2	**Case 2: With M = 1 (Down counting mode)**	If M = 1, then AND gates 2 and 4 in fig. are enabled whereas the AND gates 1 and 3 are disabled. Hence Q_A bar gets connected to the clock input of FF-B and Q_B bar gets connected to the clock input of FF-C. These connections will produce a down counter. Thus with M = 1 the circuit works as a down counter.

Modulus Counter (MOD-N Counter)

The 2-bit ripple counter is called as MOD-4 counter and 3-bit ripple counter is called as MOD-8 counter. So in general, an n-bit ripple counter is called as modulo-N counter. Where, MOD number = 2^n.

Type of modulus

- 2-bit up or down (MOD-4)
- 3-bit up or down (MOD-8)
- 4-bit up or down (MOD-16)

Application of counters

- Frequency counters
- Digital clock

- Time measurement
- A to D converter
- Frequency divider circuits
- Digital triangular wave generator.

Unit V Finite State Machines and Programmable Devices

Introduction to Finite State Machines:

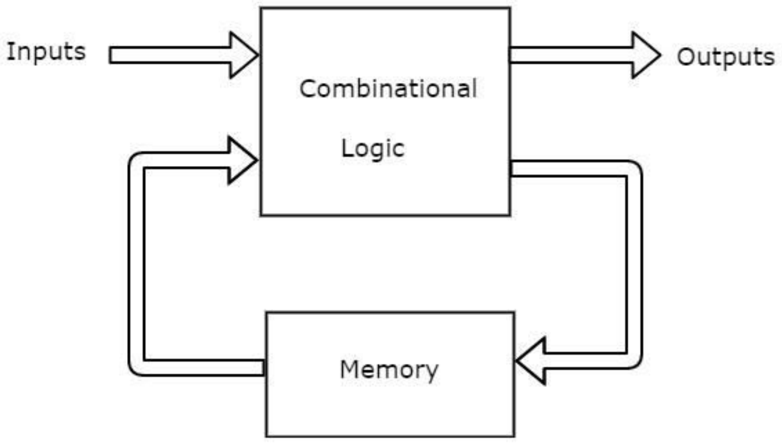
We know that synchronous sequential circuits change affect their states for every positive or negative transition of the clock signal based on the input. So, this behavior of synchronous sequential circuits can be represented in the graphical form and it is known as **state diagram**. A synchronous sequential circuit is also called as **Finite State Machine** (FSM), if it has finite number of states. There are two types of FSMs.

- Mealy State Machine
- Moore State Machine

Now, let us discuss about these two state machines one by one.

Mealy State Machine:

A Finite State Machine is said to be Mealy state machine, if outputs depend on both present inputs & present states. The **block diagram** of Mealy state machine is shown in the following figure.



As shown in figure, there are two parts present in Mealy state machine. Those are combinational logic and memory. Memory is useful to provide some or part of previous outputs present states as inputs of combinational logic.
So, based on the present inputs and present states, the Mealy state machine produces outputs. Therefore, the outputs will be valid only at positive or negative transition of the clock signal.

The **state diagram** of Mealy state machine is shown in the following figure.

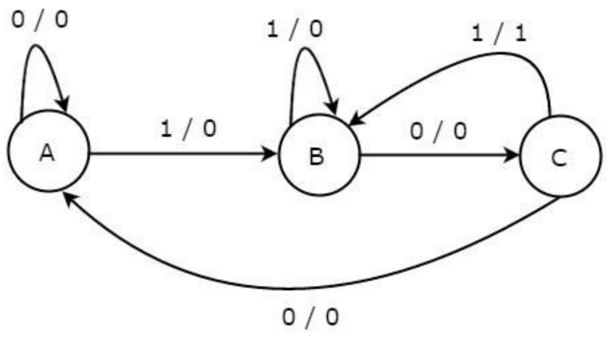


Figure 1. State Diagram

In the above figure, there are three states, namely A, B & C. These states are labelled inside the circles & each circle corresponds to one state. Transitions between these states are represented with directed lines. Here, 0 / 0, 1 / 0 & 1 / 1 denotes **input / output**. In the above figure, there are two transitions from each state based on the value of input, x.

In general, the number of states required in Mealy state machine is less than or equal to the number of states required in Moore state machine. There is an equivalent Moore state machine for each Mealy state machine.

Moore State Machine:

A Finite State Machine is said to be Moore state machine, if outputs depend only on present states. The **block diagram** of Moore state machine is shown in the following figure.

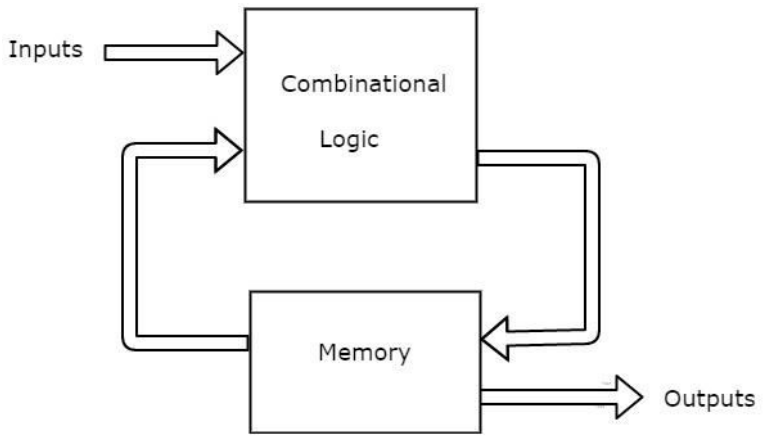


Figure 2. Moore State Machine

As shown in figure, there are two parts present in Moore state machine. Those are combinational logic and memory. In this case, the present inputs and present states determine the next states. So, based on next states, Moore state machine produces the outputs. Therefore, the outputs will be valid only after transition of the state.

The **state diagram** of Moore state machine is shown in the following figure.

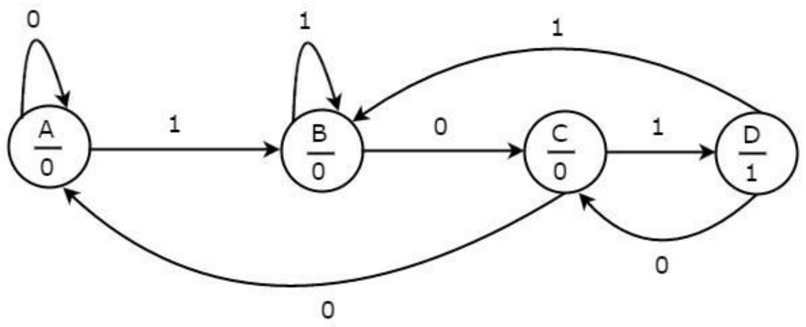


Figure 3. State Diagram for Moore State MAchine

In the above figure, there are four states, namely A, B, C & D. These states and the respective outputs are labelled inside the circles. Here, only the input value is labeled on each transition. In the above figure, there are two transitions from each state based on the value of input, x.

In general, the number of states required in Moore state machine is more than or equal to the number of states required in Mealy state machine. There is an equivalent Mealy state machine for each Moore state machine. So, based on the requirement we can use one of them.

State Reduction and State Assignment:

To illustrate the process of state reduction and state assignment first we have to know the concepts of the state diagram, state table, and state equation. In this article, we are going to learn all the topics related to state reduction and assignment.

State diagram: The state graph or state diagram is a pictorial representation of the relationships between the present state, the input state, the next state, and the output state of a sequential circuit i.e. A state diagram is a graphical representation of a sequential circuit's behavior.

Example: Consider an excitation table of J-K flip-flop

Table 1. Truth Table for J-K Flip-Flop

Q_n	Q_{n+1}	J	K
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

The state diagram of the above table is

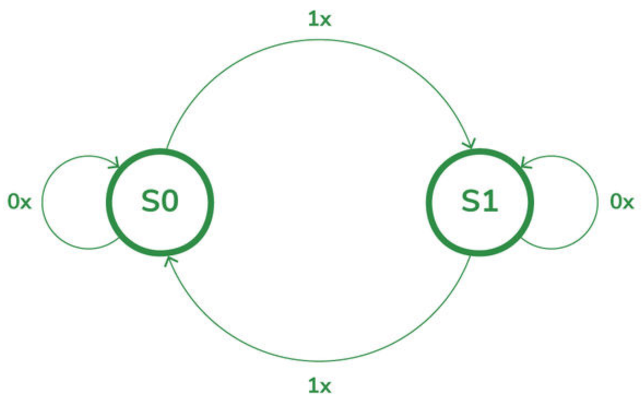


Figure 4. State Diagram of J-K flip-flop

State table: Even though the behavior of a sequential circuit can be conveniently described using a state diagram, for its implementation the information contained in the state diagram is to be translated into a state table. The tabular form of the state diagram is the state table. The present state, the next state, and the output are the three sections of the diagram.
The state table of JK flip-flop is:

Table 2. State Table for J-K Flip-Flop

Inputs		Present state	Output
J	K	Q	Q+ (Output)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

State equation: $Q_{n+1} = Q_n$ bar J + Q_n K bar

State reduction:

The state reduction technique generally prevents the addition of duplicate states. The reduction in redundant states reduces the number of flip-flops and logic gates, reducing the cost of the final circuit. Two states are said to be equivalent if every possible set of inputs generates exactly the same output and the same next state. When two states are equal, one of them can be eliminated without changing the input-output relationship. The state reduction algorithm is applied in the state table to reduce equivalent states.

State assignment:

State assignment refers to the process of assigning binary values to the states of a sequential machine. The binary values should be given to the states in such a way that flip-flop input functions may be implemented with a minimum number of logic gates.

State assignment rules are as follows:
Rule 1: States having the same next state for a given input condition should have assignments that can be grouped into logically adjacent cells in a K-map.
Rule 2: States that are the next states of a single state should have assignments that can be grouped into logically adjacent cells in a K-map.
Example 1: To explain the concept of state reduction let us consider the state table as

Table 3. State Reduction Table

	Next state		Output	
Present state	X=0	X=1	X=0	X=1
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	g	f	0	1
g	a	f	0	1

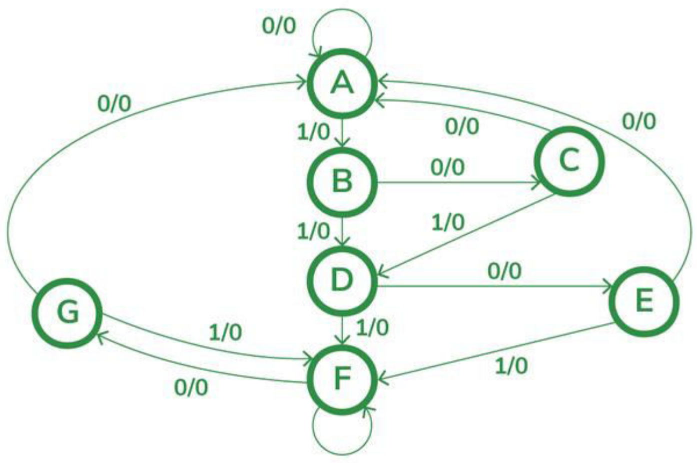


Figure 5. State diagram before reduction

Step1: First here we are supposed to identify two or more similar states in the next state and output state. In the above table if we observe states of e and g are having the same next state and output values for all combinations of input i.e. X=0 and X=1.
So eliminate the g state in the state table and wherever g is present replace it with e. Because e and g both are the same i.e. e=g.

Table 4. State Table

Present state	Next state		Output	
	X=0	X=1	X=0	X=1
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	e(g=e)	f	0	1

Step 2: Again check if any two states have similar values or not. If any two states have the same next state and output then eliminate one state.
Here d and f are having the same next state value and output. So eliminate f and wherever f is present replace it with d. Because both are the same d=f

Table 5. State Table

Present state	Next state		Output	
	X=0	X=1	X=0	X=1
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	d(d=f)	0	1
e	a	d(d=f)	0	1

Step 3: Further observe if any similar states are present or not. The states c and e are having same next states but they are having different outputs. So we cannot consider it a reduction state.

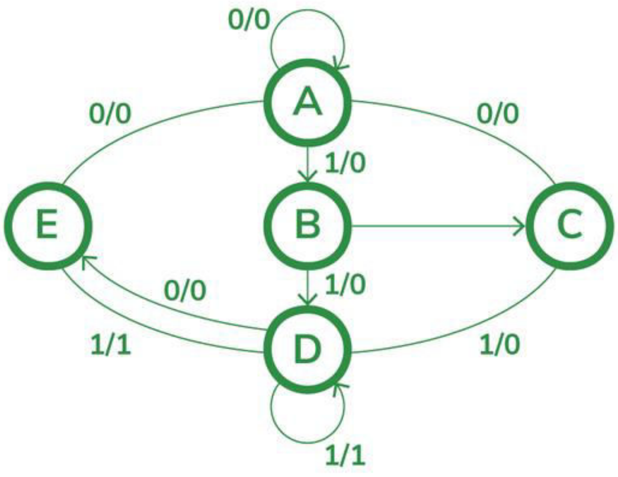


Figure 6. State diagram After reduction

Step 4: If you observed the state table, the states are represented by using the alphabet. We can not proceed further if we are having alphabets, so, assigning binary numbers to alphabets is called a state assignment.
To assign binary numbers to the state we have to consider the minimum number of bits.

The codes must contain n bits for a circuit with m states, where 2^n >= m. In this case, each state requires 2^3 >=5=>3 bits to be represented. With three bits, there are eight possible combinations, of which five can be used to represent the states.

Table 6. State Table

State	Assignment 1
	Binary
a	000
b	001
c	010
d	011
e	100

Step 5: Replacing the alphabets with binary numbers.

Table 7. Final State Table

	Next state		Output	
Present state	X=0	X=1	X=0	X=1
000	000	001	0	0
001	010	011	0	0
010	000	011	0	0
011	100	011	0	1
100	000	011	0	1

Random Access Memory-(RAM):

Random Access Memory is the internal memory of the CPU for storing data, program, and program result. It is a read/write memory which stores data until the machine is working. As soon as the machine is switched off, data is erased.

Access time in RAM is independent of the address, that is, each storage location inside the memory is as easy to reach as other locations and takes the same amount of time. Data in the RAM can be accessed randomly but it is very expensive.

RAM is volatile, i.e. data stored in it is lost when we switch off the computer or if there is a power failure. Hence, a backup Uninterruptible Power System (UPS) is often used with computers. RAM is small, both in terms of its physical size and in the amount of data it can hold.

RAM is of two types −

- Static RAM (SRAM)
- Dynamic RAM (DRAM)

Read Only Memory (ROM):

ROM stands for **Read Only Memory**. The memory from which we can only read but cannot write on it. This type of memory is non-volatile. The information is stored permanently in such memories during manufacture. A ROM stores such instructions that are required to start a computer. This operation is referred to as **bootstrap**. ROM chips are not only used in the computer but also in other electronic items like washing machine and microwave oven.

Let us now discuss the various types of ROMs and their characteristics.

MROM (Masked ROM)

The very first ROMs were hard-wired devices that contained a pre-programmed set of data or instructions. These kind of ROMs are known as masked ROMs, which are inexpensive.

PROM (Programmable Read Only Memory)

PROM is read-only memory that can be modified only once by a user. The user buys a blank PROM and enters the desired contents using a PROM program. Inside the PROM chip, there are small fuses which are burnt open during programming. It can be programmed only once and is not erasable.

EPROM (Erasable and Programmable Read Only Memory)

EPROM can be erased by exposing it to ultra-violet light for a duration of up to 40 minutes. Usually, an EPROM eraser achieves this function. During programming, an electrical charge is trapped in an insulated gate region. The charge is retained for more than 10 years because the charge has no leakage path. For erasing this charge, ultra-violet light is passed through a quartz crystal window (lid). This exposure to ultra-violet light dissipates the charge. During normal use, the quartz lid is sealed with a sticker.

EEPROM (Electrically Erasable and Programmable Read Only Memory)

EEPROM is programmed and erased electrically. It can be erased and reprogrammed about ten thousand times. Both erasing and programming take about 4 to 10 ms (millisecond). In EEPROM, any location can be selectively erased and programmed. EEPROMs can be erased one byte at a time, rather than erasing the entire chip. Hence, the process of reprogramming is flexible but slow.

Advantages of ROM

The advantages of ROM are as follows −

- Non-volatile in nature
- Cannot be accidentally changed
- Cheaper than RAMs
- Easy to test
- More reliable than RAMs
- Static and do not require refreshing
- Contents are always known and can be verified

Programmable Array Logic (PAL):

PAL is a programmable logic device that has Programmable AND array & fixed OR array. The advantage of PAL is that we can generate only the required product terms of Boolean function instead of generating all the min terms by using programmable AND gates. The **block diagram** of PAL is shown in the following figure.

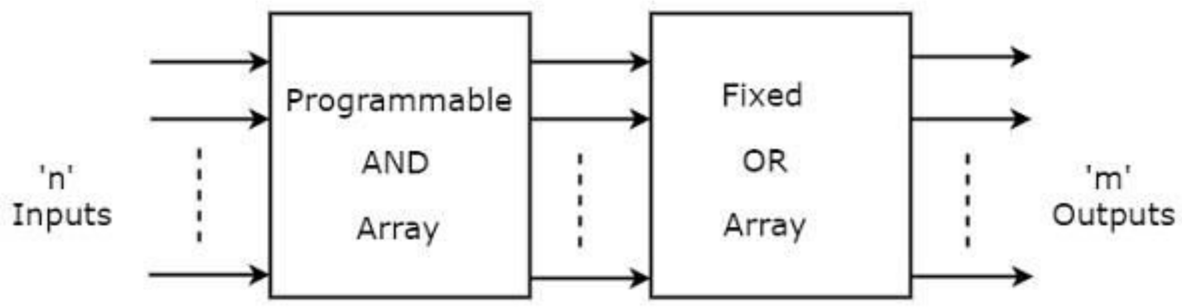


Figure 7. Block diagram for PAL

Here, the inputs of AND gates are programmable. That means each AND gate has both normal and complemented inputs of variables. So, based on the requirement, we can program any of those inputs. So, we can generate only the required **product terms** by using these AND gates.

Here, the inputs of OR gates are not of programmable type. So, the number of inputs to each OR gate will be of fixed type. Hence, apply those required product terms to each OR gate as inputs. Therefore, the outputs of PAL will be in the form of **sum of products form**.

Example

Let us implement the following **Boolean functions** using PAL.

$$A=XY+XZ'A=XY+XZ'$$

$$A=XY'+YZ'A=XY'+YZ'$$

The given two functions are in sum of products form. There are two product terms present in each Boolean function. So, we require four programmable AND gates & two fixed OR gates for producing those two functions. The corresponding **PAL** is shown in the following figure.

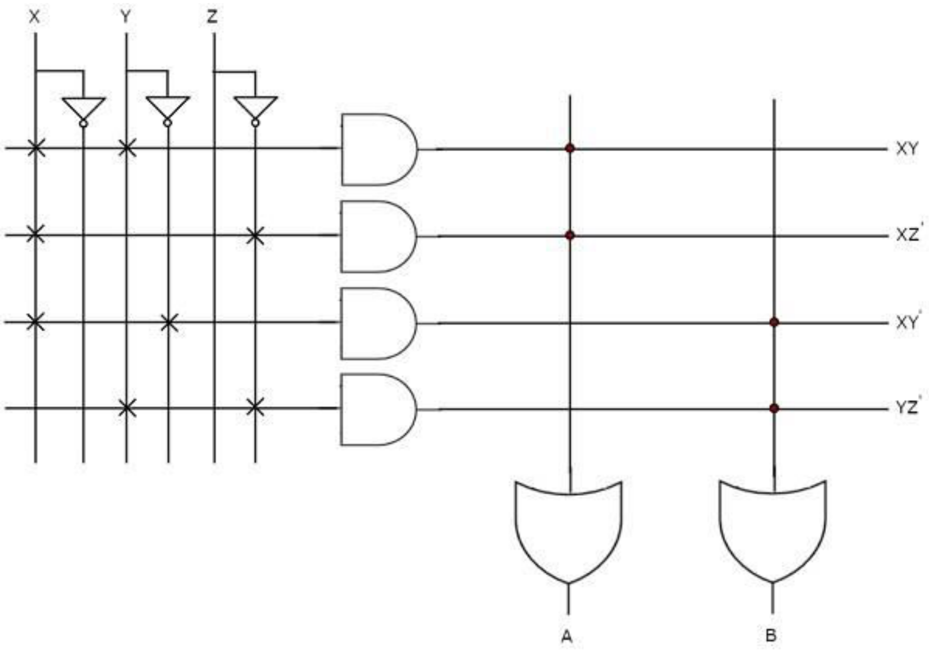


Figure 8. PAL Implementation

The **programmable AND gates** have the access of both normal and complemented inputs of variables. In the above figure, the inputs X, X′X′, Y, Y′Y′, Z & Z′Z′, are available at the inputs of each AND gate. So, program only the required literals in order to generate one product term by each AND gate. The symbol 'X' is used for programmable connections.

Here, the inputs of OR gates are of fixed type. So, the necessary product terms are connected to inputs of each **OR gate**. So that the OR gates produce the respective Boolean functions. The symbol '.' is used for fixed connections.

Programmable Logic Array (PLA):

PLA is a programmable logic device that has both Programmable AND array & Programmable OR array. Hence, it is the most flexible PLD. The **block diagram** of PLA is shown in the following figure.

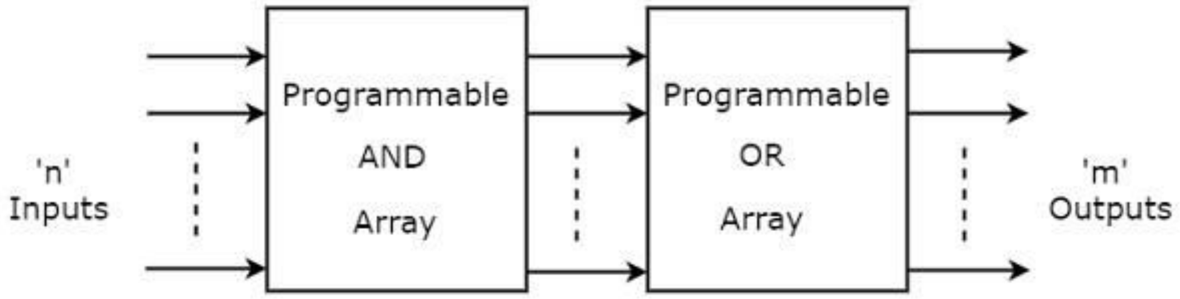


Figure 9. Block Diagram of PLA

Here, the inputs of AND gates are programmable. That means each AND gate has both normal and complemented inputs of variables. So, based on the requirement, we can program any of those inputs. So, we can generate only the required **product terms** by using these AND gates.

Here, the inputs of OR gates are also programmable. So, we can program any number of required product terms, since all the outputs of AND gates are applied as inputs to each OR gate. Therefore, the outputs of PAL will be in the form of **sum of products form**.

Example

Let us implement the following **Boolean functions** using PLA.

$$A=XY+XZ'A=XY+XZ'$$

$$B=XY'+YZ+XZ'B=XY'+YZ+XZ'$$

The given two functions are in sum of products form. The number of product terms present in the given Boolean functions A & B are two and three respectively. One product term, $Z'XZ'X$ is common in each function.

So, we require four programmable AND gates & two programmable OR gates for producing those two functions. The corresponding **PLA** is shown in the following figure.

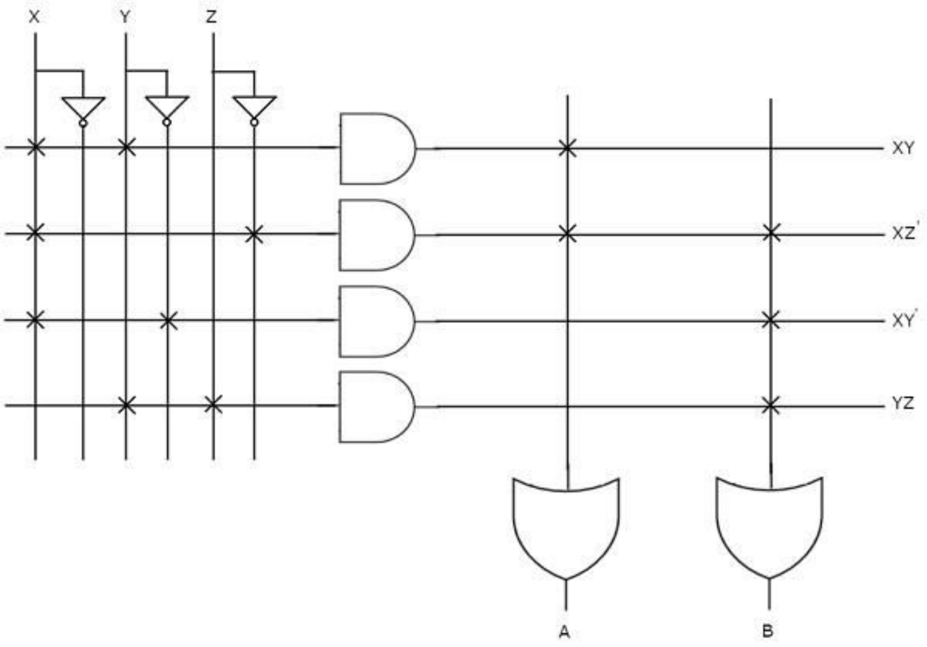


Figure 10. PLA Implementation

The **programmable AND gates** have the access of both normal and complemented inputs of variables. In the above figure, the inputs X, X′X′, Y, Y′Y′, Z & Z′Z′, are available at the inputs of each AND gate. So, program only the required literals in order to generate one product term by each AND gate.

All these product terms are available at the inputs of each **programmable OR gate**. But, only program the required product terms in order to produce the respective Boolean functions by each OR gate. The symbol 'X' is used for programmable connections.