

# **DIGITAL SYSTEM DESIGN**

Text Book:

Stephen Brown and Zvonko Vranesic, *Fundamentals of Digital Logic with Verilog Design* (3e), Tata McGraw Hill 2014.

The slides in this document are prepared using the above text book

For contact,

Email: hema.shama@manipal.edu

Mobile: 9481752532

# Variables and Functions

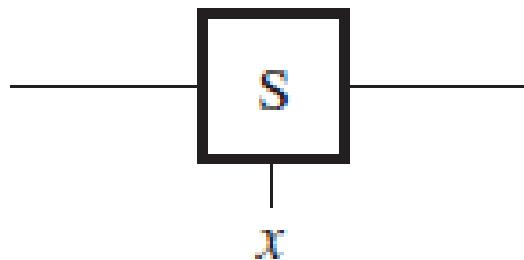


$$x = 0$$

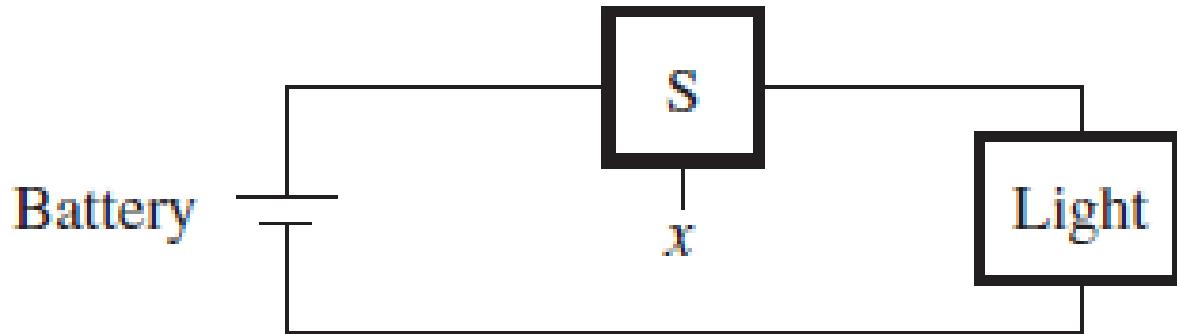


$$x = 1$$

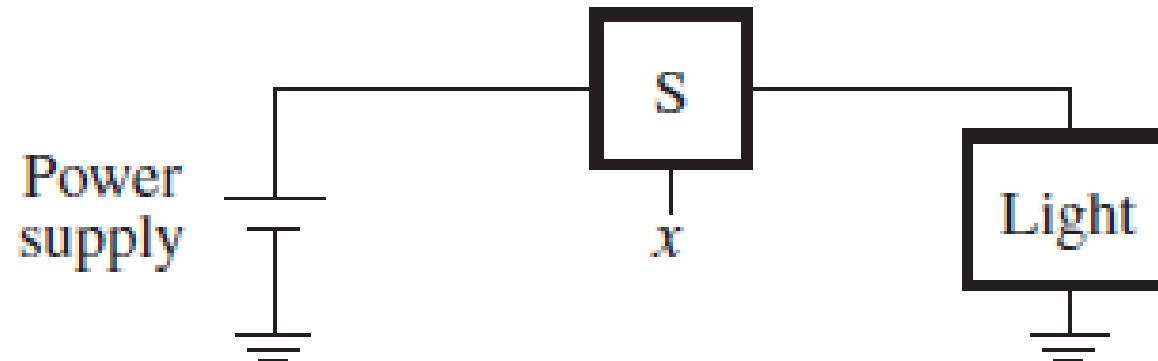
(a) Two states of a switch



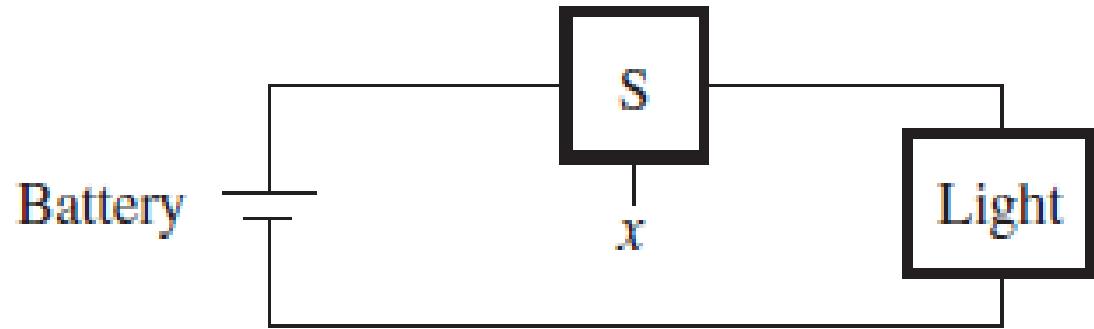
(b) Symbol for a switch



(a) Simple connection to a battery



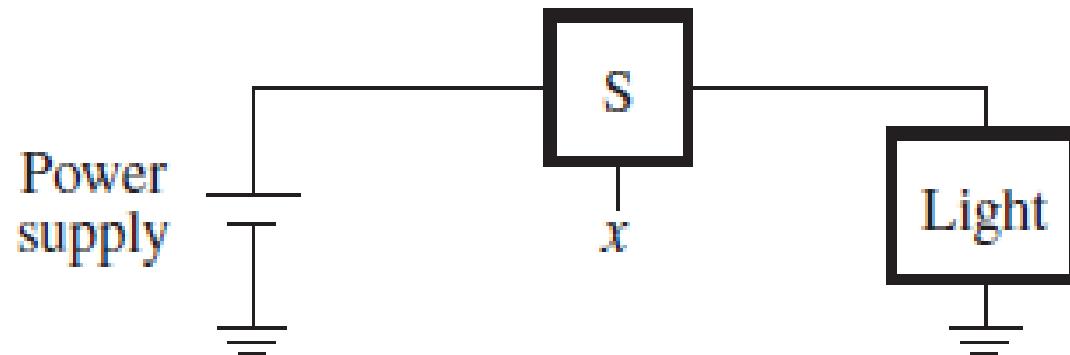
(b) Using a ground connection as the return path



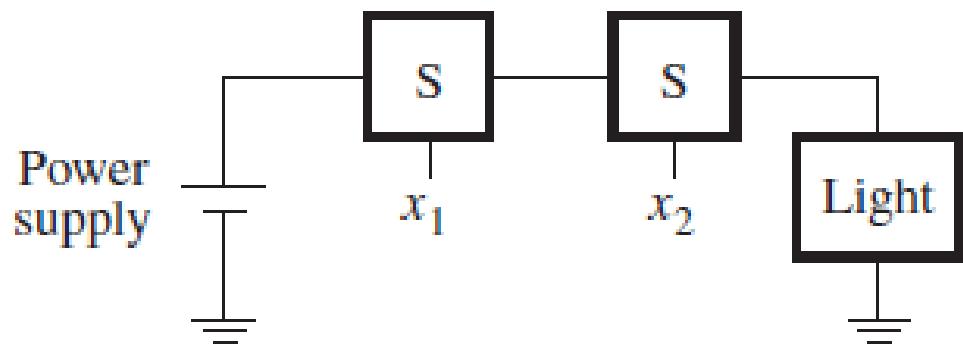
(a) Simple connection to a battery

$$L(x) = x$$

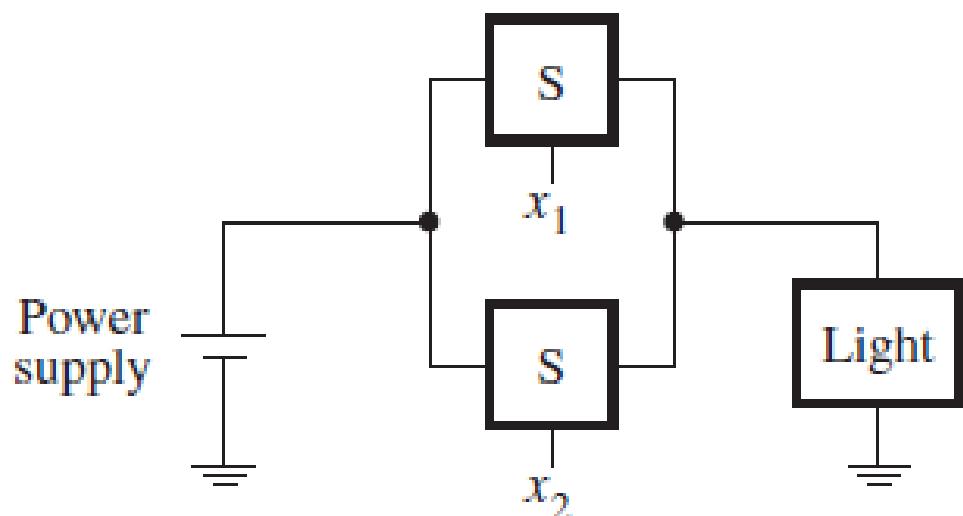
We say that  
 $L(x) = x$  is a *logic function* and that  $x$  is an *input variable*.



(b) Using a ground connection as the return path



(a) The logical AND function (series connection)



(b) The logical OR function (parallel connection)

$$L(x_1, x_2) = x_1 \cdot x_2$$

where  $L = 1$  if  $x_1 = 1$  and  $x_2 = 1$ ,  
 $L = 0$  otherwise.

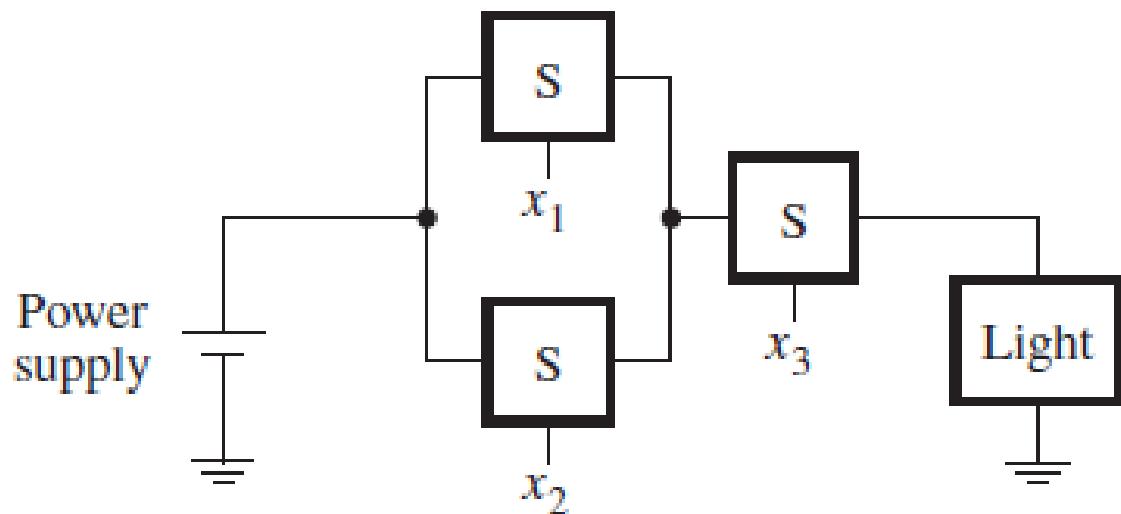
The “ $\cdot$ ” symbol is called the *AND operator*, and the circuit in Figure (a) is said to implement a *logical AND function*.

$$L(x_1, x_2) = x_1 + x_2$$

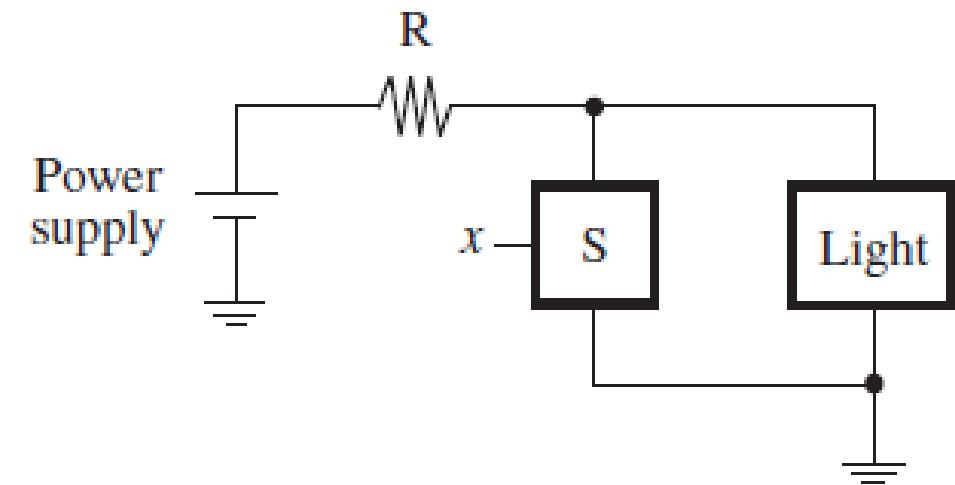
where  $L = 1$  if  $x_1 = 1$  or  $x_2 = 1$  or if  $x_1 = x_2 = 1$ ,

$L = 0$  if  $x_1 = x_2 = 0$ .

The  $+$  symbol is called the *OR operator*, and the circuit in Figure 2.3b is said to implement a *logical OR function*.



$$L(x_1, x_2, x_3) = (x_1 + x_2) \cdot x_3$$



$$L(x) = x'$$

The complement operation can be applied to a single variable or to more complex operations.

For example, if

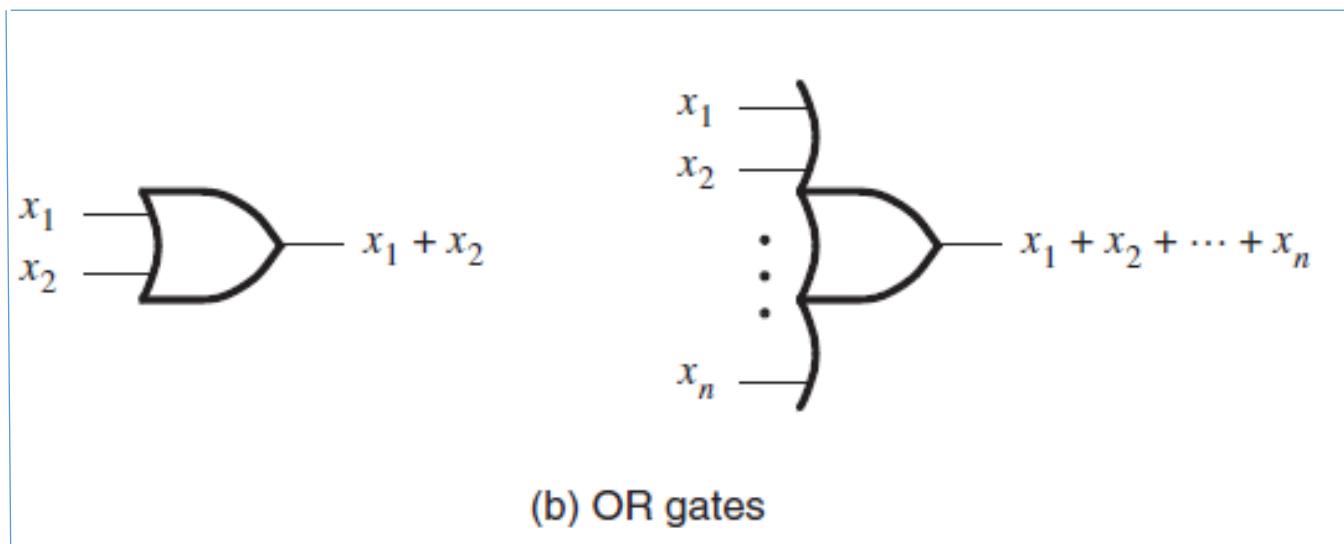
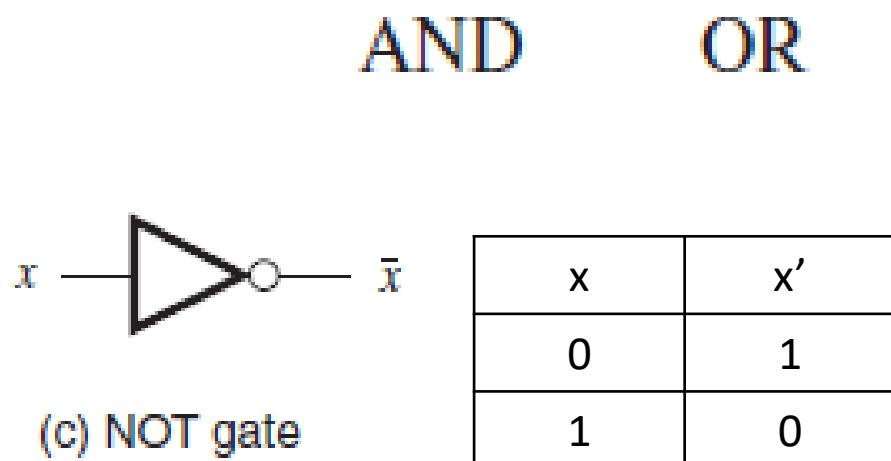
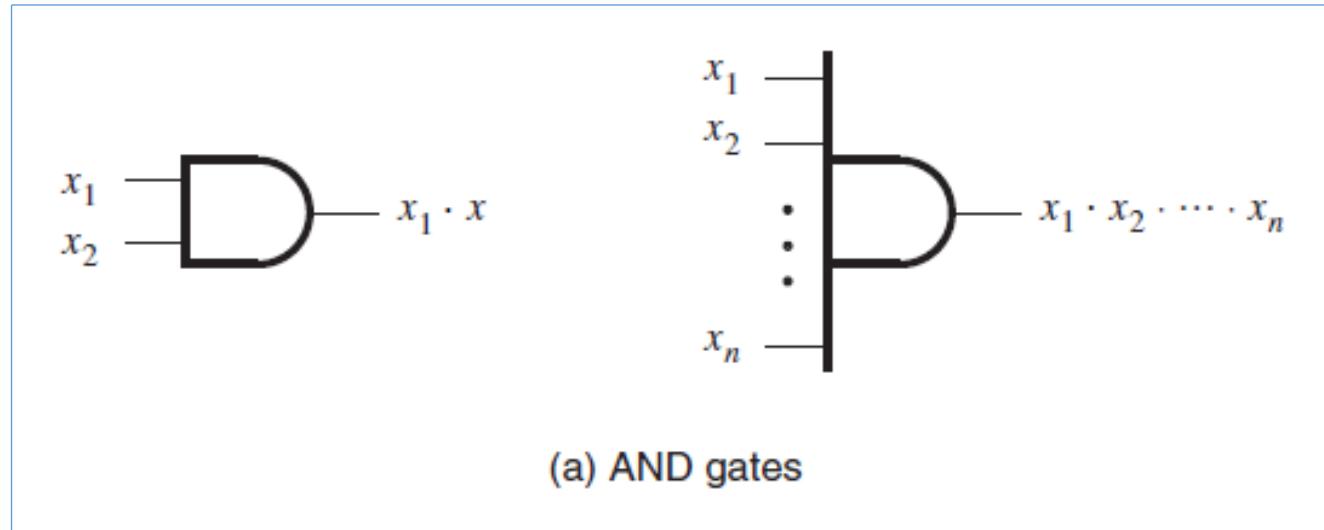
$$f(x_1, x_2) = x_1 + x_2$$

then the complement of  $f$  is

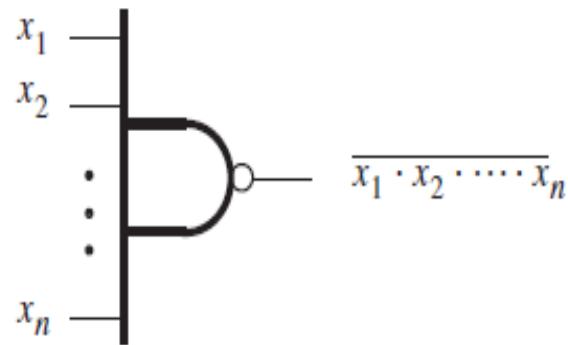
$$f'(x_1, x_2) = (x_1 + x_2)'$$

# Truth Tables

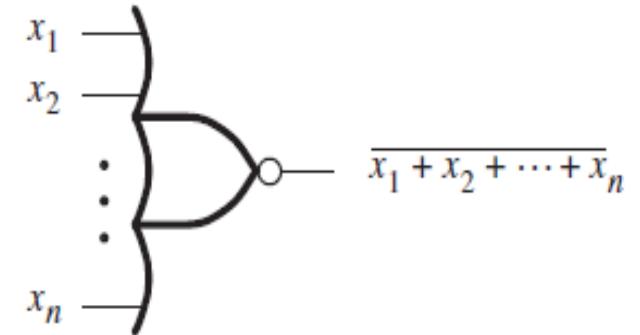
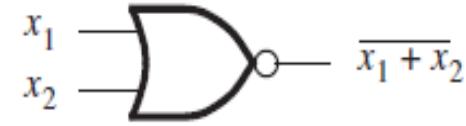
$x_1$	$x_2$	$x_1 \cdot x_2$	$x_1 + x_2$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1



- AND gate gives *high* output if and only if all of its inputs are *high*
- OR gate output is LOW if and only if all of its inputs are LOW.



(a) NAND gates



(b) NOR gates

A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

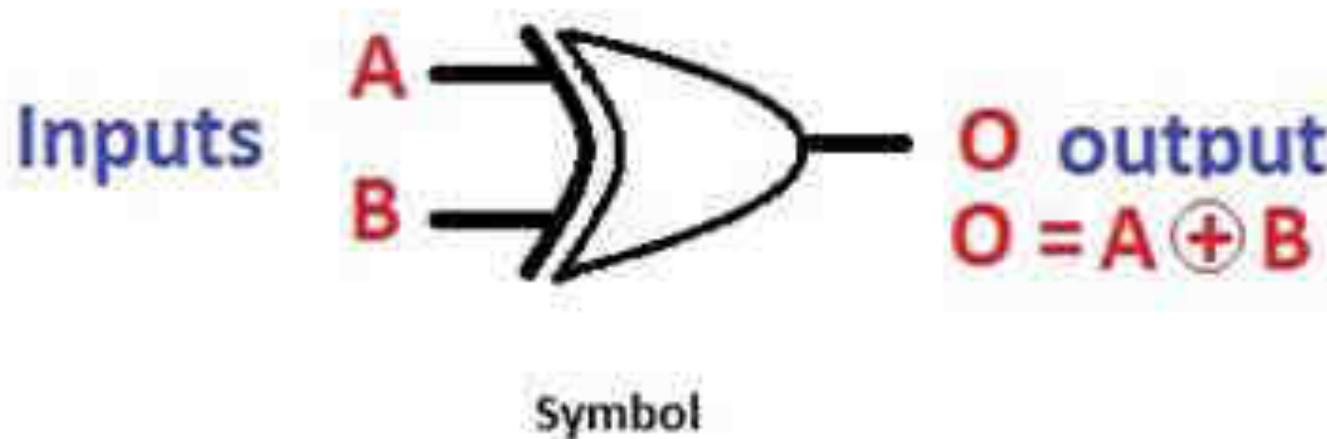
NAND

A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

NOR

- NAND gate gives LOW output if and only if all of its inputs are HIGH
- NOR gate output is HIGH if and only if all of its inputs are LOW.

## Exclusive OR (XOR)



Inputs		Output
A	B	O
0	0	0
0	1	1
1	0	1
1	1	0

Truth table

XOR gate output is HIGH if it has odd number of HIGH inputs.

# Boolean Algebra

## Axioms

$$1a. 0 \cdot 0 = 0$$

$$1b. 1 + 1 = 1$$

$$2a. 1 \cdot 1 = 1$$

$$2b. 0 + 0 = 0$$

$$3a. 0 \cdot 1 = 1 \cdot 0 = 0$$

$$3b. 1 + 0 = 0 + 1 = 1$$

$$4a. \text{If } x=0, \text{ then } \bar{x} = 1$$

$$4b. \text{If } x=1, \text{ then } \bar{x} = 0$$

## Single-Variable Theorems

$$5a. x \cdot 0 = 0$$

$$5b. x + 1 = 1$$

$$6a. x \cdot 1 = x$$

$$6b. x + 0 = x$$

$$7a. x \cdot x = x$$

$$7b. x + x = x$$

$$8a. x \cdot \bar{x} = 0$$

$$8b. x + \bar{x} = 1$$

$$9. \bar{\bar{x}} = x$$

# Duality

- Given a logic expression, its *dual* is obtained by replacing all  $+$  operators with  $\cdot$  operators, and vice versa, and by replacing all 0s with 1s, and vice versa.
- The dual of any true statement in Boolean algebra is also a true statement.

# Two- and Three-Variable Properties

$$10a. \quad x \cdot y = y \cdot x$$

*Commutative*

$$10b. \quad x + y = y + x$$

$$11a. \quad x \cdot (y \cdot z) = (x \cdot y) \cdot z$$

*Associative*

$$11b. \quad x + (y + z) = (x + y) + z$$

$$12a. \quad x \cdot (y + z) = x \cdot y + x \cdot z$$

*Distributive*

$$12b. \quad x + y \cdot z = (x + y) \cdot (x + z)$$

$$13a. \quad x + x \cdot y = x$$

*Absorption*

$$13b. \quad x \cdot (x + y) = x$$

$$14a. \quad x \cdot y + x \cdot \bar{y} = x$$

Combining

$$14b. \quad (x + y) \cdot (x + \bar{y}) = x$$

$$15a. \quad \overline{x \cdot y} = \bar{x} + \bar{y}$$

DeMorgan's theorem

$$15b. \quad \overline{x + y} = \bar{x} \cdot \bar{y}$$

$$16a. \quad x + \bar{x} \cdot y = x + y$$

$$16b. \quad x \cdot (\bar{x} + y) = x \cdot y$$

$$17a. \quad x \cdot y + y \cdot z + \bar{x} \cdot z = x \cdot y + \bar{x} \cdot z$$

Consensus

$$17b. \quad (x + y) \cdot (y + z) \cdot (\bar{x} + z) = (x + y) \cdot (\bar{x} + z)$$

# Precedence of Operations

- In the absence of parentheses, operation in a logic expression must be performed in the order: NOT, AND and OR.
- Given  $f=a.b+a'.b'$   
first,  $a'$  and  $b'$  are calculated, then  $ab$  and  $a'b'$  and finally  $ab+a'b'$

Qn: Prove the validity of the logic equation

$$(x_1 + x_3) \cdot (\bar{x}_1 + \bar{x}_3) = x_1 \cdot \bar{x}_3 + \bar{x}_1 \cdot x_3$$

Solution:

$$\text{LHS} = (x_1 + x_3) \cdot \bar{x}_1 + (x_1 + x_3) \cdot \bar{x}_3 \quad (\text{distributive property})$$

$$\text{LHS} = x_1 \cdot \bar{x}_1 + x_3 \cdot \bar{x}_1 + x_1 \cdot \bar{x}_3 + x_3 \cdot \bar{x}_3$$

$$\text{LHS} = x_3 \cdot \overline{x_1} + x_1 \cdot \overline{x_3}$$

$$\text{LHS} = \overline{x_1} \cdot \overline{x_3} + \overline{x_3} \cdot x_1$$

which is the same as the right-hand side of the initial equation.

Qn: Prove the validity of the logic equation

$$x_1 \cdot \bar{x}_3 + \bar{x}_2 \cdot \bar{x}_3 + x_1 \cdot x_3 + \bar{x}_2 \cdot x_3 = \bar{x}_1 \cdot \bar{x}_2 + x_1 \cdot x_2 + x_1 \cdot \bar{x}_2$$

$$\text{LHS} = x_1 \cdot \bar{x}_3 + x_1 \cdot x_3 + \bar{x}_2 \cdot \bar{x}_3 + \bar{x}_2 \cdot x_3$$

$$= x_1 \cdot (\bar{x}_3 + x_3) + \bar{x}_2 \cdot (\bar{x}_3 + x_3)$$

$$= x_1 \cdot 1 + \bar{x}_2 \cdot 1 \quad (x + x' = 1)$$

$$= x_1 + \bar{x}_2$$

$$\text{RHS} = \bar{x}_1 \cdot \bar{x}_2 + x_1 \cdot (\bar{x}_1 + \bar{x}_2)$$

$$= \bar{x}_1 \cdot \bar{x}_2 + x_1 \cdot 1$$

$$= \bar{x}_1 \cdot \bar{x}_2 + x_1$$

$$= x_1 + \bar{x}_1 \cdot \bar{x}_2 \quad (x + x'y = x + y)$$

$$= x_1 + \bar{x}_2$$

## Sum-of-Products and Product-of-Sums Forms

If a function  $f$  is specified in the form of a truth table, then an expression that realizes  $f$  can be obtained by considering either the rows in the table for which  $f = 1$ , or by considering the rows for which  $f = 0$ .

## Minterms

For a function of  $n$  variables, a product term in which each of the  $n$  variables appears once, either in uncomplemented or complemented form is called a *minterm*.

For a given row of the truth table, the minterm is formed by including  $x_i$  if  $x_i = 1$  and by including  $x'_i$  if  $x_i = 0$ .

Row number	$x_1$	$x_2$	$x_3$	Minterm
0	0	0	0	$m_0 = \bar{x}_1 \bar{x}_2 \bar{x}_3$
1	0	0	1	$m_1 = \bar{x}_1 \bar{x}_2 x_3$
2	0	1	0	$m_2 = \bar{x}_1 x_2 \bar{x}_3$
3	0	1	1	$m_3 = \bar{x}_1 x_2 x_3$
4	1	0	0	$m_4 = x_1 \bar{x}_2 \bar{x}_3$
5	1	0	1	$m_5 = x_1 \bar{x}_2 x_3$
6	1	1	0	$m_6 = x_1 x_2 \bar{x}_3$
7	1	1	1	$m_7 = x_1 x_2 x_3$

## Sum-of-Products Form

Any function  $f$  can be represented by a sum of minterms that correspond to the rows in the truth table for which  $f = 1$ .

Example:

$x_1$	$x_2$	$f(x_1, x_2)$
0	0	1
0	1	1
1	0	0
1	1	1

$$\begin{aligned}f &= \overline{x_1}\overline{x_2} + \overline{x_1}x_2 + x_1\overline{x_2} \\&= m_0 + m_1 + m_3\end{aligned}$$

A logic expression consisting of product (AND) terms that are summed (ORed) is said to be in the *sum-of products* (*SOP*) form. If each product term is a minterm, then the expression is called a *canonical sum-of-products* for the function  $f$

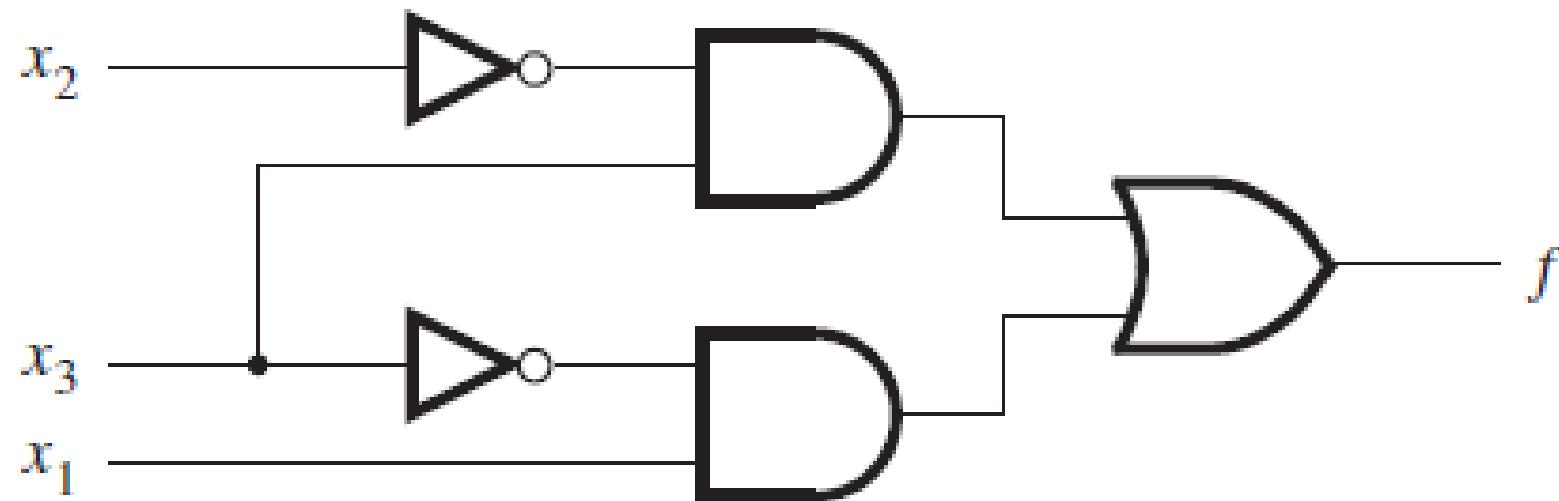
## Example2:

$x_1$	$x_2$	$x_3$	$f(x_1, x_2, x_3)$
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

$$f(x_1, x_2, x_3) = \overline{x}_1 \overline{x}_2 x_3 + x_1 \overline{x}_2 \overline{x}_3 + x_1 \overline{x}_2 x_3 + x_1 x_2 \overline{x}_3$$

$$\begin{aligned}
 f(x_1, x_2, x_3) &= \overline{x}_1 \overline{x}_2 x_3 + x_1 \overline{x}_2 \overline{x}_3 + x_1 \overline{x}_2 x_3 + x_1 x_2 \overline{x}_3 \\
 &= (\overline{x}_1 + x_1) \overline{x}_2 x_3 + x_1 (\overline{x}_2 + x_2) \overline{x}_3 \\
 &= \overline{x}_2 x_3 + x_1 \overline{x}_3
 \end{aligned}$$

- This is the minimum-cost sum-of-products expression for  $f$



The cost of a logic circuit is the total number of gates plus the total number of inputs to all gates in the circuit.

The cost of the previous network is 13, because there are five gates and eight inputs to the gates.

The previous function  $f$  can also be specified as

$$f(x_1, x_2, x_3) = \sum(m_1, m_4, m_5, m_6)$$

or even more simply as

$$f(x_1, x_2, x_3) = \sum m(1, 4, 5, 6)$$

## Qn1: Simplify

$$f(x_1, x_2, x_3) = \sum m(2, 3, 4, 6, 7)$$

$$\begin{aligned}f &= m_2 + m_3 + m_4 + m_6 + m_7 \\&= \bar{x}_1 x_2 \bar{x}_3 + \bar{x}_1 x_2 x_3 + x_1 \bar{x}_2 \bar{x}_3 + x_1 x_2 \bar{x}_3 + x_1 x_2 x_3 \\&= \bar{x}_1 x_2 (\bar{x}_3 + x_3) + x_1 (\bar{x}_2 + x_2) \bar{x}_3 + x_1 x_2 (\bar{x}_3 + x_3) \\&= \bar{x}_1 x_2 + x_1 \bar{x}_3 + x_1 x_2 \\&= (\bar{x}_1 + x_1) x_2 + x_1 \bar{x}_3 \\&= x_2 + x_1 \bar{x}_3\end{aligned}$$

## Qn2: Simplify

$$f(x_1, x_2, x_3, x_4) = \sum m(3, 7, 9, 12, 13, 14, 15)$$

$$\begin{aligned} f &= \bar{x}_1 \bar{x}_2 x_3 x_4 + \bar{x}_1 x_2 x_3 x_4 + x_1 \bar{x}_2 \bar{x}_3 x_4 + x_1 x_2 \bar{x}_3 \bar{x}_4 + x_1 x_2 \bar{x}_3 x_4 + x_1 x_2 x_3 \bar{x}_4 + x_1 x_2 x_3 x_4 \\ &= \bar{x}_1 (\bar{x}_2 + x_2) x_3 x_4 + x_1 (\bar{x}_2 + x_2) \bar{x}_3 x_4 + x_1 x_2 \bar{x}_3 (\bar{x}_4 + x_4) + x_1 x_2 x_3 (\bar{x}_4 + x_4) \\ &= \bar{x}_1 x_3 x_4 + x_1 \bar{x}_3 x_4 + x_1 x_2 \bar{x}_3 + x_1 x_2 x_3 \\ &= \bar{x}_1 x_3 x_4 + x_1 \bar{x}_3 x_4 + x_1 x_2 (\bar{x}_3 + x_3) \\ &= \bar{x}_1 x_3 x_4 + x_1 \bar{x}_3 x_4 + x_1 x_2 \end{aligned}$$

## Maxterms

- The principle of duality suggests that if it is possible to synthesize a function  $f$  by considering the rows in the truth table for which  $f = 1$ , then it should also be possible to synthesize  $f$  by considering the rows for which  $f = 0$ .
- This alternative approach uses the complements of minterms, which are called *maxterms*.

Row number	$x_1$	$x_2$	$x_3$	Maxterm
0	0	0	0	$M_0 = x_1 + x_2 + x_3$
1	0	0	1	$M_1 = x_1 + x_2 + \bar{x}_3$
2	0	1	0	$M_2 = x_1 + \bar{x}_2 + x_3$
3	0	1	1	$M_3 = x_1 + \bar{x}_2 + \bar{x}_3$
4	1	0	0	$M_4 = \bar{x}_1 + x_2 + x_3$
5	1	0	1	$M_5 = \bar{x}_1 + x_2 + \bar{x}_3$
6	1	1	0	$M_6 = \bar{x}_1 + \bar{x}_2 + x_3$
7	1	1	1	$M_7 = \bar{x}_1 + \bar{x}_2 + \bar{x}_3$

## Product-of-Sums Form

- If a given function  $f$  is specified by a truth table, then its complement  $f'$  can be represented by a sum of minterms for which  $f' = 1$ , which are the rows where  $f = 0$ .

Example:

$x_1$	$x_2$	$f(x_1, x_2)$
0	0	1
0	1	1
1	0	0
1	1	1

$$\begin{aligned} f(x_1, x_2) &= m_2 \\ &= x_1 \bar{x}_2 \end{aligned}$$

If we complement this expression using DeMorgan's theorem, the result is

$$\begin{aligned}\overline{\overline{f}} &= f = \overline{x_1 \overline{x_2}} \\ &= \overline{x_1} + x_2\end{aligned}$$

$$f = \overline{m_2} = M_2$$

## Example2:

$x_1$	$x_2$	$x_3$	$f(x_1, x_2, x_3)$
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

$$\begin{aligned}\bar{f}(x_1, x_2, x_3) &= m_0 + m_2 + m_3 + m_7 \\ &= \bar{x}_1 \bar{x}_2 \bar{x}_3 + \bar{x}_1 x_2 \bar{x}_3 + \bar{x}_1 x_2 x_3 + x_1 x_2 x_3\end{aligned}$$

Then  $f$  can be expressed as

$$\begin{aligned}f &= \overline{m_0 + m_2 + m_3 + m_7} \\&= \overline{m_0} \cdot \overline{m_2} \cdot \overline{m_3} \cdot \overline{m_7} \\&= M_0 \cdot M_2 \cdot M_3 \cdot M_7 \\&= (x_1 + x_2 + x_3)(\bar{x}_1 + \bar{x}_2 + x_3)(x_1 + \bar{x}_2 + \bar{x}_3)(\bar{x}_1 + \bar{x}_2 + \bar{x}_3)\end{aligned}$$

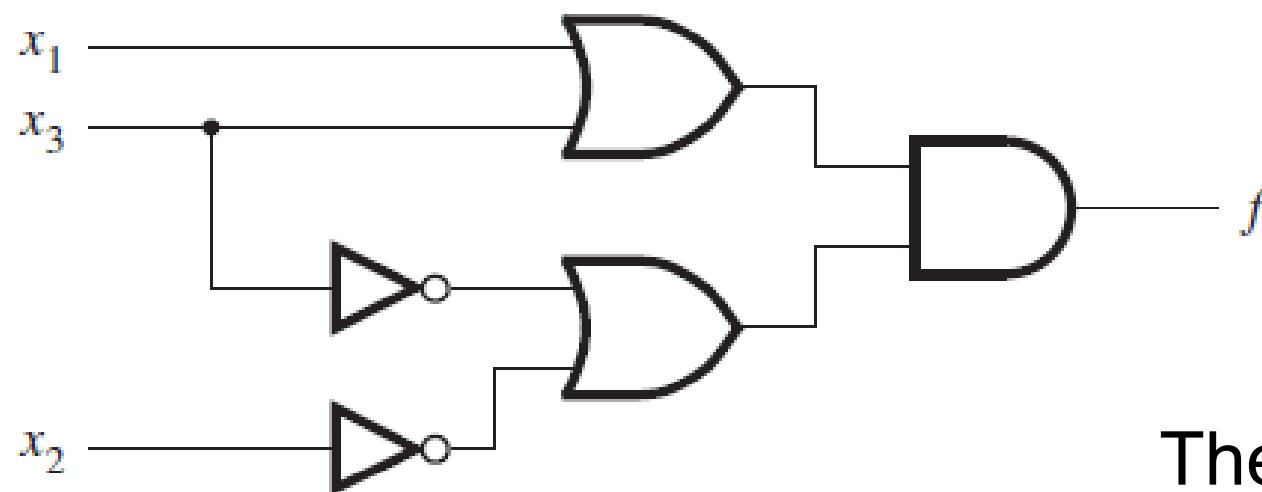
- A logic expression consisting of sum (OR) terms that are the factors of a logical product (AND) is said to be of the *product-of-sums* (POS) form.
- If each sum term is a maxterm, then the expression is called a *canonical product-of-sums* for the given function.

$$f = (x_1 + x_2 + x_3)(x_1 + \bar{x}_2 + x_3)(x_1 + \bar{x}_2 + \bar{x}_3)(\bar{x}_1 + \bar{x}_2 + \bar{x}_3)$$

Using the commutative property and the associative property,

$$= ((x_1 + x_3) + x_2)((x_1 + x_3) + \bar{x}_2)(x_1 + (\bar{x}_2 + \bar{x}_3))(\bar{x}_1 + (\bar{x}_2 + \bar{x}_3))$$

$$f = (x_1 + x_3)(\bar{x}_2 + \bar{x}_3) \quad (\text{using } (x + y)(x + y') = x)$$



The cost of this network is 13.

An alternative way of specifying our sample function is

$$f(x_1, x_2, x_3) = \Pi(M_0, M_2, M_3, M_7)$$

or more simply

$$f(x_1, x_2, x_3) = \Pi M(0, 2, 3, 7)$$

$$f(x_1,x_2,x_3)=\sum m(2,3,4,6,7)$$

$$=\Pi M\left( 0,1,5\right)$$

$$=M_0\cdot M_1\circ M_5$$

$$=(x_1+x_2+x_3)(x_1+x_2+\overline{x}_3)(\overline{x}_1+x_2+\overline{x}_3)$$

$$\begin{aligned}
 f &= (x_1 + x_2 + x_3)(x_1 + x_2 + \bar{x}_3)(\bar{x}_1 + x_2 + \bar{x}_3) \\
 &= (x_1 + x_2 + x_3)(x_1 + x_2 + \bar{x}_3)(x_1 + x_2 + \bar{x}_3)(\bar{x}_1 + x_2 + \bar{x}_3) \\
 &= ((x_1 + x_2) + x_3)((x_1 + x_2) + \bar{x}_3)(x_1 + (x_2 + \bar{x}_3))(\bar{x}_1 + (x_2 + \bar{x}_3)) \\
 &= ((x_1 + x_2) + x_3\bar{x}_3)(x_1\bar{x}_1 + (x_2 + \bar{x}_3)) \\
 &= (x_1 + x_2)(x_2 + \bar{x}_3) \quad (\text{using } (x + y)(x + y') = x)
 \end{aligned}$$

Another way of deriving this product-of-sums expression is to use the sum-of-products form of  $f'$ . Thus,

$$\begin{aligned}\bar{f}(x_1, x_2, x_3) &= \sum m(0, 1, 5) \\&= \bar{x}_1 \bar{x}_2 \bar{x}_3 + \bar{x}_1 \bar{x}_2 x_3 + x_1 \bar{x}_2 x_3 \\&= \bar{x}_1 \bar{x}_2 \bar{x}_3 + \bar{x}_1 \bar{x}_2 x_3 + \bar{x}_1 x_2 x_3 + x_1 \bar{x}_2 x_3 \\&= \bar{x}_1 \bar{x}_2 (\bar{x}_3 + x_3) + \bar{x}_2 x_3 (\bar{x}_1 + x_1) \\&= \bar{x}_1 \bar{x}_2 + \bar{x}_2 x_3\end{aligned}$$

$$\begin{aligned}
f &= \overline{\overline{x_1} \overline{x_2}} \\
&= (\overline{x_1} \overline{x_2} + \overline{x_2} \overline{x_3}) \\
&= (\overline{x_1} \overline{x_2})(\overline{x_2} \overline{x_3}) \\
&= (x_1 + x_2)(x_2 + \overline{x_3}) \\
&= x_2 + x_1 \overline{x_3}. \rightarrow \text{SOP}
\end{aligned}$$

## Exercises:

1. Determine whether or not the following expressions are valid, i.e., whether the left- and right-hand sides represent the same function.

(a)  $\bar{x}_1x_3 + x_1\bar{x}_2\bar{x}_3 + \bar{x}_1x_2 + x_1\bar{x}_2 = \bar{x}_2x_3 + x_1\bar{x}_3 + x_2\bar{x}_3 + \bar{x}_1x_2x_3$

(b)  $x_1\bar{x}_3 + x_2x_3 + \bar{x}_2\bar{x}_3 = (x_1 + \bar{x}_2 + x_3)(x_1 + x_2 + \bar{x}_3)(\bar{x}_1 + x_2 + \bar{x}_3)$

(c)  $(x_1 + x_3)(\bar{x}_1 + \bar{x}_2 + \bar{x}_3)(\bar{x}_1 + x_2) = (x_1 + x_2)(x_2 + x_3)(\bar{x}_1 + \bar{x}_3)$

2. Use algebraic manipulation to find the minimum sum-of-products expression for the function  $f = x_1x_3 + x_1\bar{x}_2 + \bar{x}_1x_2x_3 + \bar{x}_1\bar{x}_2\bar{x}_3$ .

Use algebraic manipulation to find the minimum sum-of-products expression for the function  $f = x_1x_3 + x_1\bar{x}_2 + \bar{x}_1x_2x_3 + \bar{x}_1\bar{x}_2\bar{x}_3$ .

$$\begin{aligned}f &= x_2'(x_1 + x_1'x_3') + x_3(x_1+x_1'x_2) \\&= x_2'(x_1 + x_3') + x_3(x_1+x_2) \quad (\text{Using } x+x'y = x + y) \\&= x_1x_2' + x_2'x_3' + x_1x_3 + x_2x_3 \\&= x_2'x_3' + x_2x_3 + x_1x_2' \\&\quad \text{or} \\&= x_2x_3 + x_1x_3 + x_2'x_3' \quad (\text{Using } xy + yz + x'z = xy + x'z)\end{aligned}$$

3. Use algebraic manipulation to find the minimum product-of-sums expression for the function  $f = (x_1 + x_2 + x_3) \cdot (\bar{x}_1 + \bar{x}_2 + x_3) \cdot (\bar{x}_1 + \bar{x}_2 + x_3) \cdot (\bar{x}_1 + x_2 + \bar{x}_3)$ .

$$(x_1+x_2)(x_2'+x_3)$$

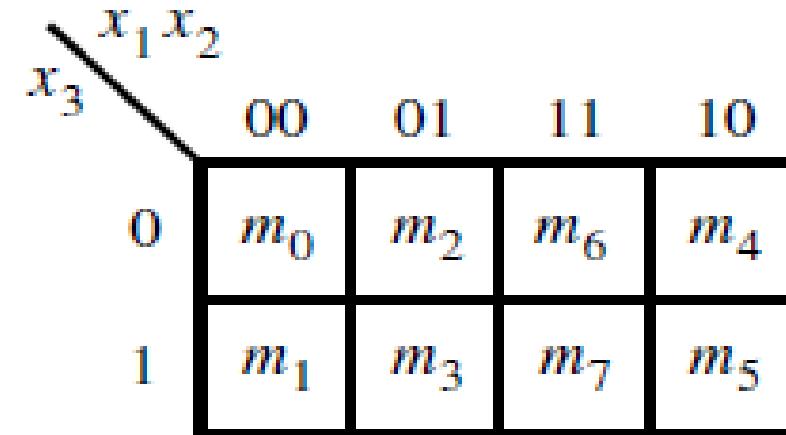
# Karnaugh map

- It is an alternative to the truth-table form for representing a function. The map consists of *cells* that correspond to the rows of the truth table.
- it allows easy recognition of minterms that can be combined

# Three-Variable Map

$x_1$	$x_2$	$x_3$	
0	0	0	$m_0$
0	0	1	$m_1$
0	1	0	$m_2$
0	1	1	$m_3$
1	0	0	$m_4$
1	0	1	$m_5$
1	1	0	$m_6$
1	1	1	$m_7$

(a) Truth table

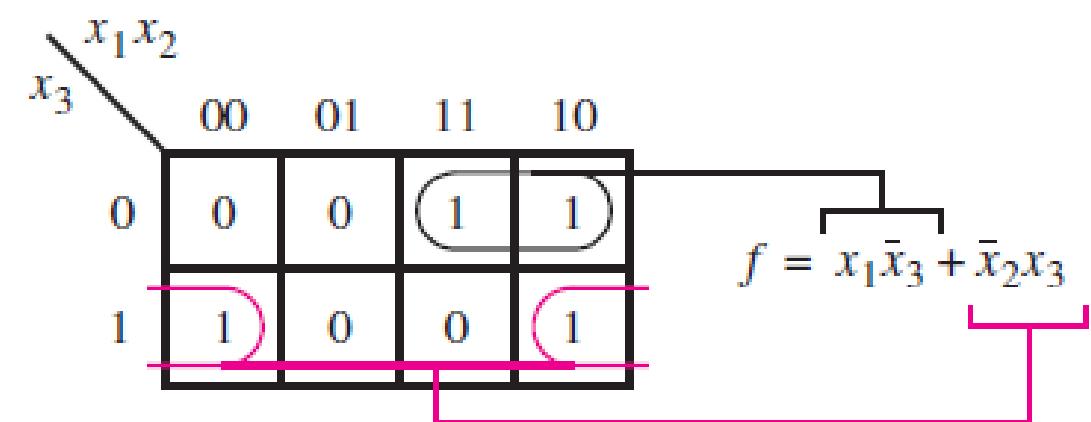


(b) Karnaugh map

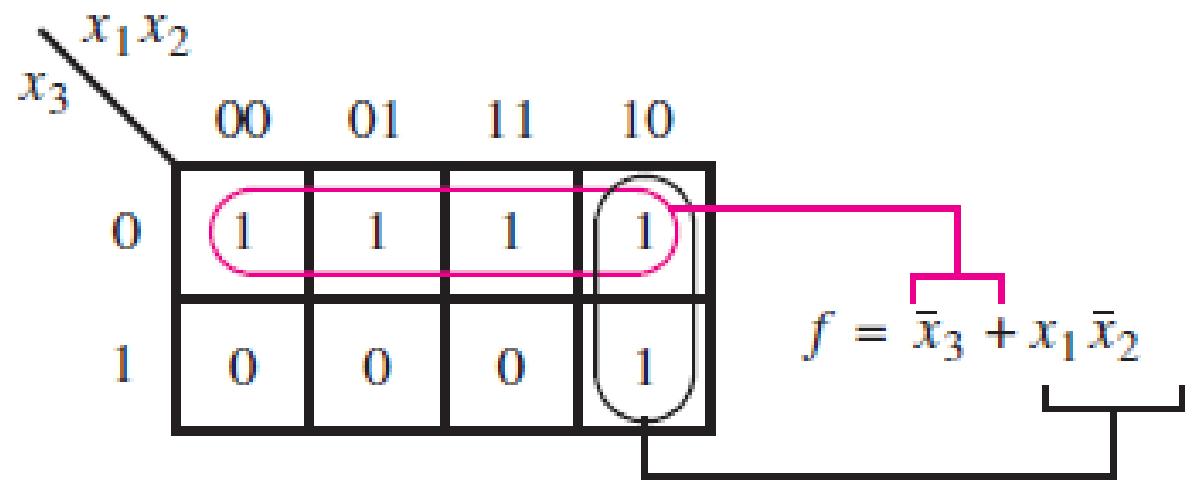
- Minterms in any two cells that are adjacent, either in the same row or the same column, can be combined.
- The adjacent cells must differ in the value of only one variable.
- Thus the columns are identified by the sequence of  $(x_1, x_2)$  values of 00, 01, 11, and 10.

- This makes the second and third columns different only in variable  $x_1$ . Also, the first and the fourth columns differ only in variable  $x_1$ , which means that these columns can be considered as being adjacent. (A sequence of codes, or valuations, where consecutive codes differ in one variable only is known as the Gray code.)

$x_1$	$x_2$	$x_3$	$f(x_1, x_2, x_3)$
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

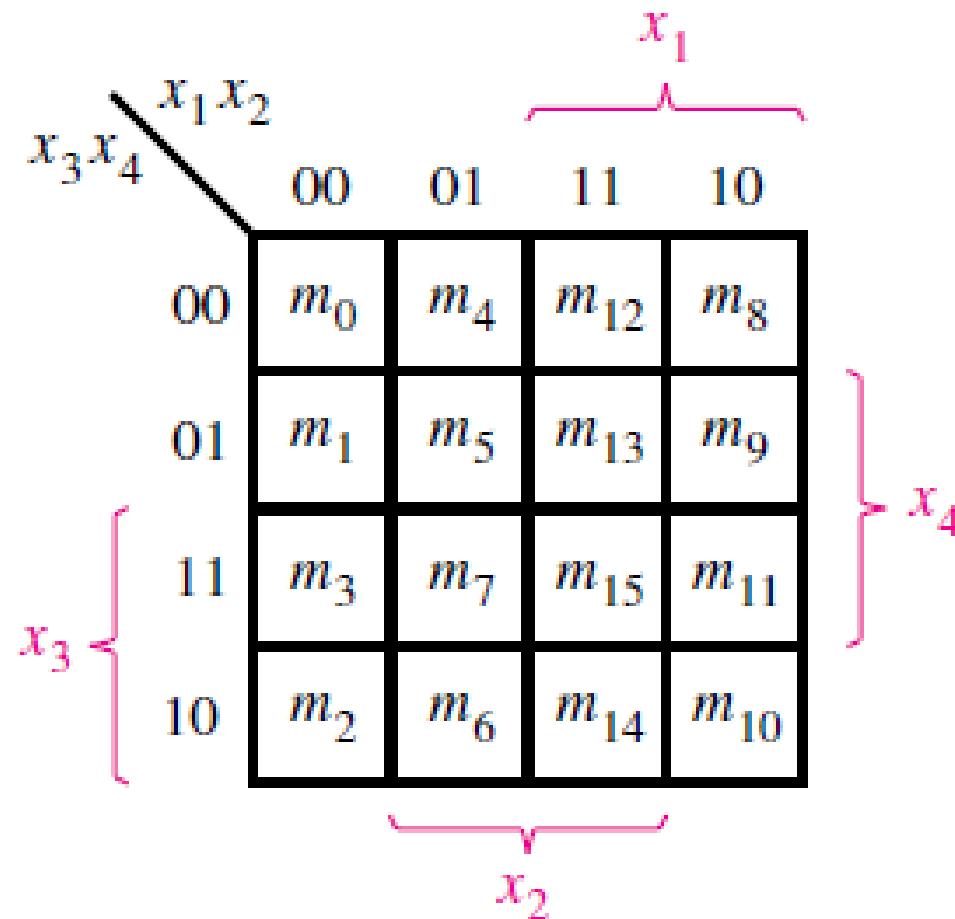


	$x_1$	$x_2$	$x_3$	$f$
1	0	0	0	1
2	0	0	1	0
3	0	1	0	1
4	0	1	1	0
5	1	0	0	1
6	1	0	1	1
7	1	1	0	1
8	1	1	1	0



$$f = \bar{x}_3 + x_1\bar{x}_2$$

# Four-Variable Map



$x_3 \ x_4$	$x_1 \ x_2$	00	01	11	10
00	0	0	0	0	0
01	0	0	1	1	
11	1	0	0	1	
10	1	0	0	1	

$f_1 = \bar{x}_2 x_3 + x_1 \bar{x}_3 x_4$

$x_3 x_4$	$x_1 x_2$	00	01	11	10
00		0	0	0	0
01		0	0	1	1
11		1	1	1	1
10		1	1	1	1

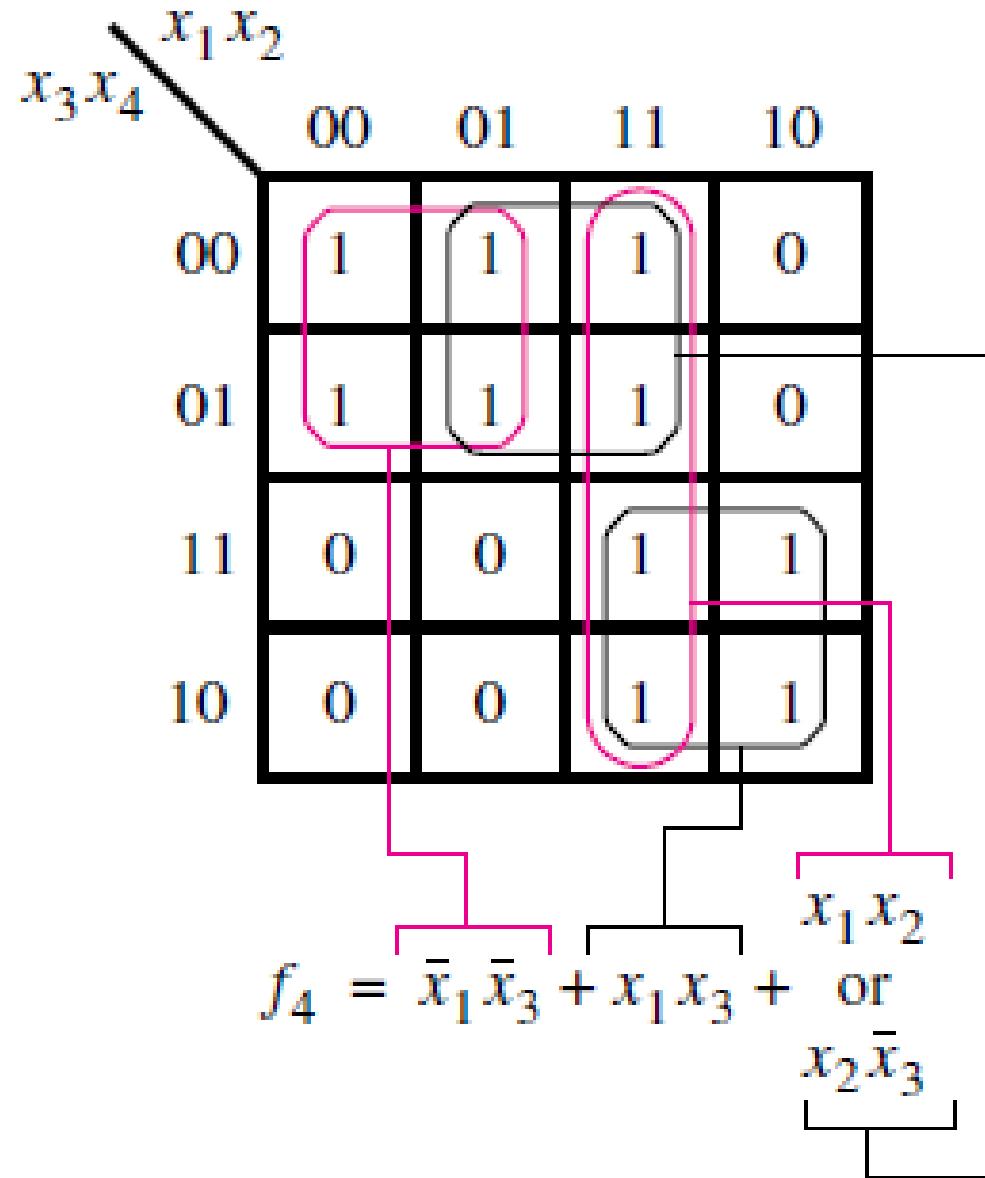
$f_2 = x_3 + x_1 x_4$

		x1, x2	00	01	11	10
		x3, x4	00	01	11	10
00	00	1	0	0	1	
01	01	0	0	0	0	
11	11	1	1	1	0	
10	10	1	1	0	1	

$x_1 x_2$	$x_3 x_4$	00	01	11	10
00	1	0	0	1	
01	0	0	0	0	
11	1	1	1	0	
10	1	1	0	1	

$$f_3 = \bar{x}_2 \bar{x}_4 + \bar{x}_1 x_3 + x_2 x_3 x_4$$

		00	01	11	10
		00	01	11	10
x3, x4	00	1	1	1	0
01	1	1	1	0	
11	0	0	1	1	
10	0	0	1	1	



# Five-Variable Map

x2, x3	00	01	11	10	
x4, x5	00	0	1	1	0
	01	0	1	1	0
	11	0	1	1	0
	10	0	1	1	0

$x_1=0$

x2, x3	00	01	11	10	
x4, x5	00	0	0	0	0
	01	1	0	0	0
	11	1	0	0	1
	10	1	0	0	1

$x_1=1$

$$f_1 = \bar{x}_1 x_3 + x_1 \bar{x}_3 x_4 + x_1 \bar{x}_2 \bar{x}_3 x_5$$

# Terminology

Some of the terminologies that are useful for describing the minimization process are:

## Literal

Each appearance of a variable, either uncomplemented or complemented, in a logical term is called a *literal*. For example, the product term  $x_1x_2x_3$  has three literals, and the sum term  $(x_1' + x_3 + x_4' + x_6)$  has four literals.

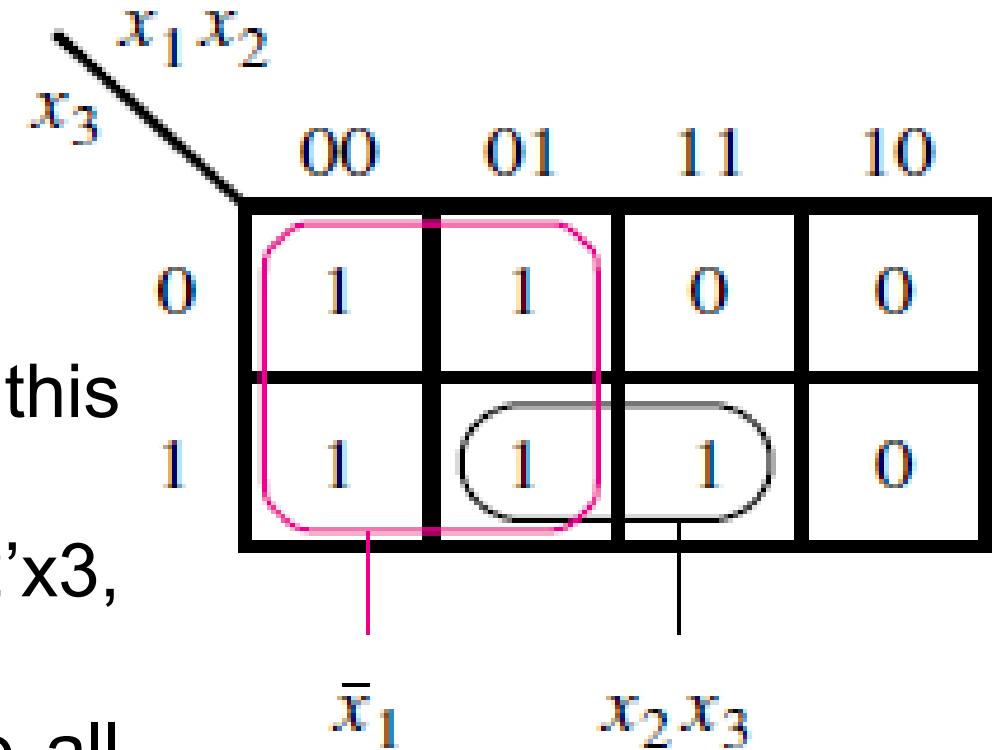
## **Implicant**

A product term that indicates the input valuation(s) for which a given function is equal to 1 is called an *implicant* of the function. The most basic implicants are the minterms.

## Example:

$$f(x_1, x_2, x_3) = \sum m(0, 1, 2, 3, 7).$$

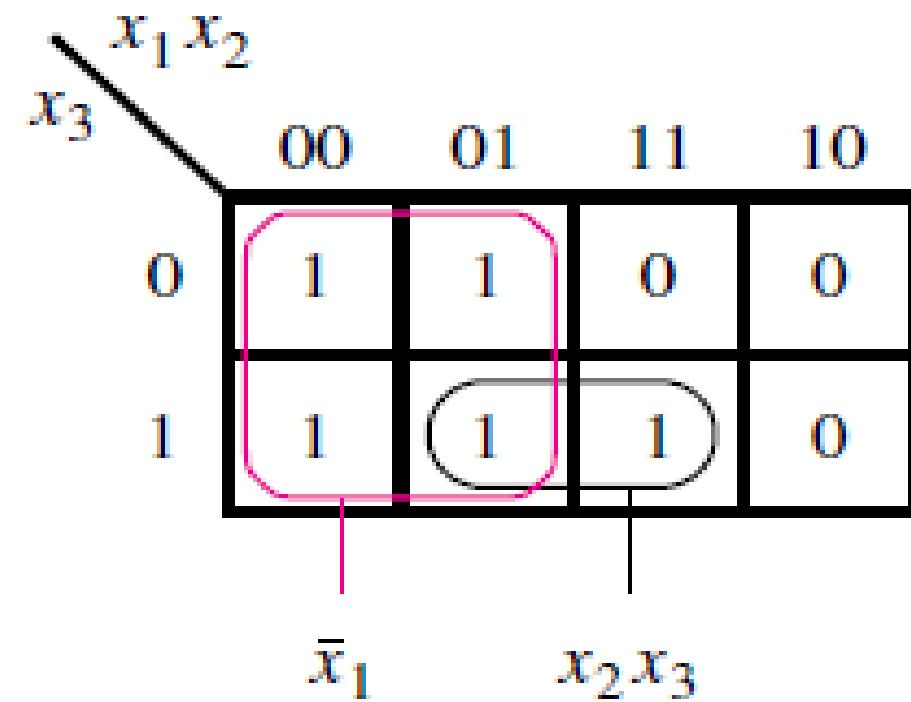
- There are 11 implicants for this function:
- five minterms:  $x_1'x_2'x_3'$ ,  $x_1'x_2'x_3$ ,  $x_1'x_2x_3'$ ,  $x_1'x_2x_3$ , and  $x_1x_2x_3$ .
- five implicants that correspond to all possible pairs of minterms:  $x_1'x_2'$  ( $m_0$  and  $m_1$ ),  $x_1'x_3'$  ( $m_0$  and  $m_2$ ),  $x_1'x_3$  ( $m_1$  and  $m_3$ ),  $x_1'x_2$  ( $m_2$  and  $m_3$ ), and  $x_2x_3$  ( $m_3$  and  $m_7$ ).
- one implicant that covers a group of four minterms:  $x_1'$ .



# Prime Implicant

An implicant is called a *prime implicant* if it cannot be combined into another implicant that has fewer literals.

Here there are two prime implicants:  
 $x_1'$  and  $x_2x_3$ .



## Cover

A collection of implicants that account for all valuations for which a given function is equal to 1 is called a *cover* of that function.

A number of different covers exist for most functions.

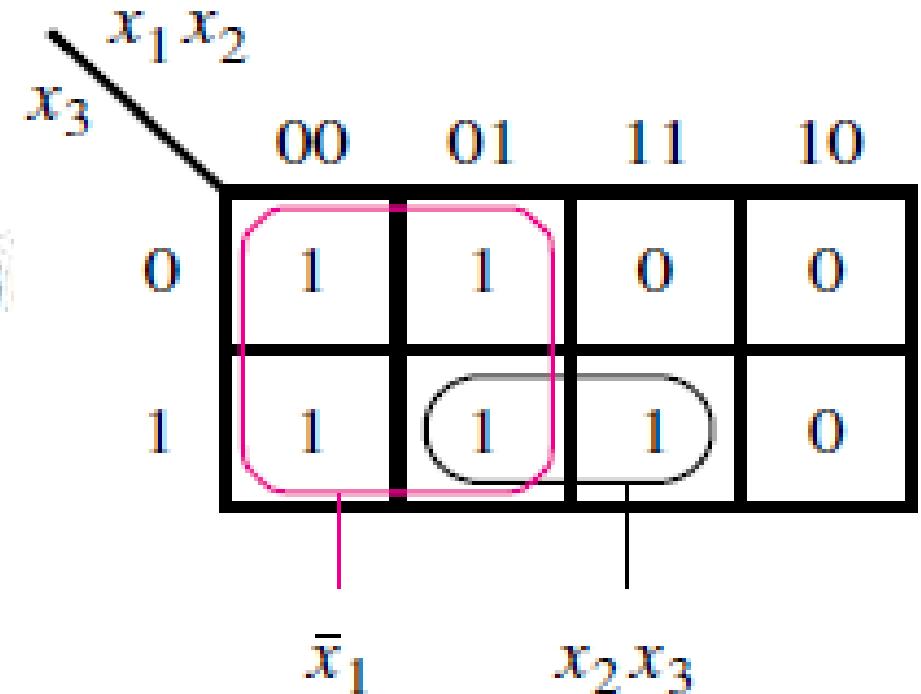
- A set of all minterms for which  $f = 1$  is a cover.
- A set of all prime implicants is a cover.

A cover defines a particular implementation of the function.

$$f = \bar{x}_1 \bar{x}_2 \bar{x}_3 + \bar{x}_1 \bar{x}_2 x_3 + \bar{x}_1 x_2 \bar{x}_3 + \bar{x}_1 x_2 x_3 + x_1 x_2 \bar{x}_3$$

$$f = \bar{x}_1 \bar{x}_2 + \bar{x}_1 x_2 + x_2 \bar{x}_3$$

$$f = \bar{x}_1 + x_2 \bar{x}_3$$



While all of these expressions represent the function  $f$  correctly, the cover consisting of prime implicants leads to the lowest-cost implementation.

# Cost

Cost of a logic circuit is the number of gates plus the total number of inputs to all gates in the circuit.

If we assume that the input variables are available in both true and complemented forms, then the cost of the expression,  $f = \underline{x}_1 \overline{x}_2 + \underline{x}_3 \overline{x}_4$  is 9. Otherwise its cost is 13.

If an inversion is needed inside a circuit, then the corresponding NOT gate and its input are included in the cost. For example, the expression

$$g = \overline{(x_1 \bar{x}_2 + x_3)} (\bar{x}_4 + x_5) \text{ has a cost of 14.}$$

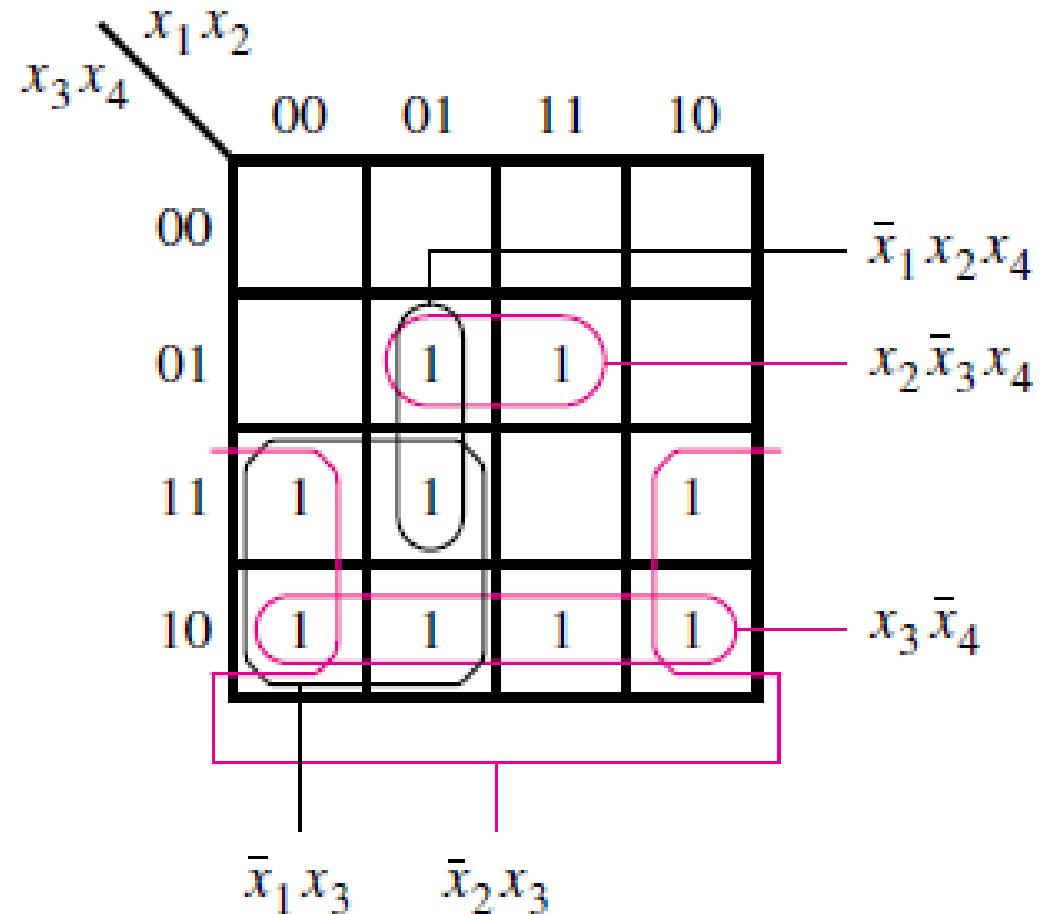
## Essential Prime Implicant

If a prime implicant includes a minterm for which  $f = 1$  that is not included in any other prime implicant, then it is called an *essential prime implicant*.

# Minimization Procedure

Consider the following examples in which there is a choice as which prime implicants to include in the final cover

- There are five prime implicants:  
 $x_1'x_3$ ,  $x_2'x_3$ ,  $x_3x_4'$ ,  $x_2x_3'x_4$ ,  
 $x_1'x_2x_4$ .
- The essential ones are  $x_2'x_3$ ,  
(because of  $m_{11}$ ),  $x_3x_4'$  (because of  
 $m_{14}$ ),  $x_2x_3'x_4$  (because of  $m_{13}$ ).
- They must be included in the cover.  
These three prime implicants cover  
all minterms for which  $f = 1$  except  
 $m_7$ . It is clear that  $m_7$  can be  
covered by either  $x_1'x_3$  or  $x_1'x_2x_4$



Because  $x_1'x_3$  has a lower cost, it is chosen for the cover. Therefore, the minimum-cost realization is

$$f = \bar{x}_2x_3 + x_2\bar{x}_4 + x_2\bar{x}_3x_4 + \bar{x}_1x_3$$

The process of finding a minimum-cost circuit involves the following steps:

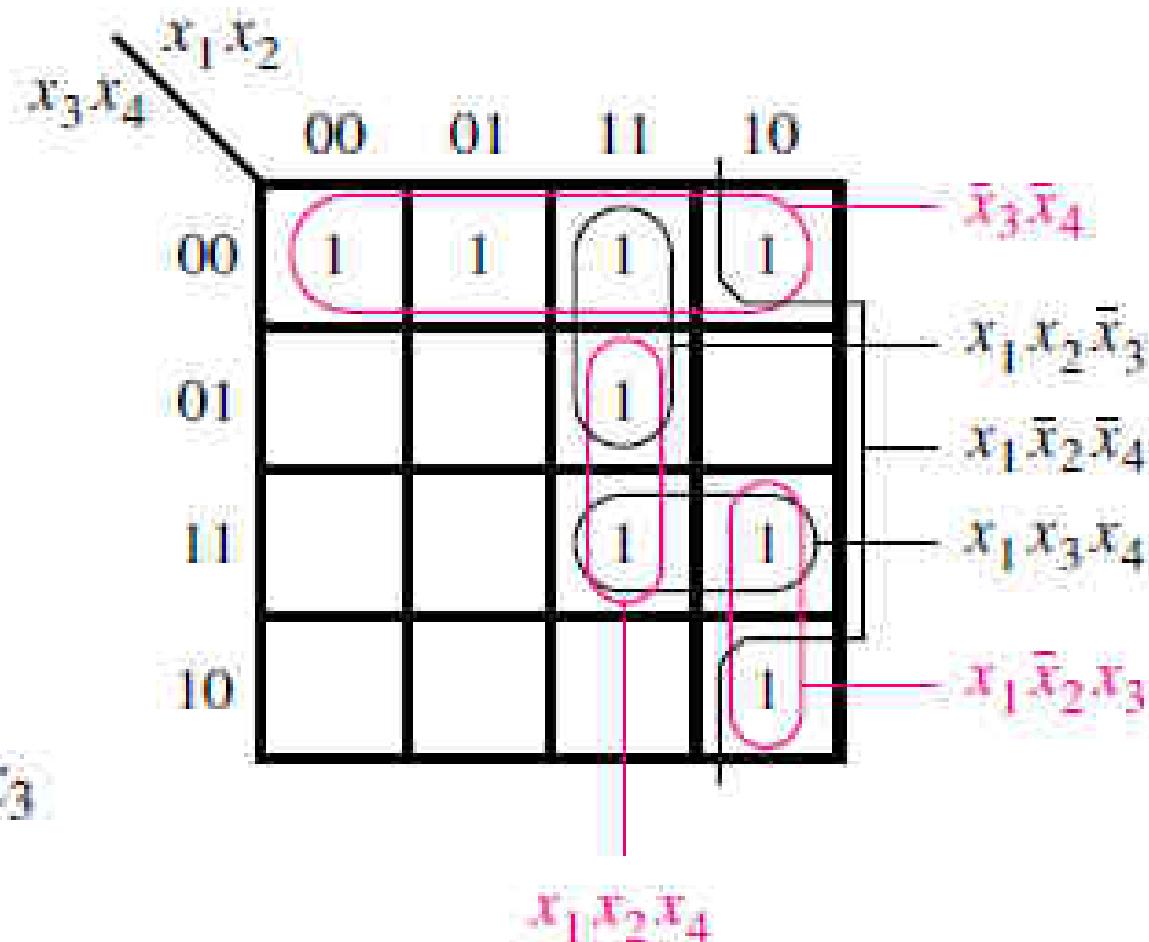
1. Generate all prime implicants for the given function  $f$ .
2. Find the set of essential prime implicants.
3. If the set of essential prime implicants covers all valuations for which  $f = 1$ , then this set is the desired cover of  $f$ . Otherwise, determine the nonessential prime implicants that should be added to form a complete minimum-cost cover.

Consider the following example:

only  $x_3'x_4'$  is essential.

Then the best choice  
to implement the  
minimum cost circuit  
is

$$f = \bar{x}_3\bar{x}_4 + x_1x_2x_4 + x_1\bar{x}_2x_3$$



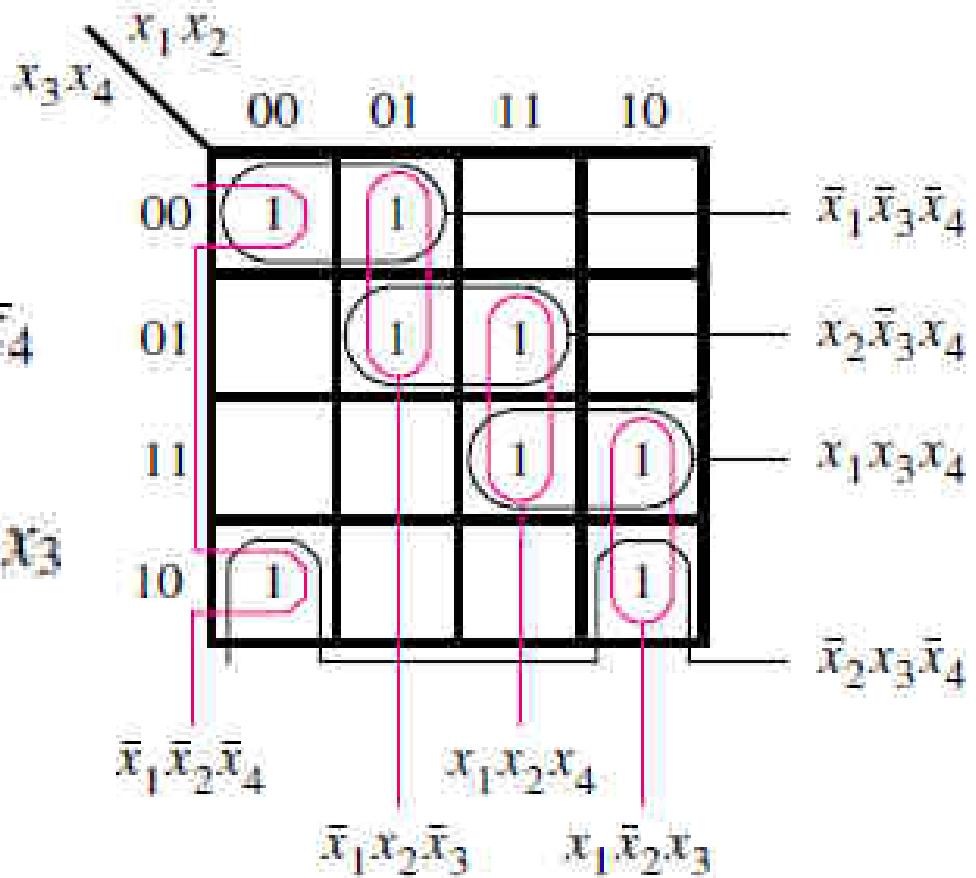
Sometimes there may not be any essential prime implicants at all.

Example:

Two alternatives:

$$f = \bar{x}_1 \bar{x}_3 \bar{x}_4 + x_2 \bar{x}_3 x_4 + x_1 x_3 x_4 + \bar{x}_2 x_3 \bar{x}_4$$

$$f = \bar{x}_1 \bar{x}_2 \bar{x}_4 + \bar{x}_1 x_2 \bar{x}_3 + x_1 x_2 x_4 + x_1 \bar{x}_2 x_3$$



# Minimization of Product-of-Sums Forms

Example1:

$$f = (\bar{x}_1 + x_2)(\bar{x}_1 + x_3)$$

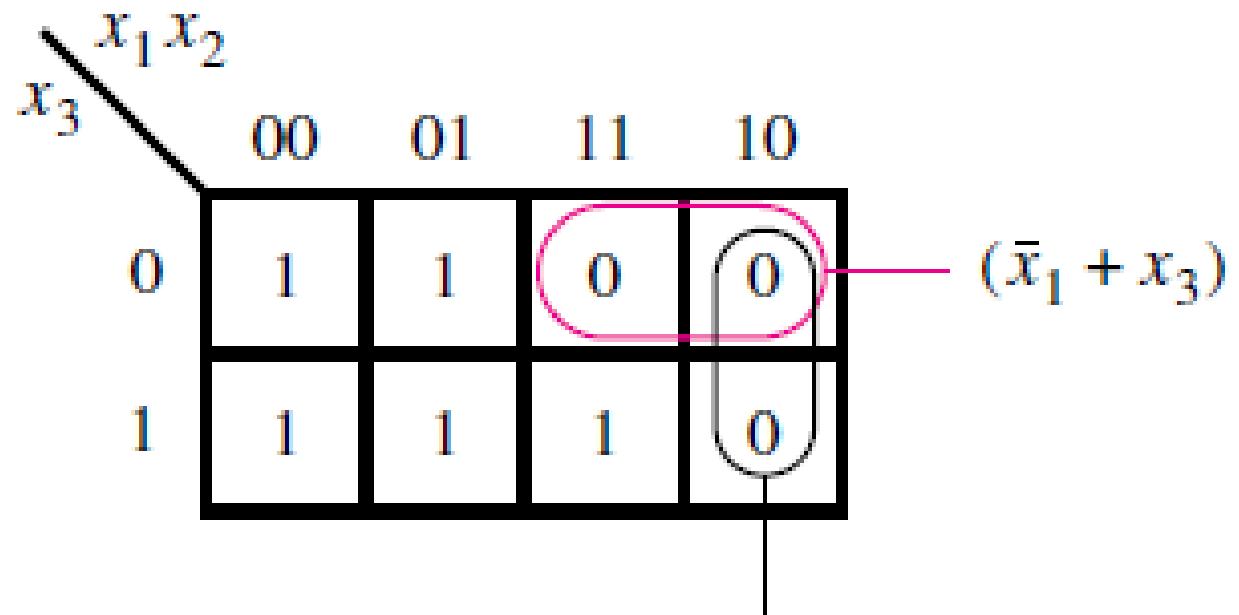
OR

$$\bar{f} = x_1 \bar{x}_2 + x_1 \bar{x}_3$$

$$f = \bar{\bar{f}} = \overline{x_1 \bar{x}_2 + x_1 \bar{x}_3}$$

$$= \overline{x_1 \bar{x}_2} \cdot \overline{x_1 \bar{x}_3}$$

$$= (\bar{x}_1 + x_2)(\bar{x}_1 + x_3)$$



Its cost is greater than the cost of the equivalent SOP implementation

$$f = \bar{x}_1 + x_2 x_3$$

SOP COST: 6 (assuming input variables are available in both true and complemented forms)

$$f = (\bar{x}_1 + x_2)(\bar{x}_1 + x_3)$$

POS COST: 9

Example2:

$$f = (x_2 + x_3)(x_3 + x_4)(\bar{x}_1 + \bar{x}_2 + \bar{x}_3 + \bar{x}_4)$$

OR

$$\bar{f} = \bar{x}_2\bar{x}_3 + \bar{x}_3\bar{x}_4 + x_1x_2x_3x_4$$

$$f = \bar{\bar{f}} = \overline{\bar{x}_2\bar{x}_3 + \bar{x}_3\bar{x}_4 + x_1x_2x_3x_4}$$

$$= \overline{\bar{x}_2\bar{x}_3} \cdot \overline{\bar{x}_3\bar{x}_4} \cdot \overline{x_1x_2x_3x_4}$$

$$= (x_2 + x_3)(x_3 + x_4)(\bar{x}_1 + \bar{x}_2 + \bar{x}_3 + \bar{x}_4)$$

$x_3$	$x_4$	$x_1$	$x_2$	
00	01	11	10	
00	0	0	0	$(x_3 + x_4)$
01	0	1	1	$(x_2 + x_3)$
11	1	1	0	
10	1	1	1	$(\bar{x}_1 + \bar{x}_2 + \bar{x}_3 + \bar{x}_4)$

Assuming that both the complemented and uncomplemented versions of the input variables  $x_1$  to  $x_4$  are available,

$$f = \bar{x}_2x_3 + x_3\bar{x}_4 + x_2\bar{x}_3x_4 + \bar{x}_1x_3 \quad \text{SOP COST: 18}$$

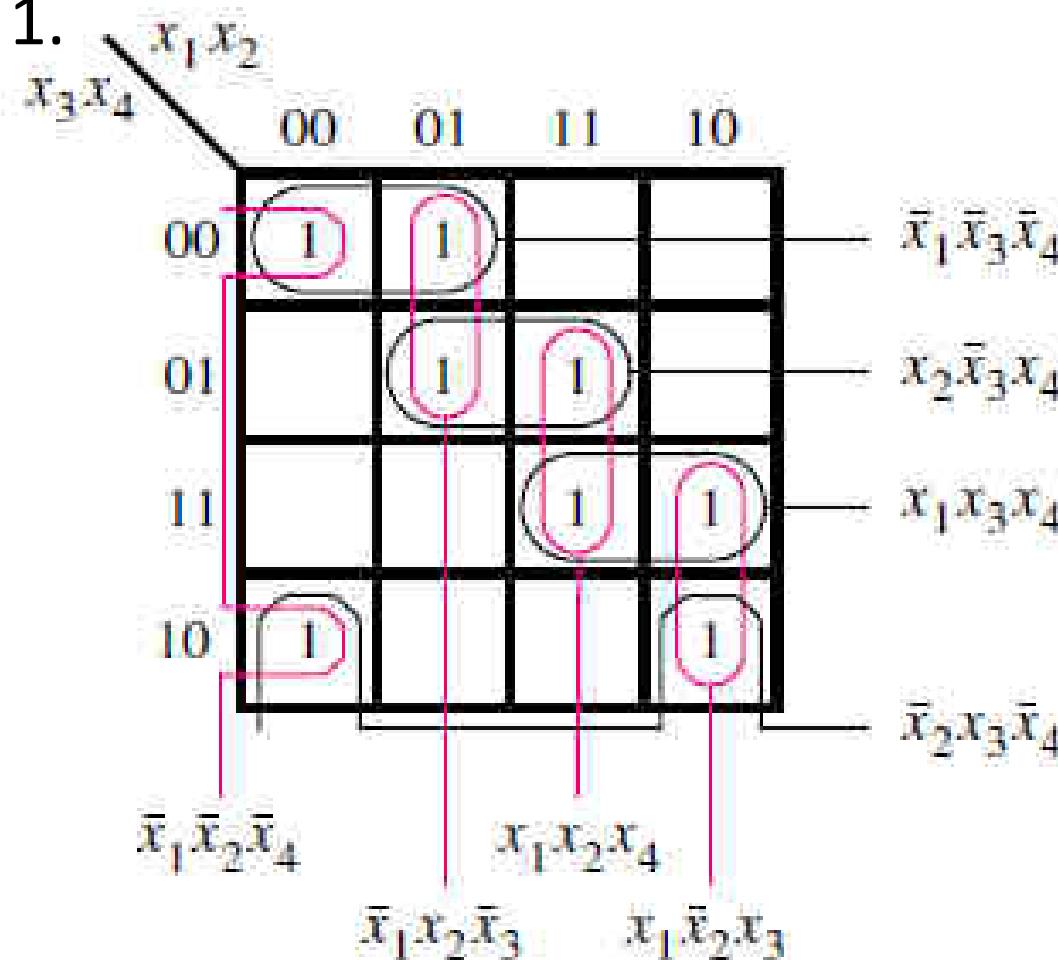
$$f = (x_2 + x_3)(x_3 + x_4)(\bar{x}_1 + \bar{x}_2 + \bar{x}_3 + \bar{x}_4) \quad \text{POS COST: 15}$$

SOP and POS implementations of a given function may or may not entail the same cost.

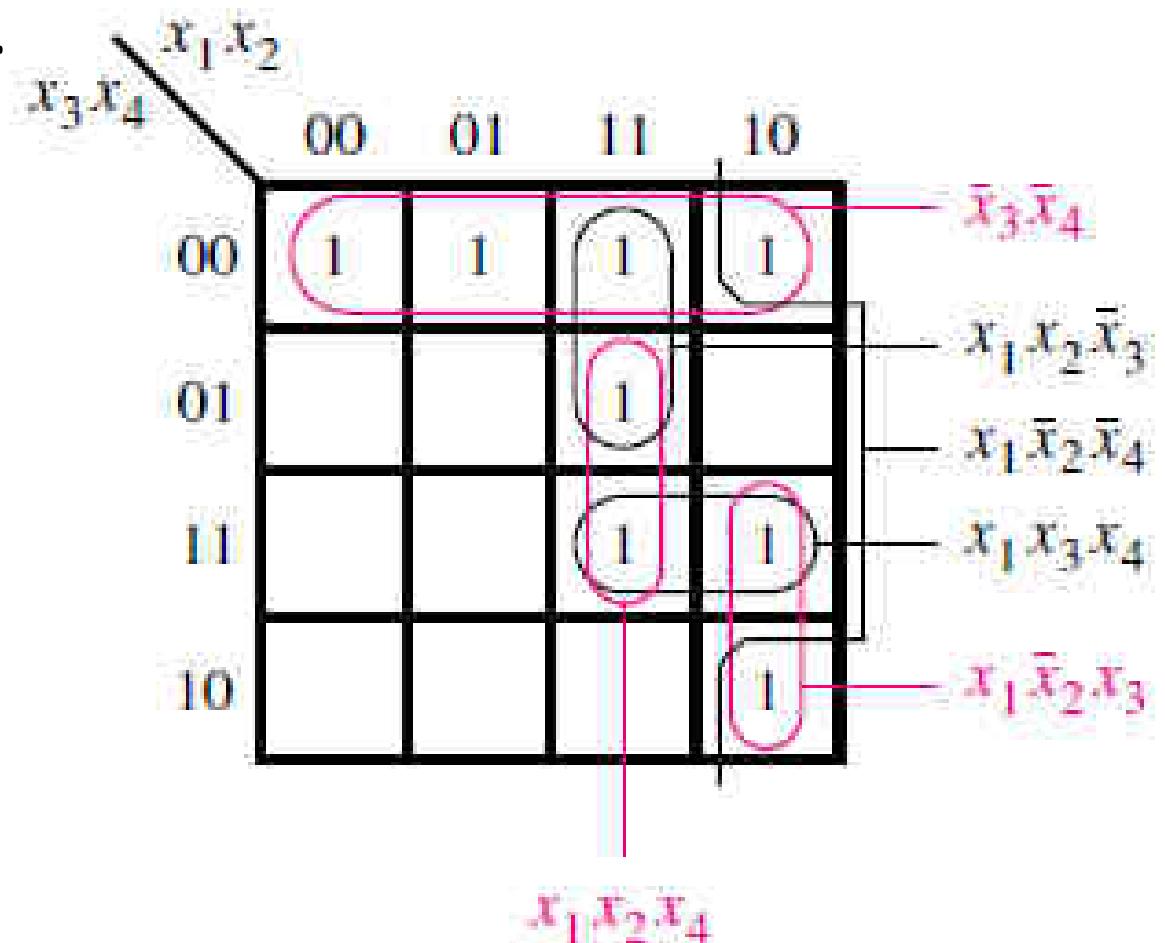
## EXERCISES:

Find the POS implementation for the following and compare the cost with the SOP form

1.



2.



1. SOP:  $f = \bar{x}_1\bar{x}_3\bar{x}_4 + x_2\bar{x}_3x_4 + x_1x_3x_4 + \bar{x}_2x_3\bar{x}_4$   
OR

$$f = \bar{x}_1\bar{x}_2\bar{x}_4 + \bar{x}_1x_2\bar{x}_3 + x_1x_2x_4 + x_1\bar{x}_2x_3$$

Cost: 3 input AND gate: 4 → 12+4=16

4 input OR gate: 1 → 4+1=5  
cost = 21

POS:

$$(x_1' + x_2 + x_3)(x_1' + x_2' + x_4)(x_1 + x_2' + x_3')(x_1 + x_2 + x_4')$$

OR

$$(x_1' + x_3 + x_4)(x_2 + x_3 + x_4')(x_1 + x_3' + x_4')(x_2' + x_3' + x_4)$$

Cost: 3 input OR gate: 4 → 12+4=16

4 input AND gate: 1 → 4+1=5  
cost = 21

2. SOP:  $f = \overline{x_1}\overline{x_4} + x_1x_2x_4 + x_1\overline{x_2}x_3$

cost: 2 input AND gate  $1 \rightarrow 2+1 = 3$

3 input AND gate  $2 \rightarrow 6+2 = 8$

3 input OR gate  $1 \rightarrow 3+1 = 4$

cost = 15

POS:

$$(x_1+x_4')(x_1+x_3')(x_2+x_3+x_4')(x_2'+x_3'+x_4)$$

Cost: 19

1. Obtain the simplest SOP and POS for  $f(x_1, \dots, x_5) = \prod M(1, 4, 6, 7, 9, 12, 15, 17, 20, 21, 22, 23, 28, 31)$ .

$x_2, x_3$	00	01	11	10
$x_4, x_5$	00	01	11	10
00	1	0	0	1
01	0	1	1	0
11	1	0	0	1
10	1	0	1	1

$x_1=0$

$x_2, x_3$	00	01	11	10
$x_4, x_5$	00	01	11	10
00	1	0	0	1
01	0	0	1	1
11	1	0	0	1
10	1	0	1	1

$x_1=1$

SOP form:  $f = \bar{x}_3\bar{x}_5 + \bar{x}_3x_4 + x_1x_4\bar{x}_5 + \bar{x}_1x_3\bar{x}_4x_5 + x_1x_2\bar{x}_4x_5$

POS form:  $f = (\bar{x}_3 + x_4 + x_5)(\bar{x}_3 + \bar{x}_4 + \bar{x}_5)(x_2 + \bar{x}_3 + \bar{x}_4)(x_1 + x_3 + x_4 + \bar{x}_5)(\bar{x}_1 + x_2 + x_4 + \bar{x}_5)$

# Incompletely Specified Functions

In digital systems it often happens that certain input conditions can never occur. For example, suppose that  $x_1$  and  $x_2$  control two interlocked switches such that both switches cannot be closed at the same time. Then the input valuations  $(x_1, x_2) = 11$  is guaranteed not to occur. Then we say that  $(x_1, x_2) = 11$  is a don't-care condition, meaning that a circuit with  $x_1$  and  $x_2$  as inputs can be designed by ignoring this condition. A function that has don't-care condition(s) is said to be incompletely specified.

$$f(x_1, \dots, x_4) = \sum m(2, 4, 5, 6, 10) + D(12, 13, 14, 15)$$

Truth table for  $f(x_1, \dots, x_4)$ :

	$x_1 x_2$	$x_3 x_4$		
	00	01	11	10
00	0	1	d	0
01	0	1	d	0
11	0	0	d	0
10	1	1	d	1

Annotations:

- Inputs  $x_1 x_2$  and  $x_3 x_4$  are shown at the top left.
- Outputs  $x_2 \bar{x}_3$  and  $x_3 \bar{x}_4$  are shown at the bottom right.
- Cells containing 1 or d are highlighted with pink circles.
- Cells containing 0 are highlighted with pink rectangles.

$$f = x_2 \bar{x}_3 + x_3 \bar{x}_4$$

Truth table for  $f(x_1, \dots, x_4)$ :

	$x_1 x_2$	$x_3 x_4$		
	00	01	11	10
00	0	1	d	0
01	0	1	d	0
11	0	0	d	0
10	1	1	d	1

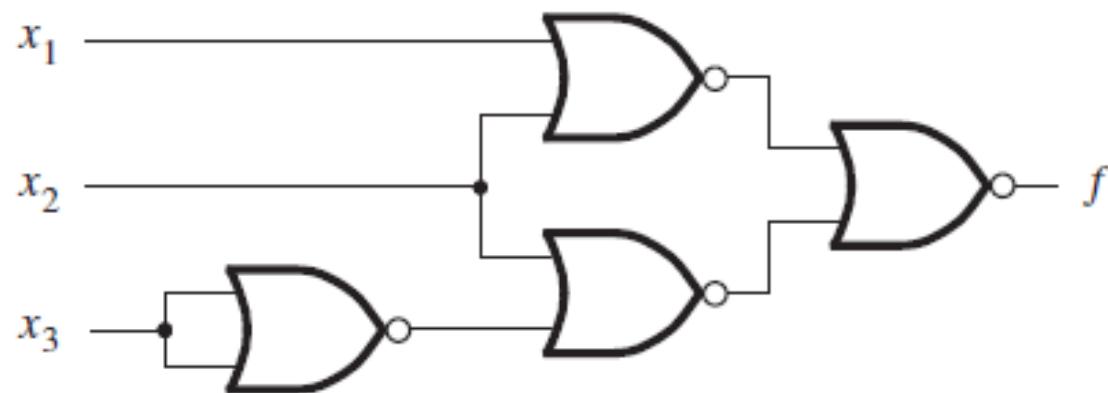
Annotations:

- Inputs  $x_1 x_2$  and  $x_3 x_4$  are shown at the top left.
- Outputs  $(x_2 + x_3)$  and  $(\bar{x}_3 + \bar{x}_4)$  are shown at the bottom right.
- Cells containing 1 or d are highlighted with pink circles.
- Cells containing 0 are highlighted with pink rectangles.

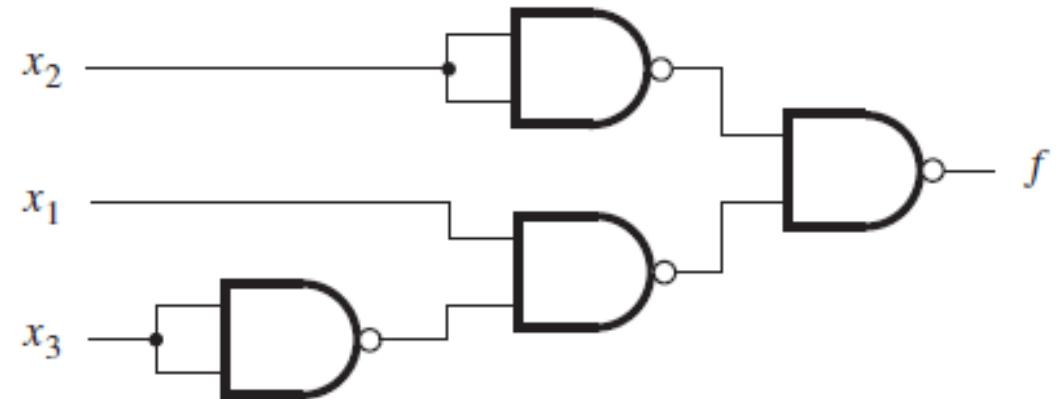
$$f = (x_2 + x_3)(\bar{x}_3 + \bar{x}_4)$$

# NAND and NOR Implementation

$$f = (x_1 + x_2)(x_2 + \bar{x}_3)$$



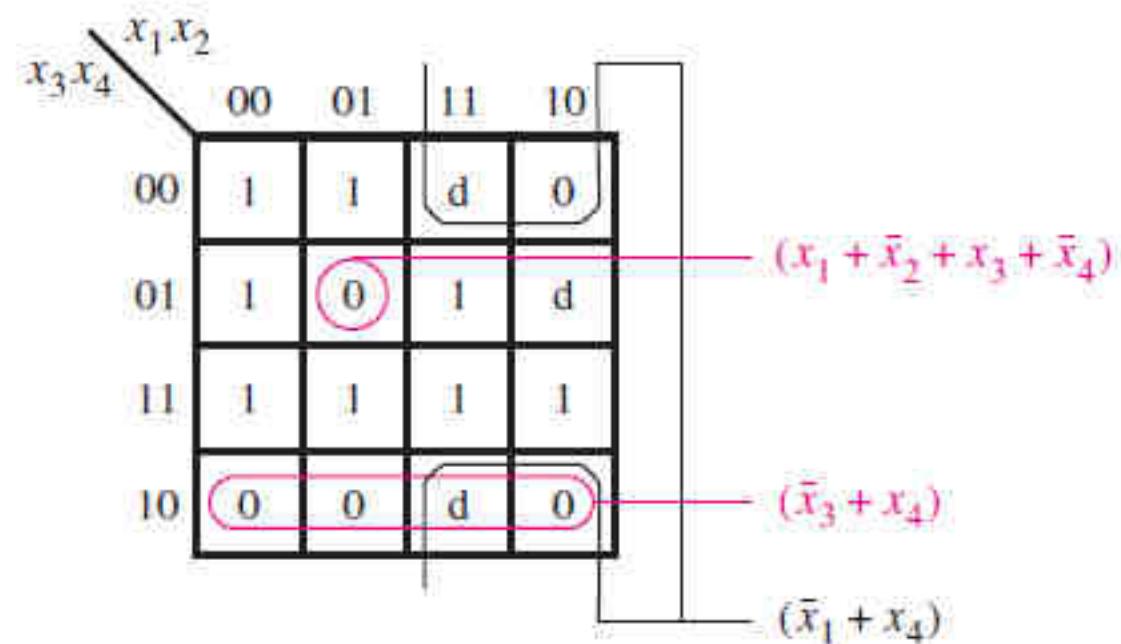
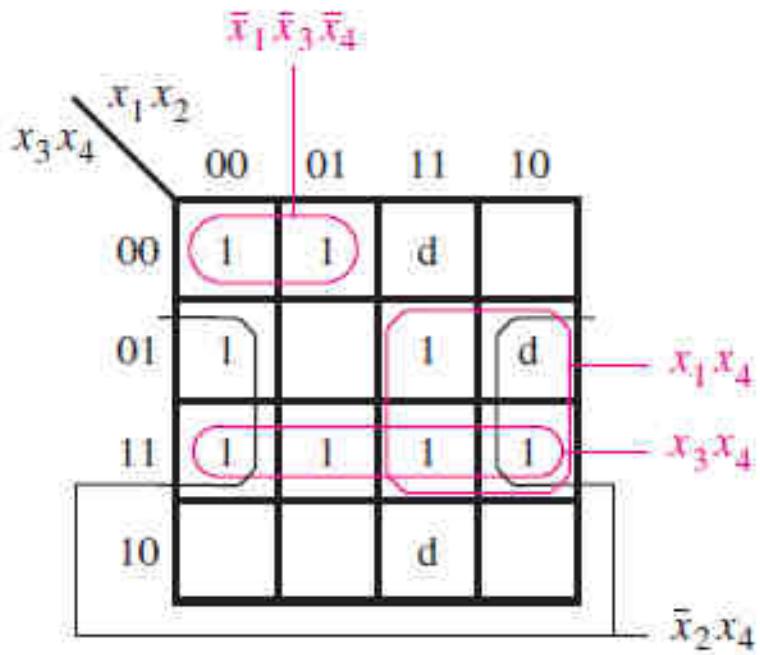
$$f = x_2 + x_1 \bar{x}_3$$



Use Karnaugh maps to find the minimum-cost SOP and POS expressions for the function

$$f(x_1, \dots, x_4) = x_1'x_3'x_4' + x_3x_4 + x_1'x_2'x_4 + x_1x_2x_3'x_4$$

assuming that there are also don't-cares defined as  $D = \Sigma(9, 12, 14)$ .



$$f = x_3x_4 + \bar{x}_1\bar{x}_3\bar{x}_4 + \bar{x}_2x_4 + x_1x_4$$

$$f = (\bar{x}_3 + x_4)(\bar{x}_1 + x_4)(x_1 + \bar{x}_2 + x_3 + \bar{x}_4)$$

A three-variable logic function that is equal to 1 if any two or all three of its variables are equal to 1 is called a *majority* function. Design a minimum-cost SOP circuit that implements this majority function.

a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

# Multilevel Synthesis

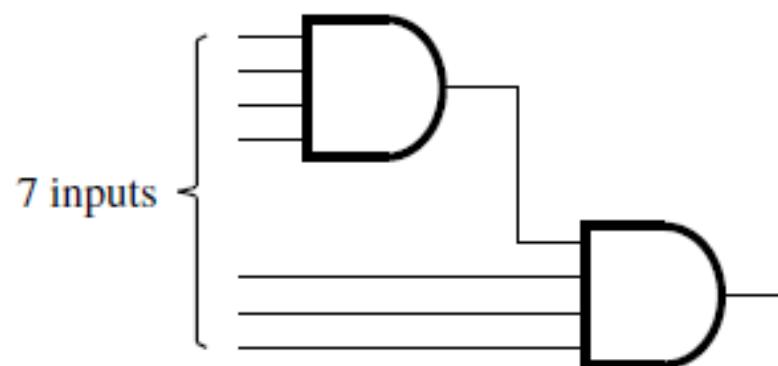
$$f(x_1, \dots, x_7) = x_1x_3\bar{x}_6 + x_1x_4x_5\bar{x}_6 + x_2x_3x_7 + x_2x_4x_5x_7$$

## Factoring

$$\begin{aligned} f &= x_1\bar{x}_6(x_3 + x_4x_5) + x_2x_7(x_3 + x_4x_5) \\ &= (x_1\bar{x}_6 + x_2x_7)(x_3 + x_4x_5) \end{aligned}$$

## Fan-in Problem

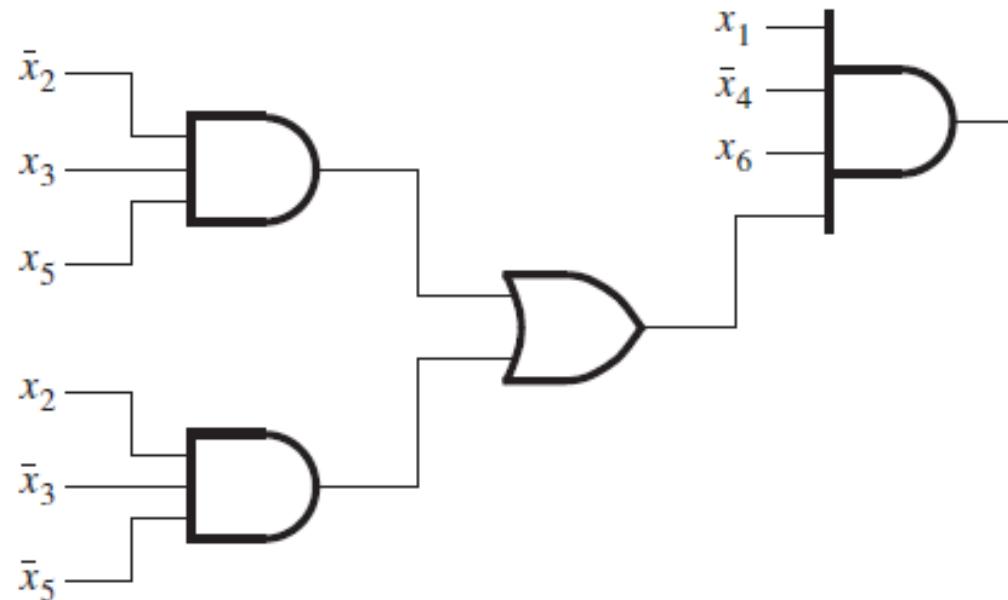
No. of inputs that a gate can drive is called fan in



Factoring can be used to deal with the fan-in problem.

$$\bar{f} = \bar{x}_1 \bar{x}_2 x_3 \bar{x}_4 x_5 x_6 + x_1 \bar{x}_2 \bar{x}_3 \bar{x}_4 \bar{x}_5 x_6$$

$$f = x_1 \bar{x}_4 x_6 (\bar{x}_2 x_3 x_5 + \bar{x}_2 \bar{x}_3 \bar{x}_5)$$



# Functional Decomposition

- Using factoring, multilevel circuits are used to deal with fan-in limitations. However, such circuits may be preferable even if fan-in is not a problem. In some cases the multilevel circuits may reduce the cost of implementation. On the other hand, they usually imply longer propagation delays, because they use multiple stages of logic gates.
- Complexity of a logic circuit can often be reduced by decomposing a two-level circuit into subcircuits, where one or more subcircuits implement functions that may be used in several places to construct the final circuit

## Example 1

	$x_1 x_2$	$x_3 x_4$	00	01	11	10	
00	0	1	1	0			$x_2 \bar{x}_3 \bar{x}_4$
01	0	0	1	1			$x_1 \bar{x}_3 x_4$
11	0	1	1	0			$x_2 \bar{x}_3 x_4$
10	0	0	1	1			$x_1 \bar{x}_3 \bar{x}_4$

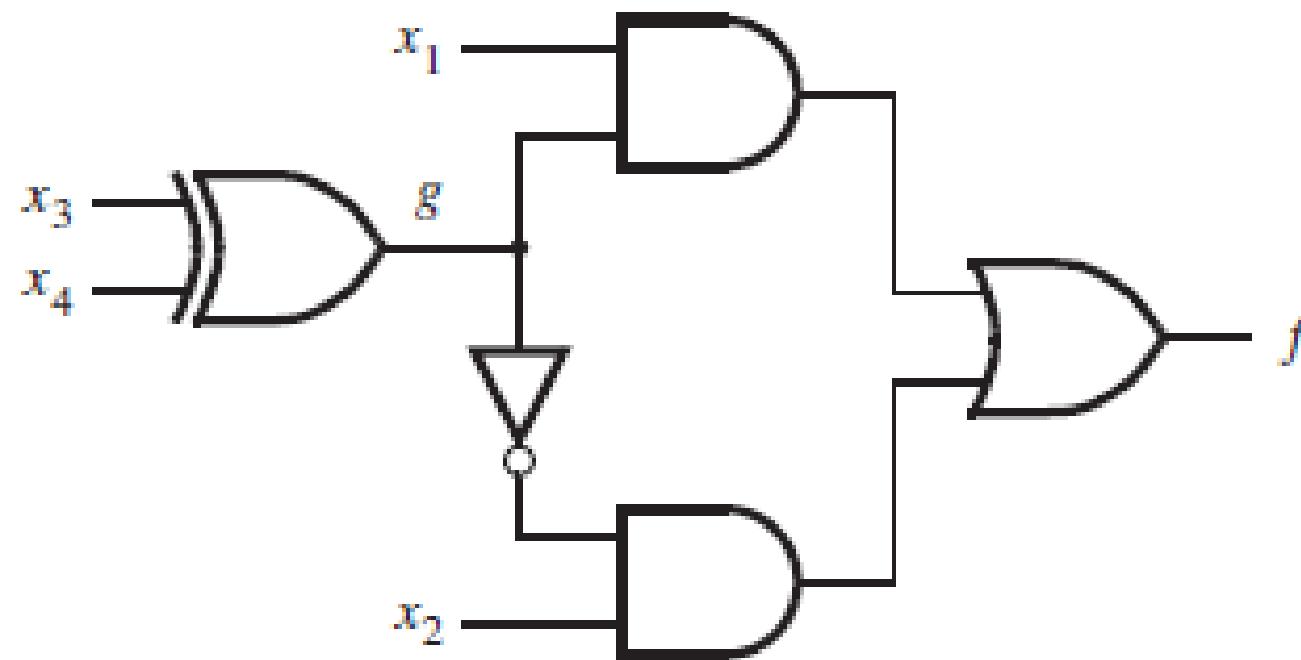
$$f = x_1 \bar{x}_3 x_4 + x_1 x_3 \bar{x}_4 + x_2 \bar{x}_3 \bar{x}_4 + x_2 x_3 \bar{x}_4$$

$$f = x_1 (\bar{x}_3 x_4 + x_3 \bar{x}_4) + x_2 (\bar{x}_3 \bar{x}_4 + x_3 \bar{x}_4)$$

- Assuming that inputs are available only in their true form, this expression has seven gates and 18 inputs to gates, for a total cost of 25.
- To perform functional decomposition we need to find one or more subfunctions that depend on only a subset of the input variables.

Now let  $g(x_3, x_4) = x_3'x_4 + x_3x_4'$  and observe that  $g' = x_3'x_4' + x_3x_4$ . The decomposed function  $f$  can be written as

$$f = x_1g + x_2g'$$



## Example 2

$$f = x_1 \bar{x}_3 x_4 + x_1 x_3 \bar{x}_4 + x_2 \bar{x}_3 x_4 + x_2 x_3 \bar{x}_4 + \bar{x}_3 x_4 x_5 + x_3 \bar{x}_4 x_5 + x_1 \bar{x}_2 \bar{x}_3 \bar{x}_4 x_5 + \bar{x}_1 \bar{x}_2 x_3 x_4 \bar{x}_5$$

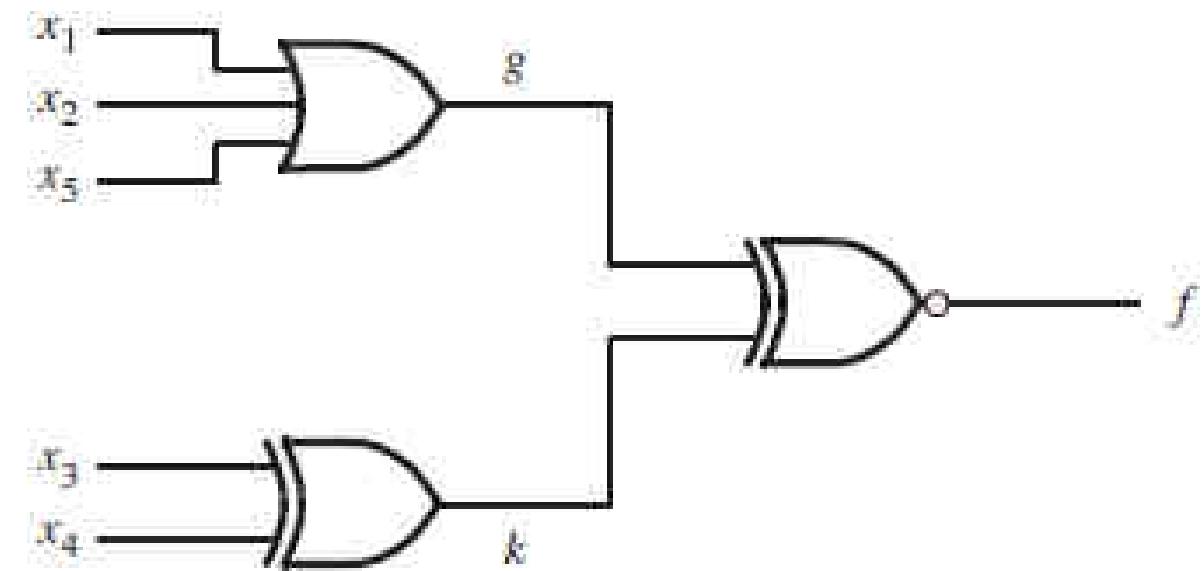
$$= x_3'x_4(x_1+x_2+x_5) + x_3x_4'(x_1+x_2+x_5) + x_1'x_2'x_5'(x_3'x_4'+x_3x_4)$$

$$= (x_1+x_2+x_5)(x_3'x_4+x_3x_4') + x_1'x_2'x_5'(x_3'x_4'+x_3x_4)$$

let  $g=x_1+x_2+x_5$ , and  $k= x_3'x_4+x_3x_4'$ . Then

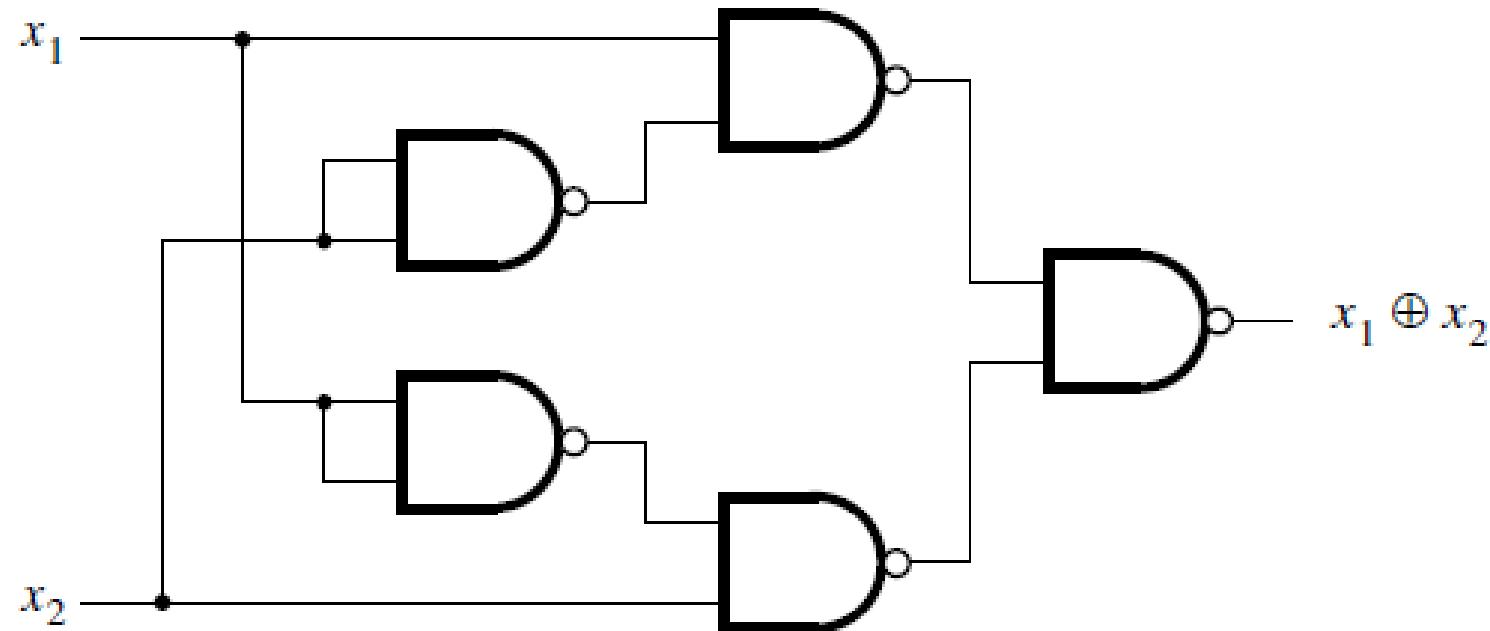
$$f= gk +g'k' = (g \text{ XOR } k)'$$

- It requires a total of three gates and seven inputs to gates, for a total cost of 10.
- The largest fan-in of any gate is three.
- The minimum cost SOP has a cost of 55 (14 gates and 41 inputs) with largest fan in 8



Implement the XOR function using only NAND gates.

$$x_1 \text{ XOR } x_2 = x_1'x_2 + x_1x_2'$$



Exploit functional decomposition to find a better implementation of XOR using only NAND gates. Let the symbol  $\uparrow$  represent the NAND operation so that  $x_1 \uparrow x_2 = (x_1 \cdot x_2)'$ .

$$x_1 \oplus x_2 = x_1 \bar{x}_2 + \bar{x}_1 x_2$$

$$x_1 \oplus x_2 = (x_1 \uparrow \bar{x}_2) \uparrow (\bar{x}_1 \uparrow x_2)$$

To find a decomposition, we can manipulate the term  $(x_1 \uparrow x_2')$  as follows:

$$(x_1 \uparrow \bar{x}_2) = \overline{(x_1 \bar{x}_2)} = \overline{(x_1(\bar{x}_1 + \bar{x}_2))} = (x_1 \uparrow (\bar{x}_1 + \bar{x}_2))$$

We can perform a similar manipulation for  $(x_1' \uparrow x_2)$

$$(x_1' \uparrow x_2) = (x_1' x_2)' = (x_2(x_1' + x_2'))' = (x_2 \uparrow (x_1' + x_2'))$$

$$x_1 \oplus x_2 = (x_1 \uparrow (\bar{x}_1 + \bar{x}_2)) \uparrow ((\bar{x}_1 + \bar{x}_2) \uparrow x_2)$$

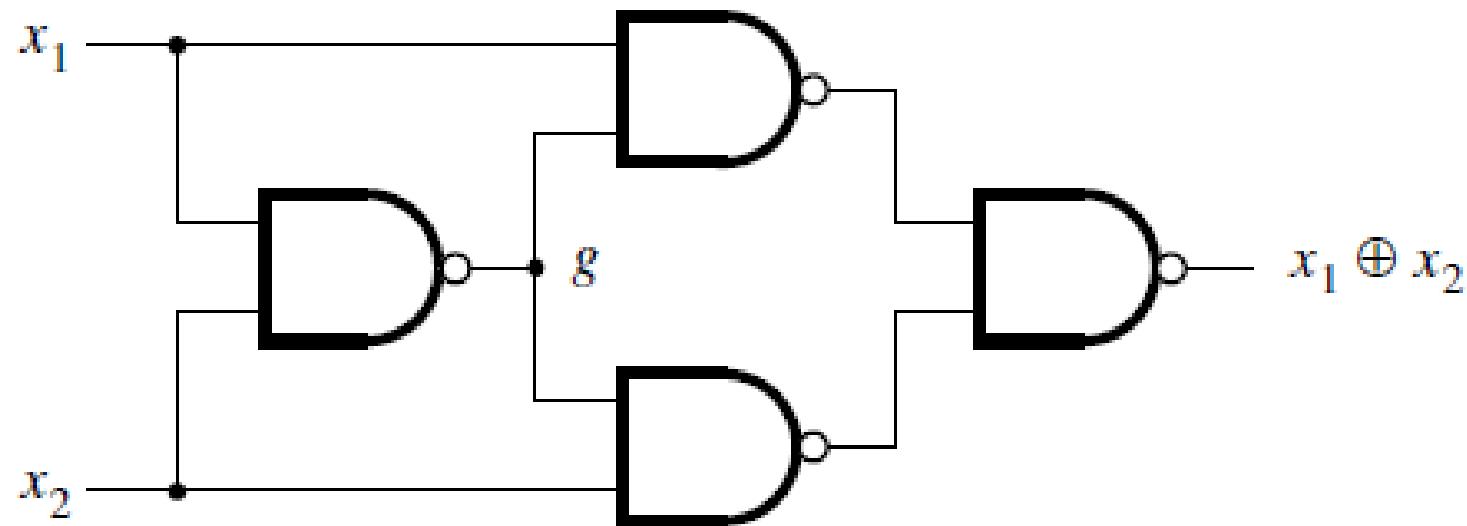
DeMorgan's theorem states that  $x_1' + x_2' = x_1 \uparrow x_2$ ;  
hence we can write

$$\bar{x}_1 \oplus \bar{x}_2 = (\bar{x}_1 \uparrow (x_1 \uparrow x_2)) \uparrow ((x_1 \uparrow x_2) \uparrow x_2)$$

Now we have a decomposition

$$x_1 \oplus x_2 = (x_1 \uparrow g) \uparrow (g \uparrow x_2)$$

$$g = x_1 \uparrow x_2$$



Optimal NAND gate implementation

Find the minimum-cost circuit for the function  $f(x_1, \dots, x_4) = m(0, 4, 8, 13, 14, 15)$ .

Assume that the input variables are available in uncomplemented form only.

$$f = p \cdot h + \bar{g} \cdot \bar{h}, \text{ where } p = \overline{x_1 x_2 x_3 x_4} \quad h = x_3 + x_4$$

# ARITHMETIC CIRCUITS

## Addition of Unsigned Numbers

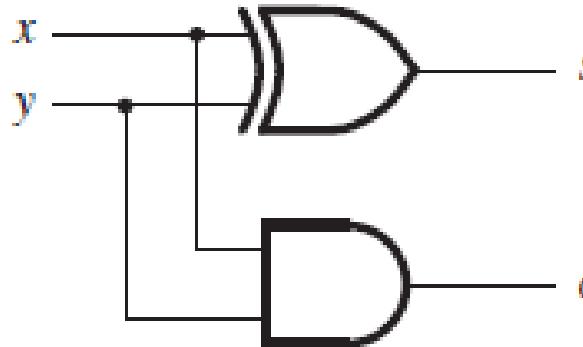
The one-bit addition entails four possible combinations,

$$\begin{array}{r} x \\ + y \\ \hline c \ s \end{array} \quad \begin{array}{r} 0 \\ + 0 \\ \hline 0 \ 0 \end{array} \quad \begin{array}{r} 0 \\ + 1 \\ \hline 0 \ 1 \end{array} \quad \begin{array}{r} 1 \\ + 0 \\ \hline 0 \ 1 \end{array} \quad \begin{array}{r} 1 \\ + 1 \\ \hline 1 \ 0 \end{array}$$

Carry      Sum

		Carry	Sum
$x$	$y$	$c$	$s$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(b) Truth table



(c) Circuit



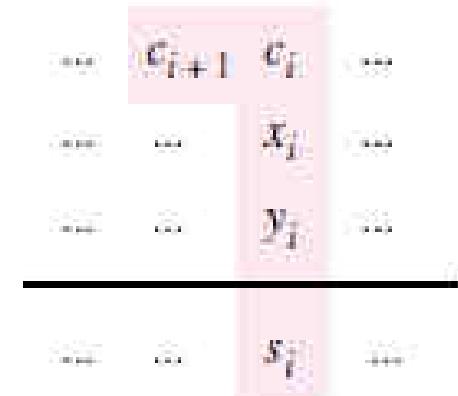
(d) Graphical symbol

The sum bit  $s$  is the XOR function. The carry  $c$  is the AND function of inputs  $x$  and  $y$ . A circuit realization of these functions is shown in Fig. c. This circuit, which implements the addition of only two bits, is called a *half-adder*.

# Multibit Addition

Generated carries  $\rightarrow$  1 1 1 0

$$\begin{array}{r} X = x_4 x_3 x_2 x_1 x_0 \\ + Y = y_4 y_3 y_2 y_1 y_0 \\ \hline S = s_4 s_3 s_2 s_1 s_0 \end{array} \quad \begin{array}{r} 0 1 1 1 1 \\ + 0 1 0 1 0 \\ \hline 1 1 0 0 1 \end{array} \quad \begin{array}{r} (15)_{10} \\ + (10)_{10} \\ \hline (25)_{10} \end{array}$$



(a) An example of addition

(b) Bit position  $i$

for each bit position  $i$ , the addition operation may include a *carry-in* from bit position  $i - 1$ .

This observation leads to the design of a logic circuit that has three inputs  $x_i$ ,  $y_i$ , and  $c_i$ , and produces the two outputs  $s_i$  and  $c_{i+1}$ .

$c_i$	$x_i$	$y_i$	$c_{i+1}$	$s_i$
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

$c_i$	$x_i y_i$	00	01	11	10
0			1		1
1		1		1	

$$s_i = x_i \oplus y_i \oplus c_i$$

$c_i$	$x_i y_i$	00	01	11	10
0				1	
1			1	1	1

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

(a) Truth table

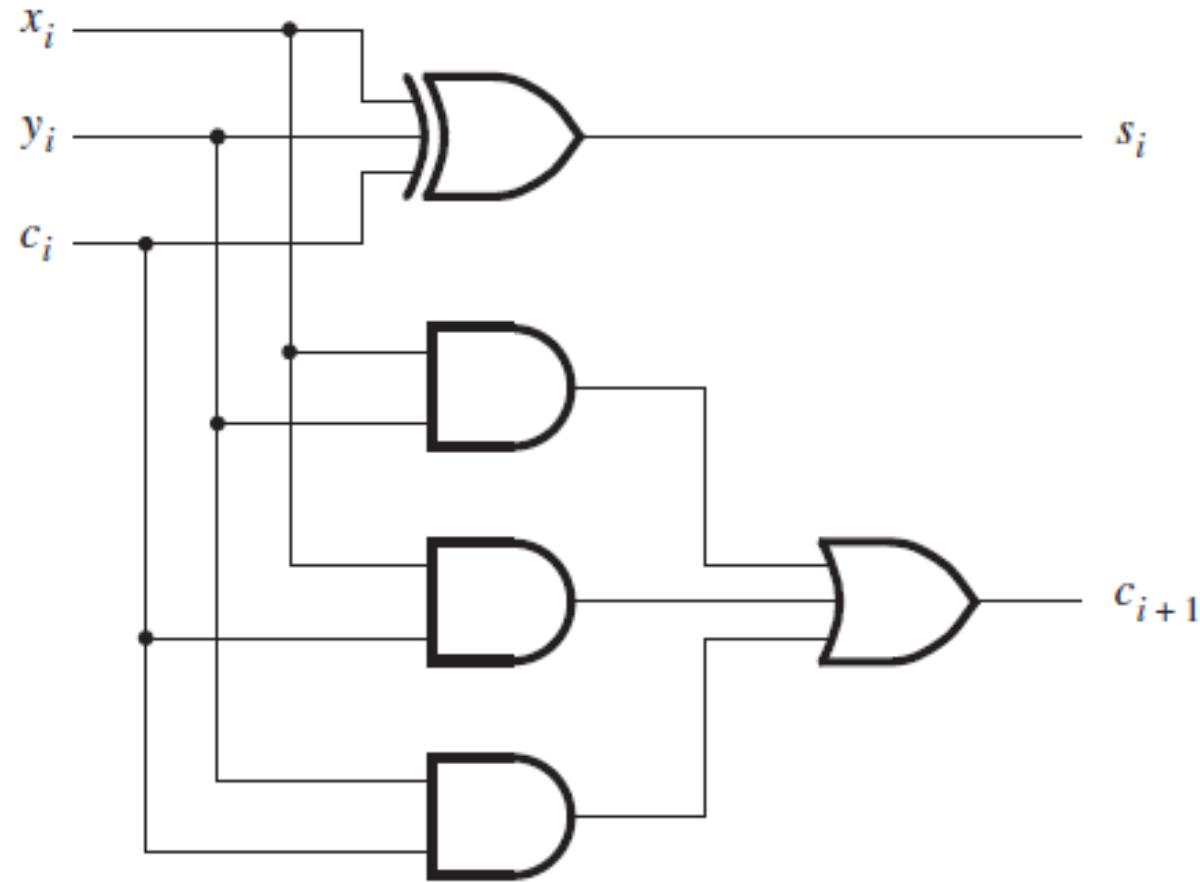
$$s_i = \overline{x_i}y_i\overline{c}_i + x_i\overline{y_i}\overline{c}_i + \overline{x_i}\overline{y_i}c_i + x_iy_i c_i$$

$$s_i = (\overline{x_i}y_i + x_i\overline{y_i})\overline{c}_i + (\overline{x_i}\overline{y_i} + x_iy_i)c_i$$

$$= (x_i \oplus y_i)\overline{c}_i + \overline{(x_i \oplus y_i)}c_i$$

$$= (x_i \oplus y_i) \oplus c_i$$

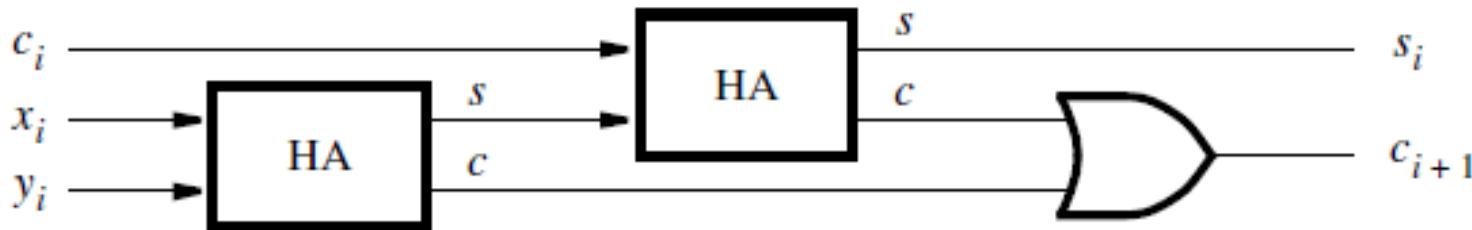
$$s_i = x_i \oplus y_i \oplus c_i$$



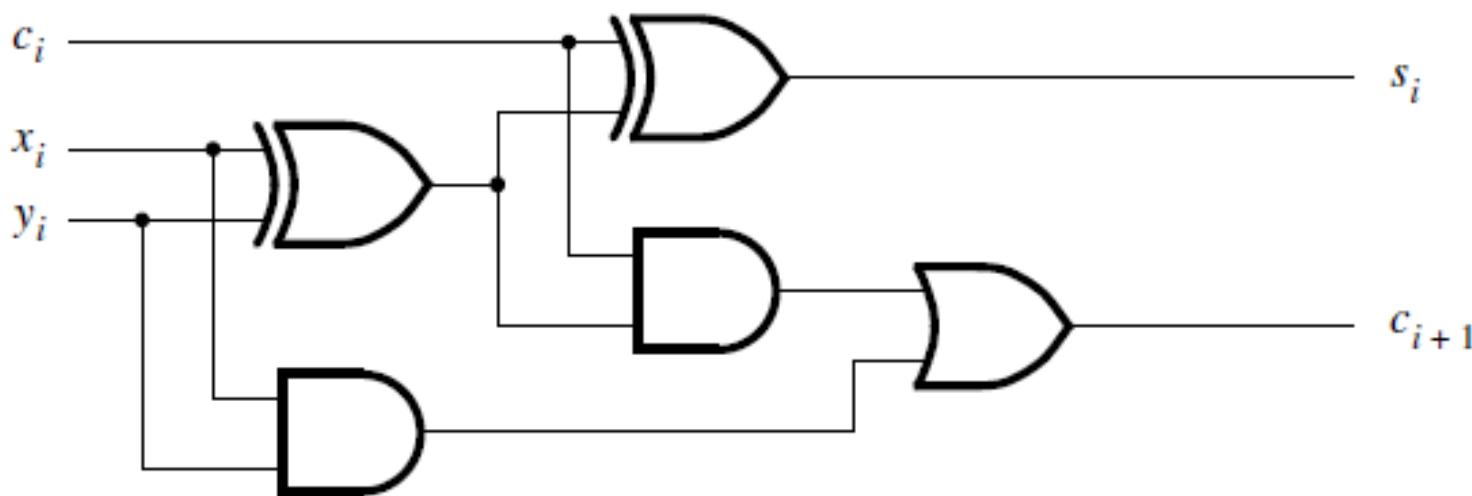
(c) Circuit

This circuit is known as a *full-adder*.

# Decomposed Full-Adder



(a) Block diagram



(b) Detailed diagram

$$S1 = x_i \text{ XOR } y_i$$

$$C1 = x_i y_i$$

$$S2 = S_i = S1 \text{ XOR } c_i = x_i \text{ XOR } y_i \text{ XOR } c_i$$

$$C2 = S1 c_i = (x_i \text{ XOR } y_i) c_i$$

$$= x_i' y_i c_i + x_i y_i' c_i$$

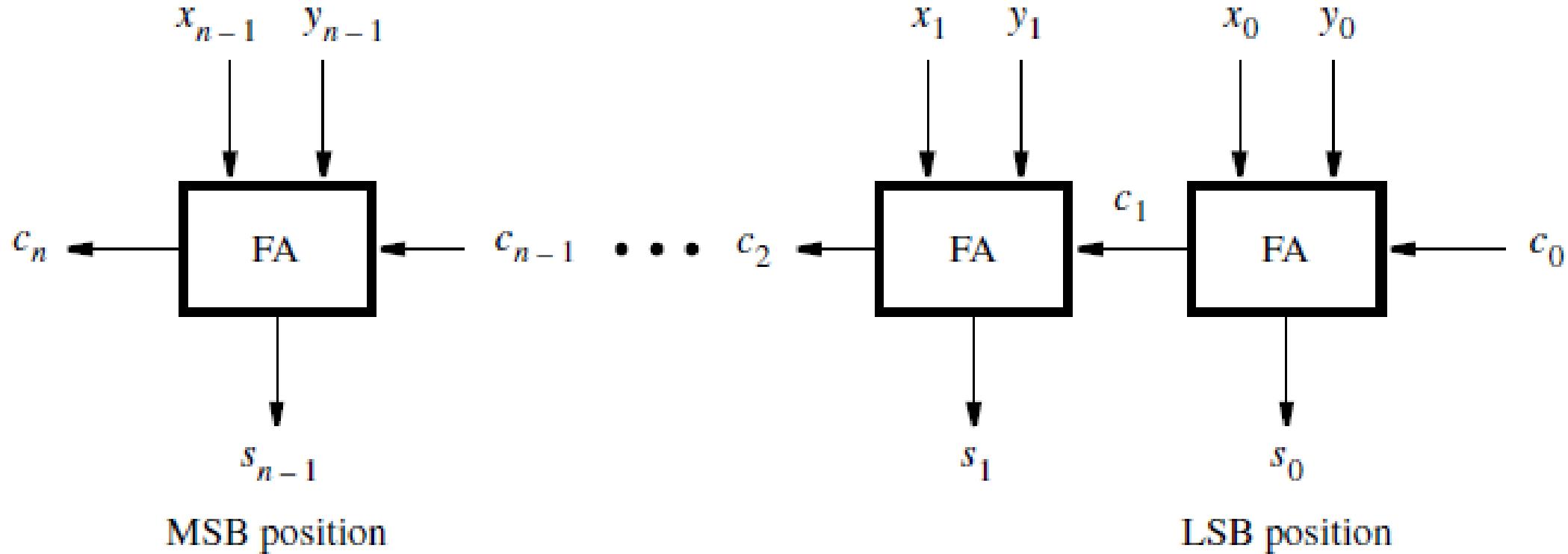
$$C_{i+1} = x_i y_i + x_i' y_i c_i + x_i y_i' c_i$$

$$= x_i y_i (c_i + c_i') + x_i' y_i c_i + x_i y_i' c_i$$

$$= x_i y_i c_i + x_i y_i c_i' + x_i' y_i c_i + x_i y_i' c_i$$

$$= x_i y_i + y_i c_i + x_i c_i$$

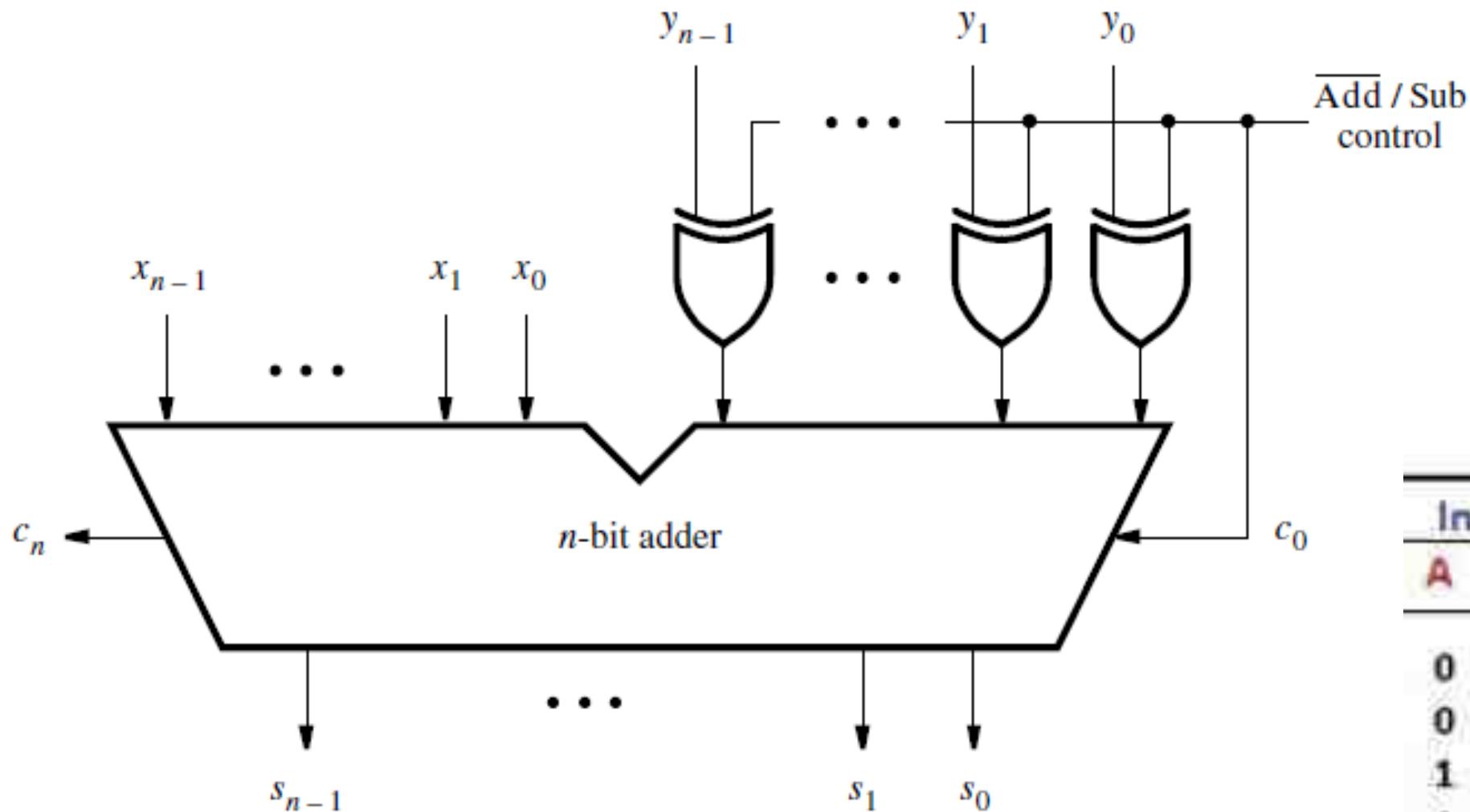
# Ripple-Carry Adder



For each bit position we can use a full-adder circuit, connected as shown in Figure

- When the operands  $X$  and  $Y$  are applied as inputs to the adder, it takes some time before the output sum,  $S$ , is valid. Each full-adder introduces a certain delay before its  $s_i$  and  $c_{i+1}$  outputs are valid. Let this delay be denoted as  $\Delta t$ .
- Thus the carry-out from the first stage,  $c_1$ , arrives at the second stage  $\Delta t$  after the application of the  $x_0$  and  $y_0$  inputs. The carry-out from the second stage,  $c_2$ , arrives at the third stage with a  $2\Delta t$  delay, and so on.
- The signal  $c_{n-1}$  is valid after a delay of  $(n - 1) \Delta t$ , which means that the complete sum is available after a delay of  $n\Delta t$ . Because of the way the carry signals “ripple” through the full-adder stages, the circuit is called a ripple-carry adder.

# Adder and Subtractor Unit



Inputs		Output
A	B	$A \text{ XOR } B$
0	0	0
0	1	1
1	0	1
1	1	0

When Add/Sub = 0, it performs  $X + Y$ .

When Add/Sub = 1, it performs

$$X + Y' + 1$$

=  **$X + 2\text{'s complement of } Y$**

$$= X - Y$$

# Arithmetic Overflow

The result of addition or subtraction is supposed to fit within the significant bits used to represent the numbers. If  $n$  bits are used to represent signed numbers, then the result must be in the range  $-2^{n-1}$  to  $2^{n-1} - 1$ . If the result does not fit in this range, then we say that *arithmetic overflow* has occurred.

$$\begin{array}{r} (+7) \\ + (+2) \\ \hline (+9) \end{array} \quad \begin{array}{r} 0111 \\ + 0010 \\ \hline 1001 \end{array}$$

$$\begin{array}{r} (-7) \\ + (+2) \\ \hline (-5) \end{array} \quad \begin{array}{r} 1001 \\ + 0010 \\ \hline 1011 \end{array}$$

$$\begin{array}{l} c_4 = 0 \\ c_3 = 1 \end{array}$$

$$\begin{array}{l} c_4 = 0 \\ c_3 = 0 \end{array}$$

$$\begin{array}{r} (+7) \\ + (-2) \\ \hline (+5) \end{array} \quad \begin{array}{r} 0111 \\ + 1110 \\ \hline 10101 \end{array}$$

$$\begin{array}{r} (-7) \\ + (-2) \\ \hline (-9) \end{array} \quad \begin{array}{r} 1001 \\ + 1110 \\ \hline 10111 \end{array}$$

$$\begin{array}{l} c_4 = 1 \\ c_3 = 1 \end{array}$$

$$\begin{array}{l} c_4 = 1 \\ c_3 = 0 \end{array}$$

We are using four-bit numbers. If both numbers have the same sign, the magnitude of the result is 9, which cannot be represented with 4 bits in 2's complement representation. Therefore, overflow occurs. When the numbers have opposite signs, there is no overflow. The key to determine whether overflow occurs is the carry-out from the sign-bit position, called  $c_4$  in the figure and from the previous bit position, called  $c_3$ . The figure indicates that overflow occurs when these carry-outs have different values, and a correct sum is produced when they have the same value. Indeed, this is true in general for both addition and subtraction of 2's-complement numbers.

$$\begin{aligned}\text{Overflow} &= c_3 \bar{c}_4 + \bar{c}_3 c_4 \\ &= c_3 \oplus c_4\end{aligned}$$

For  $n$ -bit numbers we have

$$\text{Overflow} = c_{n-1} \oplus c_n$$

An alternative and more intuitive way of detecting the arithmetic overflow is to observe that overflow occurs if both summands have the same sign but the resulting sum has a different sign. Let  $X = x_3x_2x_1x_0$  and  $Y = y_3y_2y_1y_0$  represent four-bit 2's-complement numbers, and let  $S = s_3s_2s_1s_0$  be the sum  $S = X + Y$ . Then

$$\text{Overflow} = x_3 y_3 \bar{s}_3 + \bar{x}_3 \bar{y}_3 s_3$$

The carry-out and overflow signals indicate whether the result of a given addition is too large to fit into the number of bits available to represent the sum. The carry-out is meaningful only when unsigned numbers are involved, while the overflow is meaningful only in the case of signed numbers.

# Fast Adders

The addition and subtraction of numbers are fundamental operations that are performed frequently in the course of a computation. The speed with which these operations are performed has a strong impact on the overall performance of a computer.

Let us take a closer look at the speed of the adder/subtractor unit. We are interested in the largest delay from the time the operands  $X$  and  $Y$  are presented as inputs, until the time all bits of the sum  $S$  and the final carry-out,  $c_n$ , are valid.

The delay for the carry-out signal in the full adder circuit,  $\Delta t$ , is equal to two gate delays.

The final result of the addition will be valid after a delay of  $n\Delta t$ , which is equal to  $2n$  gate delays.

In addition to the delay in the ripple-carry path, there is also a delay in the XOR gates that feed either the true or complemented value of  $Y$  to the adder inputs. If this delay is equal to one gate delay, then the total delay of the circuit is  $2n + 1$  gate delays. For a large  $n$ , say  $n = 32$  or  $n = 64$ , the delay would lead to unacceptably poor performance.

The longest delay is along the path from the  $y_i$  input, through the XOR gate and through the carry circuit of each adder stage. The longest delay is often referred to as the *critical-path delay*, and the path that causes this delay is called the *critical path*.

## Carry-Lookahead Adder

To reduce the delay caused by the effect of carry propagation through the ripple-carry adder, we can attempt to evaluate quickly for each stage whether the carry-in from the previous stage will have a value 0 or 1. If a correct evaluation can be made in a relatively short time, then the performance of the complete adder will be improved.

$$C_{i+1} = x_i y_i + \alpha c_i + \beta i c_i$$

$$c_{i+1} = x_i y_i + (x_i + y_i) c_i$$

$$c_{i+1} = g_i + p_i c_i$$

$$g_i = x_i y_i$$

$$p_i = x_i + y_i$$

$g_i$  is called the *generate* function.

$p_i$  is called the *propagate* function.

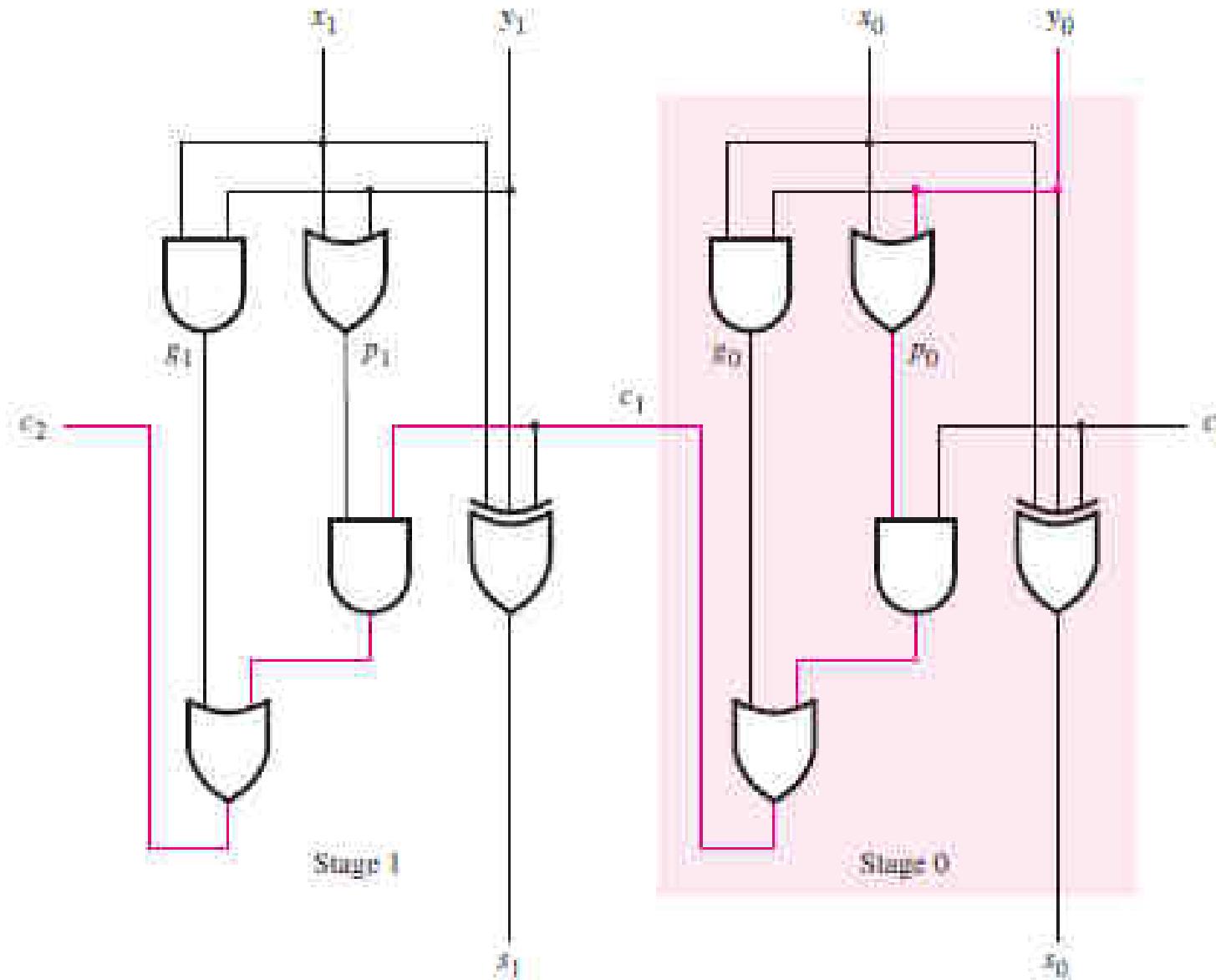
Expanding in terms of stage  $i - 1$  gives

$$\begin{aligned}c_{i+1} &= g_i + p_i(g_{i-1} + p_{i-1}c_{i-1}) \\&= g_i + p_i g_{i-1} + p_i p_{i-1} c_{i-1}\end{aligned}$$

The same expansion for other stages, ending with stage 0, gives

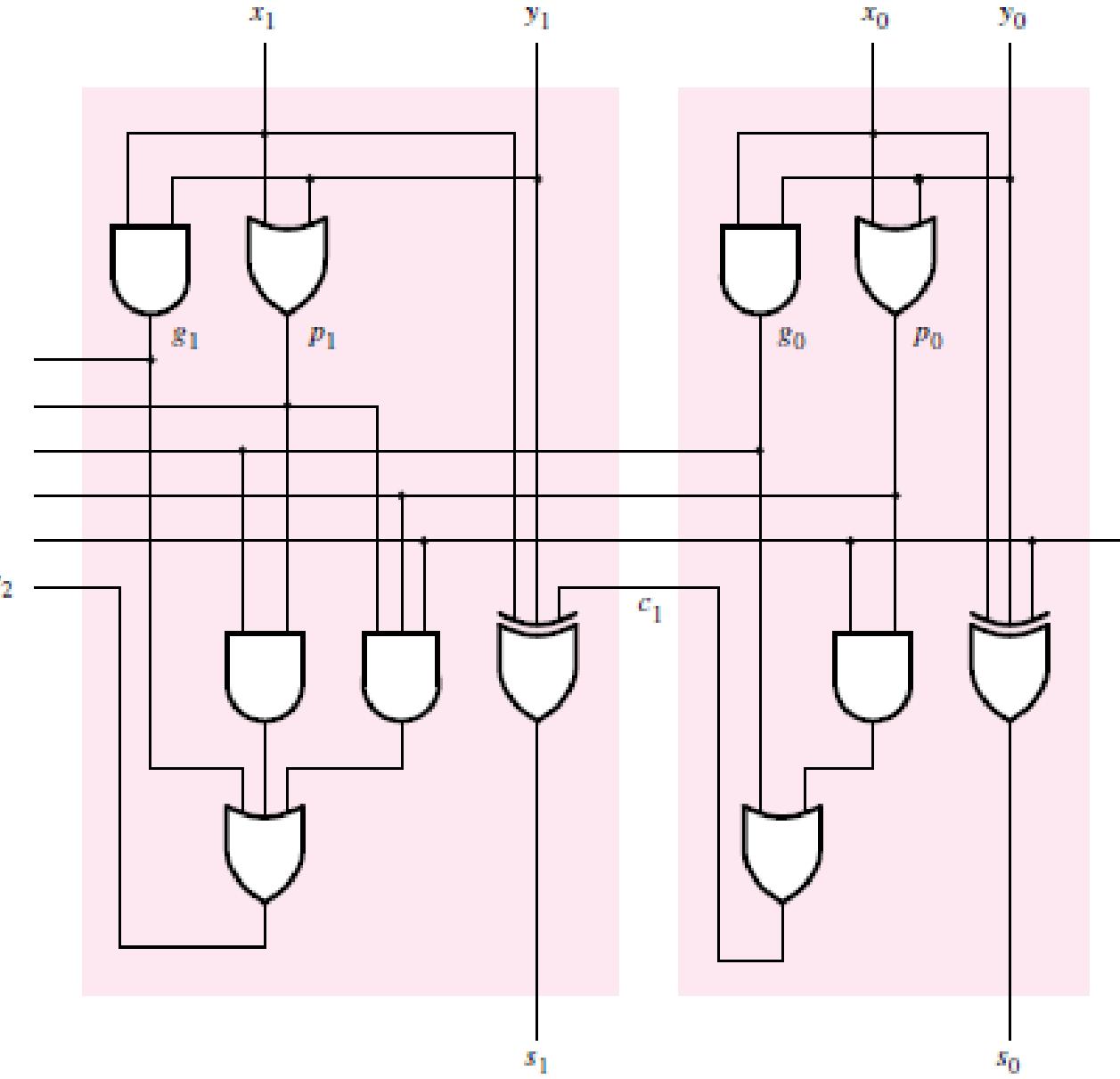
$$c_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \cdots + p_i p_{i-1} \cdots p_2 p_1 g_0 + p_i p_{i-1} \cdots p_1 p_0 c_0$$

This expression represents a two-level AND-OR circuit in which  $c_{i+1}$  is evaluated very quickly. An adder based on this expression is called a *carry-lookahead adder*.



A ripple-carry adder

The critical path is from inputs  $x_0$  and  $y_0$  to the output  $c_2$ . It passes through **five gates**. The path in other stages of an  $n$ -bit adder is the same as in stage 1. Therefore, the total number of gate delays along the critical path is  $2n + 1$ .

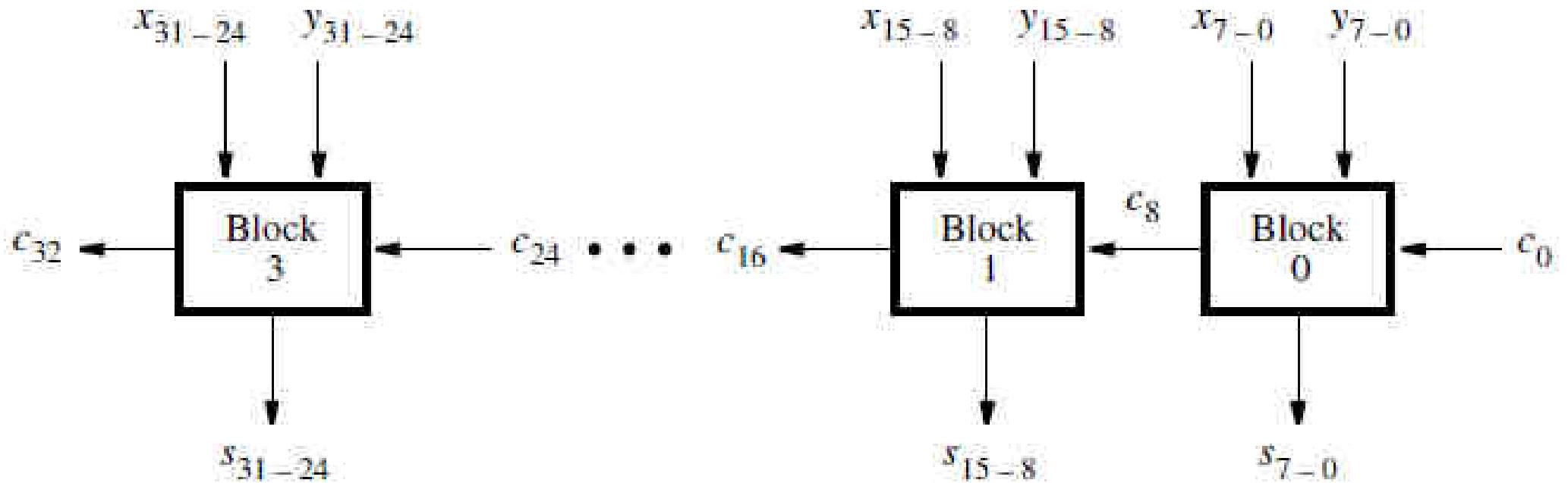


The first two stages of a carry-lookahead adder

All carry signals are produced after three gate delays: one gate delay is needed to produce the generate and propagate signals  $g_0$ ,  $g_1$ ,  $p_0$ , and  $p_1$ , and two more gate delays are needed to produce  $c_1$  and  $c_2$  concurrently. Extending the circuit to  $n$  bits, the final carry-out signal  $c_n$  would also be produced after only three gate delays.

The total delay in the  $n$ -bit carry-lookahead adder is four gate delays. The values of all  $gi$  and  $pi$  signals are determined after one gate delay. It takes two more gate delays to evaluate all carry signals. Finally, it takes one more gate delay (XOR) to generate all sum bits. The key to the good performance of the adder is quick evaluation of carry signals.

- The complexity of an  $n$ -bit carry-lookahead adder increases rapidly as  $n$  becomes larger.
- To reduce the complexity, we can use a hierarchical approach in designing large adders.
- Suppose that we want to design a 32-bit adder. We can divide this adder into 4 eight-bit blocks, such that block 0 adds bits 7 . . . 0, block 1 adds bits 15 . . . 8, block 2 adds bits 23 . . . 16, and block 3 adds bits 31 . . . 24. Then we can implement each block as an eight-bit carry-lookahead adder. The carry-out signals from the four blocks are  $c_8$ ,  $c_{16}$ ,  $c_{24}$ , and  $c_{32}$ . Now we have two possibilities.
  - We can connect the four blocks as four stages in a ripple-carry adder. Thus while carry-lookahead is used within each block, the carries ripple between the blocks.



A hierarchical carry-lookahead adder with ripple-carry between blocks.

gi, pi : 1 gate delay

c1-c8 : 2 gate delay

c9-c16 : 2 gate delay

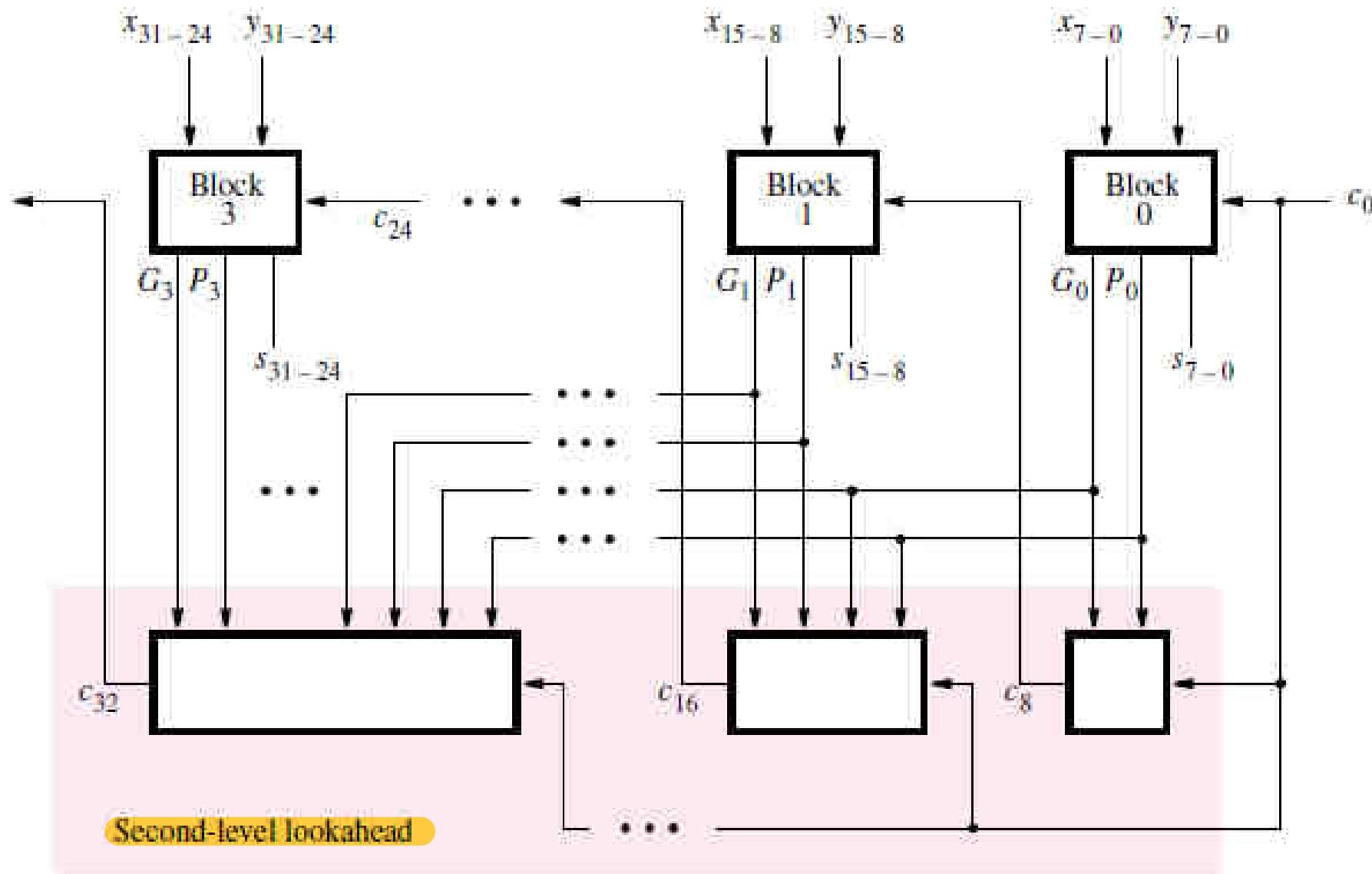
c17-c24 : 2 gate delay

c25-c32 : 2 gate delay

One more gate delay to produce the sum bits form  
the last block

Total : 10 gate dealy.

Instead of using a ripple-carry approach between blocks, a faster circuit can be designed in which a second-level carry-lookahead is performed to produce quickly the carry signals between blocks. The structure of this “hierarchical carry-lookahead adder” is shown in the figure below



A hierarchical carry-lookahead adder.

- Each block in the top row includes an eight-bit carry-lookahead adder, based on generate and propagate signals for each stage in the block.
- Instead of producing a carry-out signal from the most-significant bit of the block, each block produces generate and propagate signals for the entire block. Let  $G_j$  and  $P_j$  denote these signals for each block  $j$ .
- Now  $G_j$  and  $P_j$  can be used as inputs to a second-level carrylookahead circuit at the bottom of the Figure, which evaluates all carries between blocks.
- We can derive the block generate and propagate signals for block 0 by examining the expression for  $c_8$

$$c_8 = g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4 + p_7p_6p_5p_4g_3 + p_7p_6p_5p_4p_3g_2 \\ + p_7p_6p_5p_4p_3p_2g_1 + p_7p_6p_5p_4p_3p_2p_1g_0 + p_7p_6p_5p_4p_3p_2p_1p_0c_0$$

$$P_0 = p_7p_6p_5p_4p_3p_2p_1p_0$$

$$G_0 = g_7 + p_7g_6 + p_7p_6g_5 + \dots + p_7p_6p_5p_4p_3p_2p_1g_0$$

$$c_8 = G_0 + P_0c_0$$

For block 1 the expressions for  $G_1$  and  $P_1$  have the same form as for  $G_0$  and  $P_0$  except that each subscript  $i$  is replaced by  $i + 8$ . The expressions for  $G_2$ ,  $P_2$ ,  $G_3$ , and  $P_3$  are derived in the same way. The expression for the carry-out of block 1,  $c_{16}$ , is

$$\begin{aligned}
 c_{16} &= G_1 + P_1 c_8 \\
 &= G_1 + P_1 G_0 + P_1 P_0 c_0
 \end{aligned}$$

Similarly, the expressions for  $c_{24}$  and  $c_{32}$  are

$$\begin{aligned}
 c_{24} &= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0 \\
 c_{32} &= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0
 \end{aligned}$$

Using this scheme, it takes two more gate delays to produce the carry signals  $c_8$ ,  $c_{16}$ ,  $c_{24}$ , and  $c_{32}$  than the time needed to generate the  $G_j$  and  $P_j$  functions. Therefore, since  $G_j$  and  $P_j$  require three gate delays,  $c_8$ ,  $c_{16}$ ,  $c_{24}$ , and  $c_{32}$  are available after five gate delays. The time needed to add two 32-bit numbers involves these five gate delays plus two more to produce the internal carries in blocks 1, 2, and 3, plus one more gate delay (XOR) to generate each sum bit. This gives a total of eight gate delays.

It takes  $2n + 1$  gate delays to add two numbers using a ripple-carry adder. For 32-bit numbers this implies 65 gate delays. It is clear that the carry-lookahead adder offers a large performance improvement.

## Technology Considerations

consider the expressions for the first eight carries in CLA:

$$c_1 = g_0 + p_0 c_0$$

$$c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$$

...

$$\begin{aligned} c_8 = & g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4 + p_7 p_6 p_5 p_4 g_3 \\ & + p_7 p_6 p_5 p_4 p_3 g_2 + p_7 p_6 p_5 p_4 p_3 p_2 g_1 + \\ & p_7 p_6 p_5 p_4 p_3 p_2 p_1 g_0 + p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0 c_0 \end{aligned}$$

Suppose that the maximum fan-in of the gates is four inputs. Then it is impossible to implement all of these expressions with a two-level AND-OR circuit. The biggest problem is  $c_8$ , where one of the AND gates requires nine inputs; moreover, the OR gate also requires nine inputs. To meet the fan-in constraint, we can rewrite the expression for  $c_8$  as

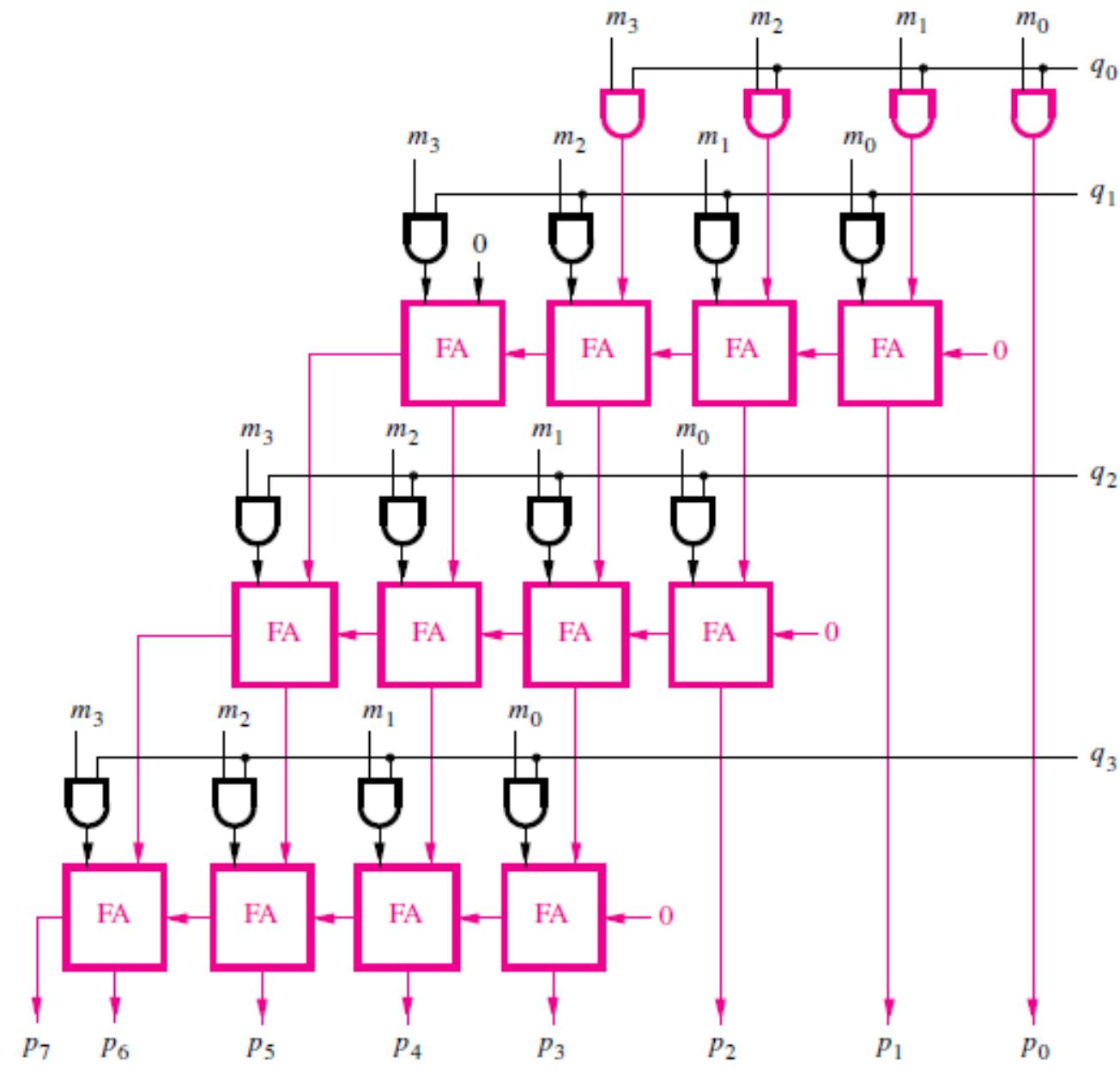
$$\begin{aligned}c_8 = & (g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4) \\& + [(p_7p_6p_5p_4)(g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0)] \\& + (p_7p_6p_5p_4)(p_3p_2p_1p_0)c_0\end{aligned}$$

To implement this expression we need ten AND gates and three OR gates. The propagation delay in generating  $c_8$  consists of one gate delay to develop all  $g_i$  and  $p_i$ , two gate delays to produce the sum-of-products terms in parentheses, one gate delay to form the product term in square brackets, and one delay for the final ORing of terms. Hence  $c_8$  is valid after five gate delays, rather than the three gate delays that would be needed without the fan-in constraint.

# Multiplication

Two binary numbers can be multiplied using the same method as we use for decimal numbers.

	$m_3$	$m_2$	$m_1$	$m_0$				
	×	$q_3$	$q_2$	$q_1$	$q_0$			
Partial product 0	$m_3 q_0$	$m_2 q_0$	$m_1 q_0$	$m_0 q_0$				
	+	$m_3 q_1$	$m_2 q_1$	$m_1 q_1$	$m_0 q_1$			
Partial product 1	$PP1_5$	$PP1_4$	$PP1_3$	$PP1_2$	$PP1_1$			
	+	$m_3 q_2$	$m_2 q_2$	$m_1 q_2$	$m_0 q_2$			
Partial product 2	$PP2_6$	$PP2_5$	$PP2_4$	$PP2_3$	$PP2_2$			
	+	$m_3 q_3$	$m_2 q_3$	$m_1 q_3$	$m_0 q_3$			
Product P	$P_7$	$P_6$	$P_5$	$P_4$	$P_3$	$P_2$	$P_1$	$P_0$



# Binary-Coded-Decimal Representation

**Table 3.3** Binary-coded decimal digits.

Decimal digit	BCD code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

# BCD Addition

$$\begin{array}{r} X \\ + Y \\ \hline Z \end{array} \quad \begin{array}{r} 0111 \\ + 0101 \\ \hline 1100 \end{array} \quad \begin{array}{r} 7 \\ + 5 \\ \hline 12 \end{array}$$
$$\begin{array}{r} + 0110 \\ \hline \end{array}$$

carry →  $\underbrace{10010}_{S=2}$

$$\begin{array}{r} X \\ + Y \\ \hline Z \end{array} \quad \begin{array}{r} 1000 \\ + 1001 \\ \hline 10001 \end{array} \quad \begin{array}{r} 8 \\ + 9 \\ \hline 17 \end{array}$$
$$\begin{array}{r} + 0110 \\ \hline \end{array}$$

carry →  $\underbrace{10111}_{S=7}$

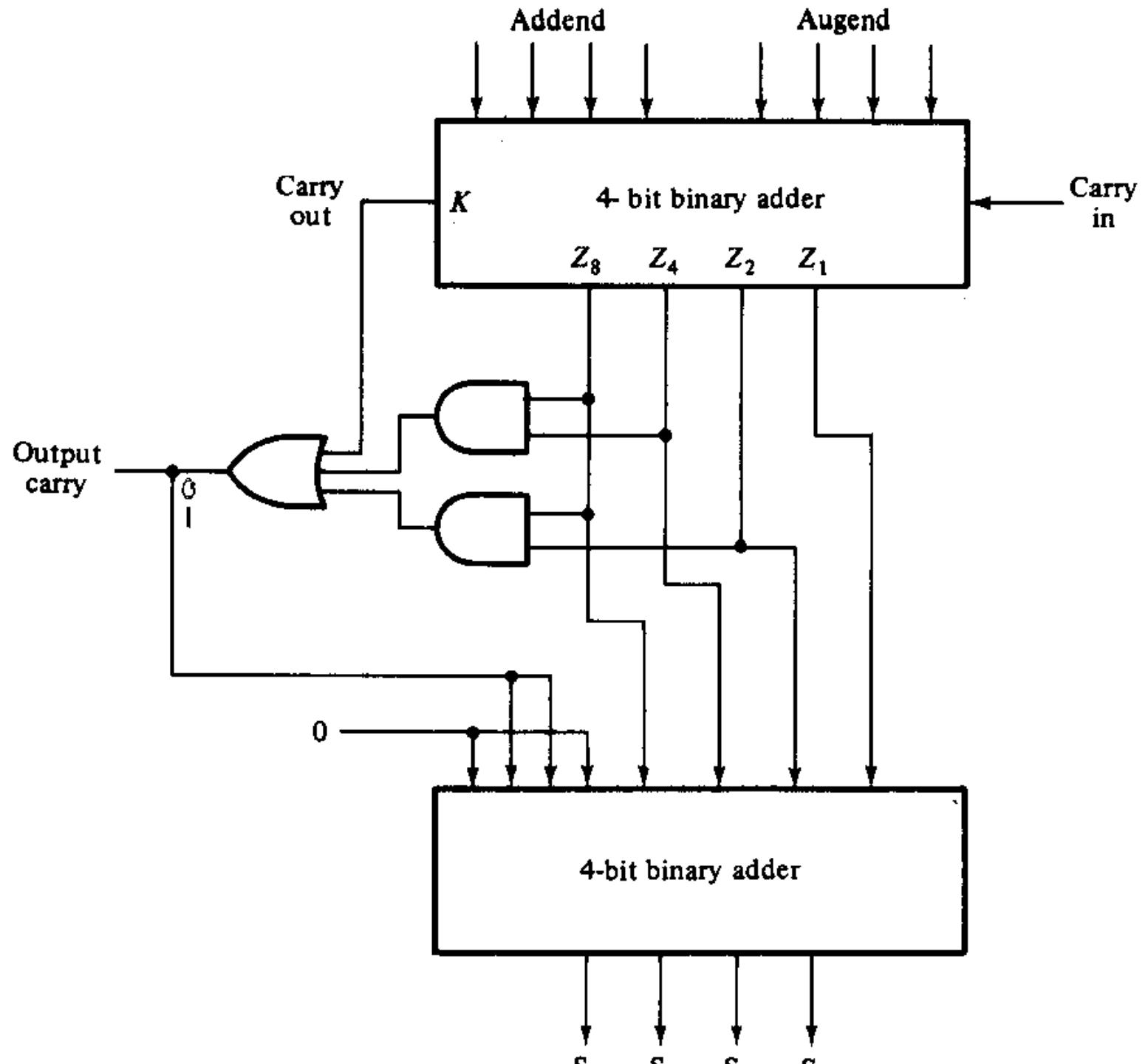
Binary Sum					BCD Sum					Decimal
$K$	$Z_8$	$Z_4$	$Z_2$	$Z_1$	$C$	$S_8$	$S_4$	$S_2$	$S_1$	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
<hr/>										
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

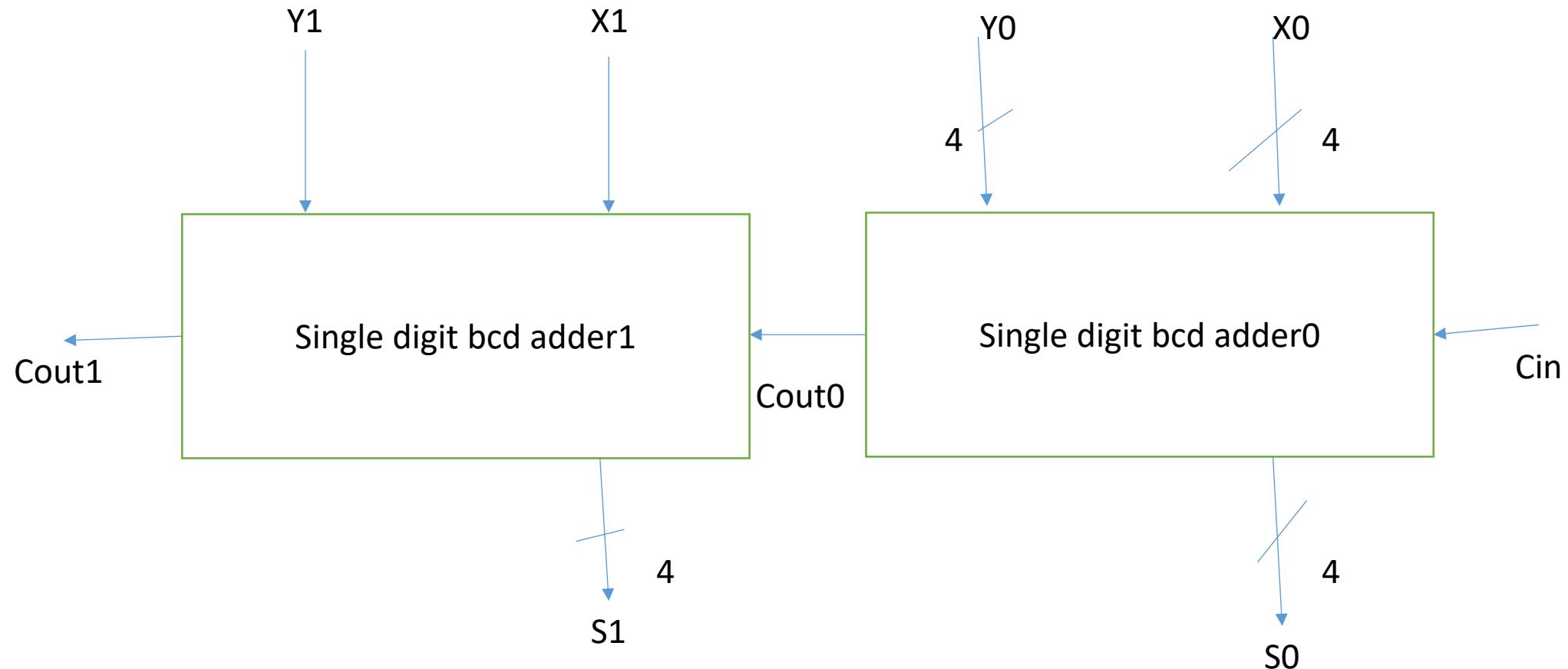
$z_8, z_4$	$z_2, z_1$	00	01	11	10
00	0	0	0	0	0
01	0	0	0	0	0
11	1	1	1	1	1
10	0	0	1	1	1

Correction should be done when the carry  $k = 1$  or when the expression  $z_8z_4 + z_8z_2$  evaluates to 1. Therefore the expression for the correction circuit is

$$C = K + Z_8Z_4 + Z_8Z_2$$

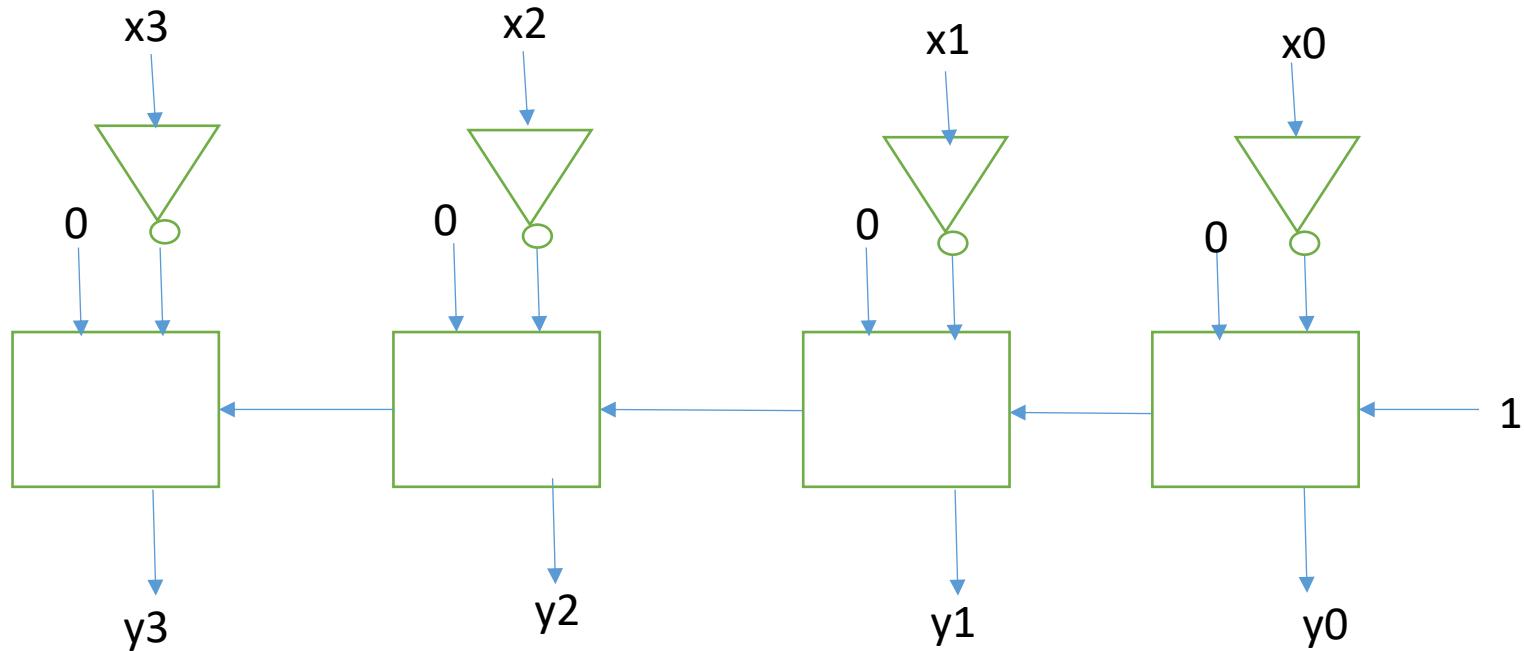
$$C = K + Z_8Z_4 + Z_8Z_2$$





## Exercise 1

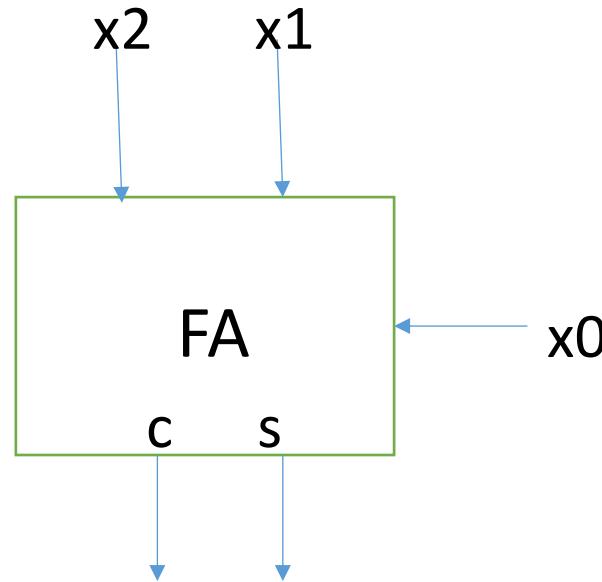
Design a circuit using full adders to generate 2's complement of a 4 bit number X



$y_3y_2y_1y_0$  is the 2's complement of  $X$

## Exercise 2

Design a circuit using full adders to find the no. of 1's in a three bit unsigned number



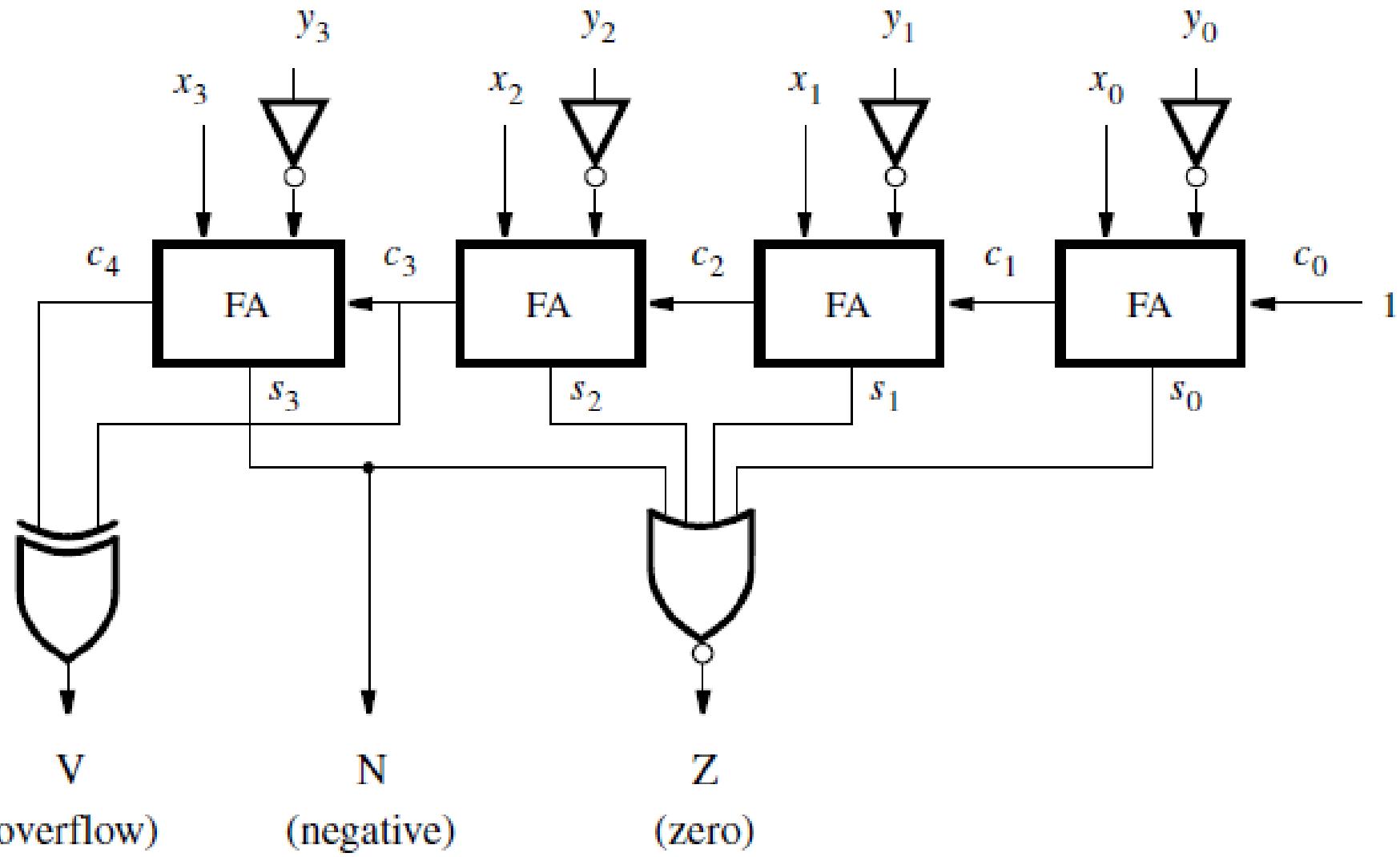
CS is the result in binary.

## Example

**Problem:** In computer computations it is often necessary to compare numbers. Two four-bit signed numbers,  $X = x_3x_2x_1x_0$  and  $Y = y_3y_2y_1y_0$ , can be compared by using the subtractor circuit. The three outputs denote the following:

- $Z = 1$  if the result is 0; otherwise  $Z = 0$
- $N = 1$  if the result is negative; otherwise  $N = 0$
- $V = 1$  if arithmetic overflow occurs; otherwise  $V = 0$

Show how  $Z$ ,  $N$ , and  $V$  can be used to determine the cases  $X = Y$ ,  $X < Y$ ,  $X \leq Y$ ,  $X > Y$ , and  $X \geq Y$ .



Consider first the case  $X < Y$ , where the following possibilities may arise:

- If  $X$  and  $Y$  have the same sign there will be no overflow, hence  $V = 0$ . Then for both positive and negative  $X$  and  $Y$  the difference will be negative ( $N = 1$ ).
- If  $X$  is negative and  $Y$  is positive, the difference will be negative ( $N = 1$ ) if there is no overflow ( $V = 0$ ); but the result will be positive ( $N = 0$ ) if there is overflow ( $V = 1$ ).

Therefore, if  $X < Y$  then  $N \oplus V = 1$ .

The case  $X = Y$  is detected by  $Z = 1$ . Then,  $X \leq Y$  is detected by  $Z + (N \oplus V) = 1$ .

$X > Y$  if  $(Z + (N \oplus V))' = 1$

$X \geq Y$  if  $(N \oplus V)' = 1$ .

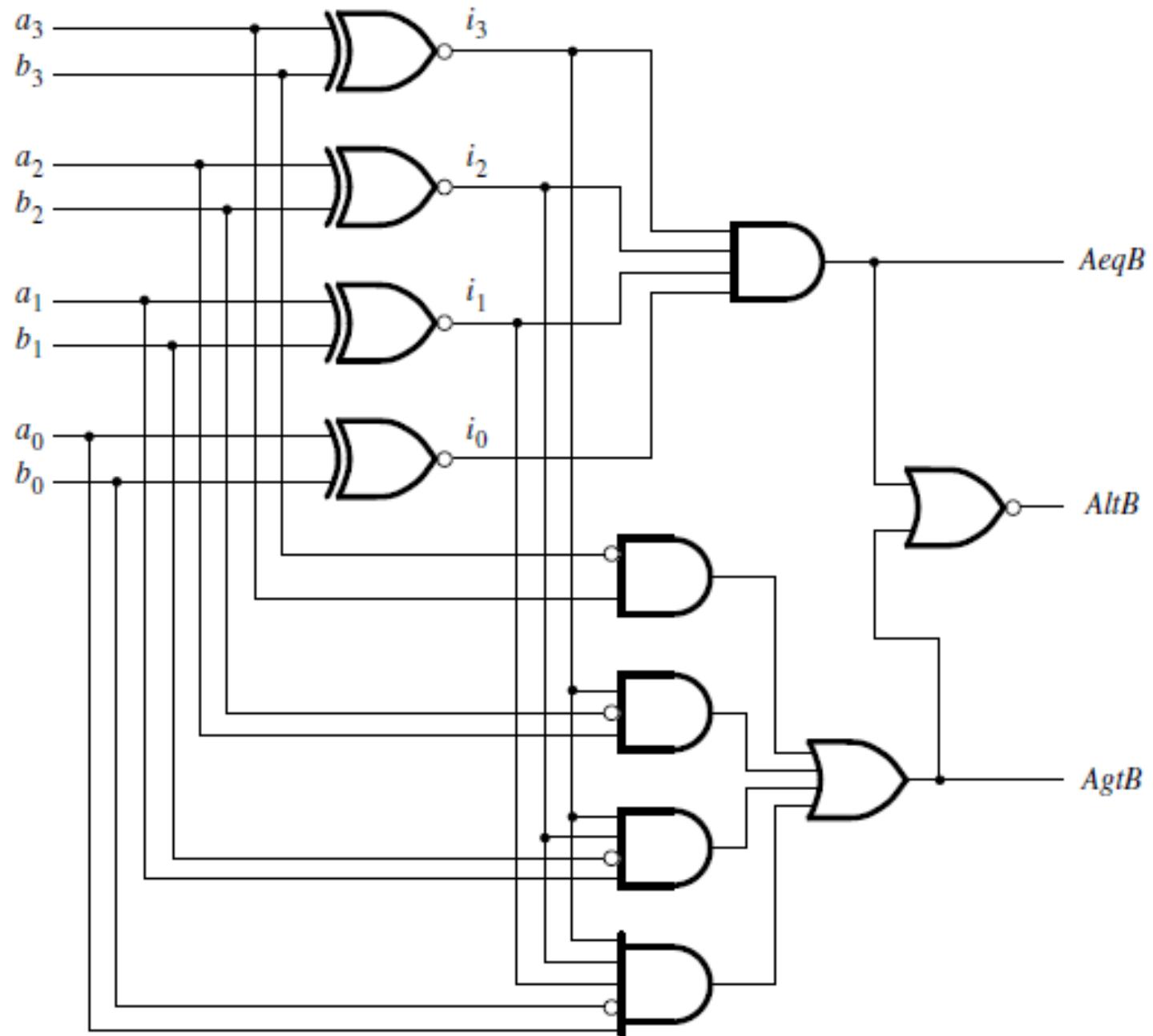
## Arithmetic Comparison Circuits

Let  $A = a_3a_2a_1a_0$  and  $B = b_3b_2b_1b_0$ . Define a set of intermediate signals called  $i_3, i_2, i_1$ , and  $i_0$ . Each signal,  $i_k$ , is 1 if the bits of  $A$  and  $B$  with the same index are equal.

That is,  $i_k = (a_k \oplus b_k)'$ . The comparator's  $A \text{eq} B$  output is then given by  $A \text{eq} B = i_3i_2i_1i_0$

$$A \text{eq} B = \overline{a_3\bar{b}_3} + \overline{i_3a_2\bar{b}_2} + \overline{i_3i_2a_1\bar{b}_1} + \overline{i_3i_2i_1a_0\bar{b}_0}$$

$$A \neq B = \overline{A \text{eq} B} + A \text{gt} B$$



# **Introduction to Verilog HDL**

Verilog was originally intended for simulation and verification of digital circuits. Subsequently, using CAD (Computer Aided Design) tools, the Verilog Codes are synthesized into hardware implementation of the described circuit. We are using the Verilog in the lab for simulation and verification of digital circuits

There are two ways of describing the circuit in Verilog.

1. Using Verilog constructs, describe the structure of the circuit in terms of circuit elements, such as logic gates. A larger circuit is defined by writing code that connects such elements together. This approach is referred to as the *structural* representation of logic circuits.
2. Using logic expressions and Verilog programming constructs that define the desired behavior of the circuit, but not its actual structure in terms of gates, describe a circuit more abstractly. This is called the *behavioral* representation.

# Structural Specification of Logic Circuits

Verilog includes a set of *gate-level primitives* that correspond to commonly-used logic gates.

A gate is represented by indicating its functional name, output, and inputs.

For example, a two-input AND gate, with output  $y$  and inputs  $x_1$  and  $x_2$ , is denoted as

**and** ( $y$ ,  $x_1$ ,  $x_2$ );

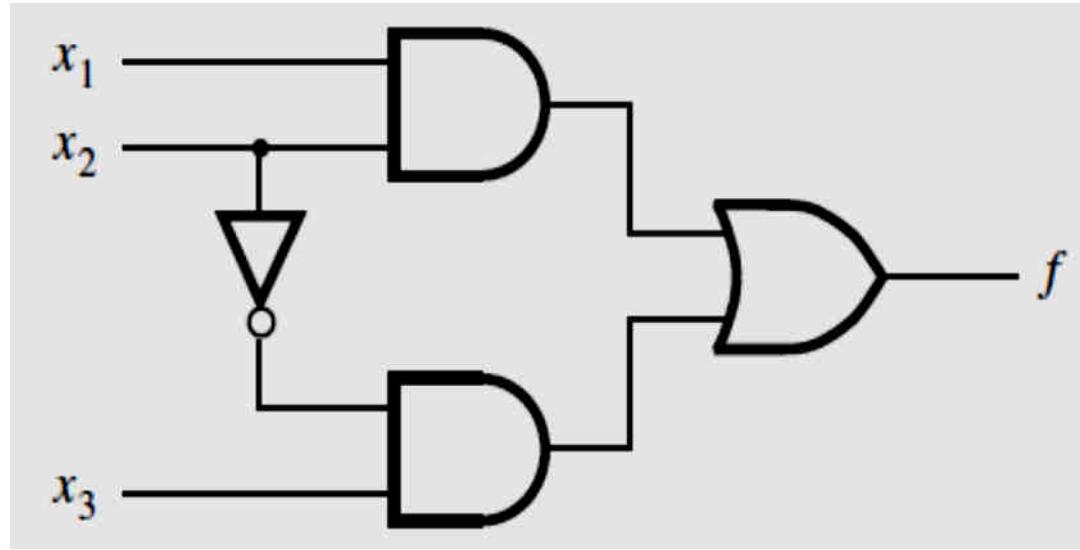
A four-input OR gate is specified as

**or** ( $y$ ,  $x_1$ ,  $x_2$ ,  $x_3$ ,  $x_4$ );

# The available Verilog gate-level primitives are

Name	Description	Usage
and	$f = (a \cdot b \cdot \dots)$	<b>and</b> ( $f, a, b, \dots$ )
nand	$f = \overline{(a \cdot b \cdot \dots)}$	<b>nand</b> ( $f, a, b, \dots$ )
or	$f = (a + b + \dots)$	<b>or</b> ( $f, a, b, \dots$ )
nor	$f = \overline{(a + b + \dots)}$	<b>nor</b> ( $f, a, b, \dots$ )
xor	$f = (a \oplus b \oplus \dots)$	<b>xor</b> ( $f, a, b, \dots$ )
xnor	$f = (a \odot b \odot \dots)$	<b>xnor</b> ( $f, a, b, \dots$ )
not	$f = \overline{a}$	<b>not</b> ( $f, a$ )

A logic circuit is specified in the form of a *module* that contains the statements that define the circuit. A module has inputs and outputs, which are referred to as its *ports*. The word port is a commonly-used term that refers to an input or output connection to an electronic circuit.



```
module example1(x1,x2,x3,f);
  input x1,x2,x3;
  output f;
  and (g,x1,x2);
  not (k,x2);
  and (h,k,x3);
  or (f,g,h);
endmodule
```

The first statement gives the module a name, *example1*, and indicates that there are four port signals. The next two statements declare that  $x_1$ ,  $x_2$ , and  $x_3$  are to be treated as **input** signals, while  $f$  is the output. The actual structure of the circuit is specified in the four statements that follow. The NOT gate gives  $k = x_2'$ . The AND gates produce  $g = x_1x_2$  and  $h = kx_3$ . The outputs of AND gates are combined in the OR gate to form  $f = g + h = x_1x_2 + kx_3 = x_1x_2 + x_2'x_3$ . The module ends with the **endmodule** statement.

## Example2:

It defines a circuit that has four input signals,  $x_1$ ,  $x_2$ ,  $x_3$ , and  $x_4$ , and three output signals,  $f$ ,  $g$ , and  $h$ . It implements the logic functions

$$g = x_1x_3 + x_2x_4$$

$$h = (x_1 + x_3')(x_2' + x_4)$$

$$f = g + h$$

```
module example2 (x1, x2, x3, x4, f, g, h);  
input x1, x2, x3, x4;  
output f, g, h;  
and (z1, x1, x3);  
and (z2, x2, x4);  
or (g, z1, z2);  
or (z3, x1, ~x3);  
or (z4, ~x2, x4)  
and (h, z3, z4);  
or (f, g, h);  
endmodule
```

Instead of using explicit NOT gates to define  $x_2$  and  $x_3$ , we have used the Verilog operator “~” (tilde character on the keyboard) to denote complementation. Thus,  $x_2$  is indicated as  $\sim x_2$  in the code.

## **Verilog Syntax**

The names of modules and signals in Verilog code follow two simple rules:

- The name must start with a letter, and it can contain any letter or number plus the “\_” underscore and “\$” characters.
- Verilog is case sensitive

Indentation and blank lines can be used to make separate parts of the code easily recognizable.

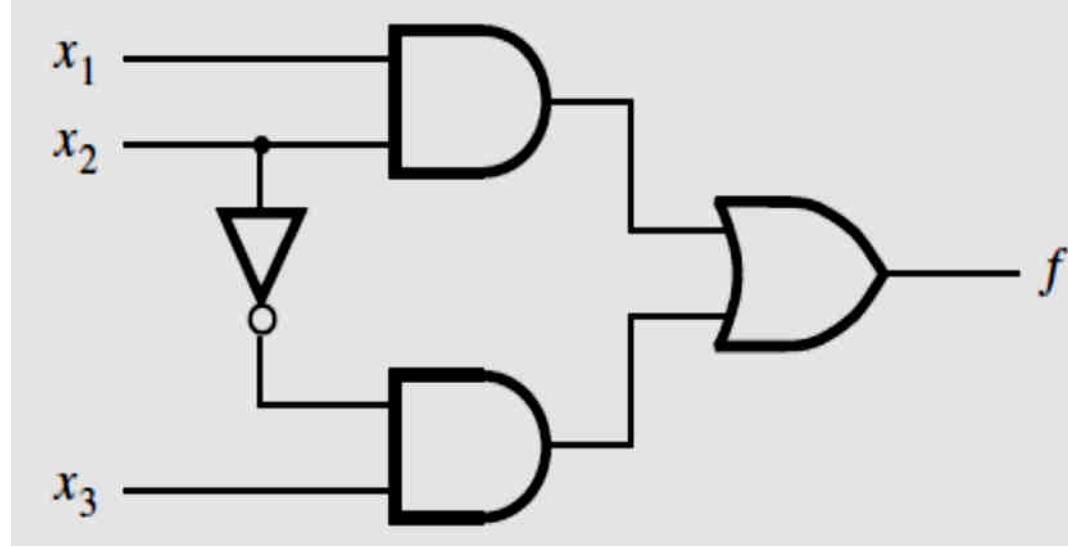
Comments may be included in the code to improve its readability.

A comment begins with the double slash “//” and continues to the end of the line.

# Behavioral Specification of Logic Circuits

Using gate-level primitives can be tedious when large circuits have to be designed. An alternative is to use more abstract expressions and programming constructs to describe the behavior of a logic circuit. One possibility is to define the circuit using logic expressions.

Behavioral specification of a logic circuit defines only its behavior. CAD synthesis tools use this specification to construct the actual circuit. The detailed structure of the synthesized circuit will depend on the technology used.



```
module example1(x1,x2,x3,f);
    input x1,x2,x3;
    output f;
    assign f=(x1 & x2)| (~x2 & x3);
endmodule
```

The AND and OR operations are indicated by the “&” and “|” Verilog operators, respectively. The *assign* keyword provides a *continuous assignment* for the signal *f*.

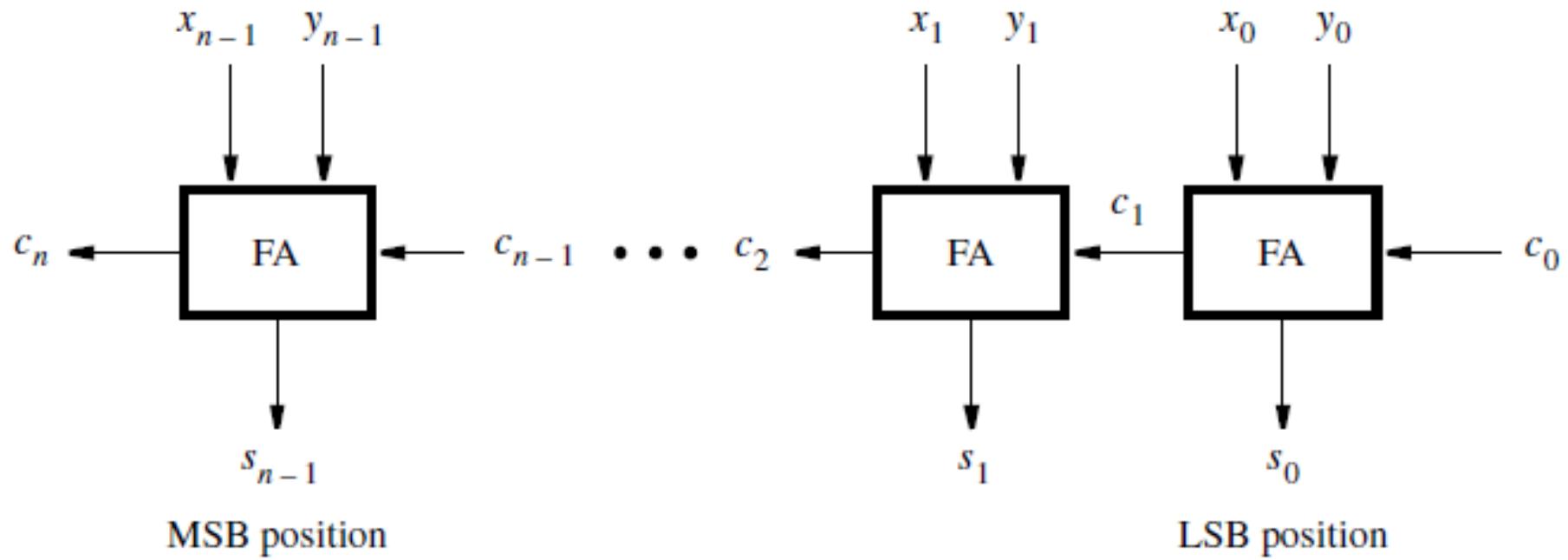
Behavioral code for example 2 is as follows:

```
module example2 (x1, x2, x3, x4, f, g, h);  
input x1, x2, x3, x4;  
output f, g, h;  
assign g = (x1 & x3) | (x2 & x4);  
assign h = (x1 | x3) & (x2 | x4);  
assign f = g | h;  
endmodule
```

Operator type	Operator Symbols	Operation Performed	Number of operands
Bitwise	$\sim$ $\&$ $ $ $^$ $\sim^$ or $^{~~}$	1's complement Bitwise AND Bitwise OR Bitwise XOR Bitwise XNOR	1 2 2 2 2
Logical	! $\&\&$ $\ $	NOT AND OR	1 2 2
Reduction	$\&$ $\sim\&$ $ $ $\sim $ $^$ $\sim^$ or $^{~~}$	Reduction AND Reduction NAND Reduction OR Reduction NOR Reduction XOR Reduction XNOR	1 1 1 1 1 1
Arithmetic	$+$ $-$ $*$ $/$	Addition Subtraction 2's complement Multiplication Division	2 2 1 2 2
Relational	$>$ $<$ $\geq$ $\leq$	Greater than Lesser than Greater than or equal to Lesser than or equal to	2 2 2 2
Equality	$\equiv$ $\neq$	Logical equality Logical inequality	2 2
Shift	$\gg$ $\ll$	Right shift Left shift	2 2
Concatenation	{}	Concatenation	Any number
Replication	{()}	Replication	Any number
Conditional	?:	Conditional	3

# Hierarchical Verilog Code

For larger designs, it is often convenient to create a hierarchical structure in the Verilog code, in which there is a *top-level* module that includes multiple instances of *lower-level* modules.



Let  $n=4$

```
module adder4 (carryin, x3, x2, x1, x0, y3, y2, y1, y0, s3, s2, s1, s0, carryout);
    input carryin, x3, x2, x1, x0, y3, y2, y1, y0;
    output s3, s2, s1, s0, carryout;
    fulladd stage0 (carryin, x0, y0, s0, c1);
    fulladd stage1 (c1, x1, y1, s1, c2);
    fulladd stage2 (c2, x2, y2, s2, c3);
    fulladd stage3 (c3, x3, y3, s3, carryout);
endmodule
```

```
module fulladd (Cin, x, y, s, Cout);
    input Cin, x, y;
    output s, Cout;
    assign s = x & y & Cin;
    assign Cout = (x & y) | (x & Cin) | (y & Cin);
endmodule
```

- A Verilog module can be included as a subcircuit in another module.
- The general form of a module instantiation statement is given below.

```
module_name instance_name (.port_name ( [expression] )
{, .port_name ( [expression] )} );
```

- The *instance\_name* can be any legal Verilog identifier and the port connections specify how the module is connected to the rest of the circuit.
- The same module can be instantiated multiple times in a given design provided that each instance name is unique.
- Each *port\_name* is the name of a port in the subcircuit, and each expression specifies a connection to that port.

- **Named port connections** -The syntax `.port_name` is provided so that the order of signals listed in the instantiation statement does not have to be the same as the order of the ports given in the module statement of the subcircuit.
- **Ordered port connections**-If the port connections are given in the same order as in the subcircuit, then `.port_name` is not needed.

## Using Vectored Signals

- Multibit signals are called *vectors*.
- An example of an input vector is

**input** [3:0] W;

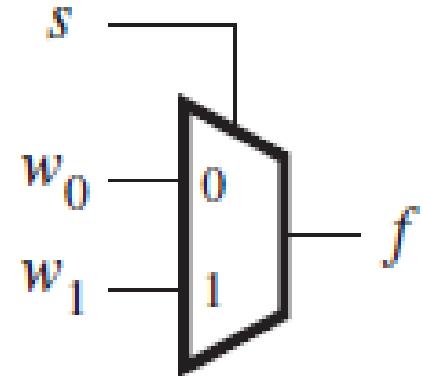
- This statement defines  $W$  to be a four-bit vector. Its individual bits can be referred to using an index value in square brackets.
- The most-significant bit (MSB) is referred to as  $W[3]$  and the least-significant bit (LSB) is  $W[0]$ .
- **input** [0:3] W; Here  $W[0]$  is MSB and  $W[3]$  LSB

```
module adder4 (carryin, X, Y, S, carryout);
input carryin;
input [3:0] X, Y;
output [3:0] S;
output carryout;
wire [3:1] C;
fulladd stage0 (carryin, X[0], Y[0], S[0], C[1]);
fulladd stage1 (C[1], X[1], Y[1], S[1], C[2]);
fulladd stage2 (C[2], X[2], Y[2], S[2], C[3]);
fulladd stage3 (.Cout(carryout), .s(S[3]), .y(Y[3]), .x(X[3]), .Cin(C[3]));
endmodule
```

# Multiplexers

A multiplexer circuit has a number of data inputs, one or more select inputs, and one output. It passes the signal value on one of the data inputs to the output. The data input is selected by the values of the select inputs.

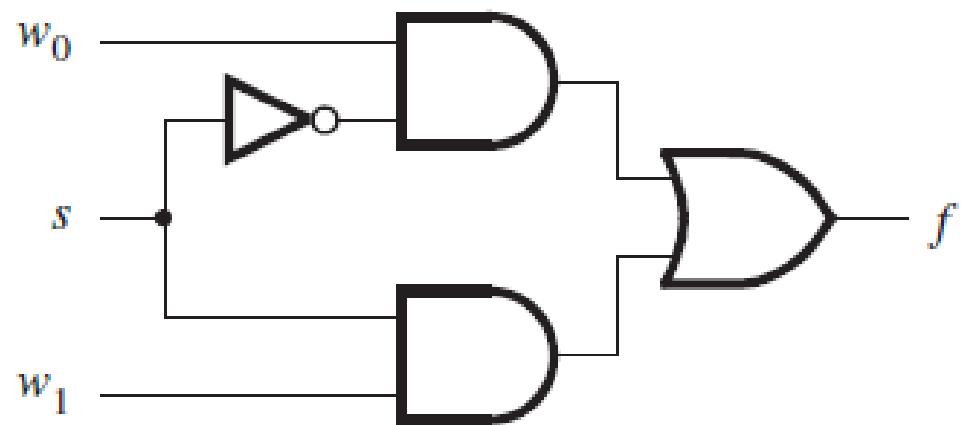
# 2-to-1 multiplexer.



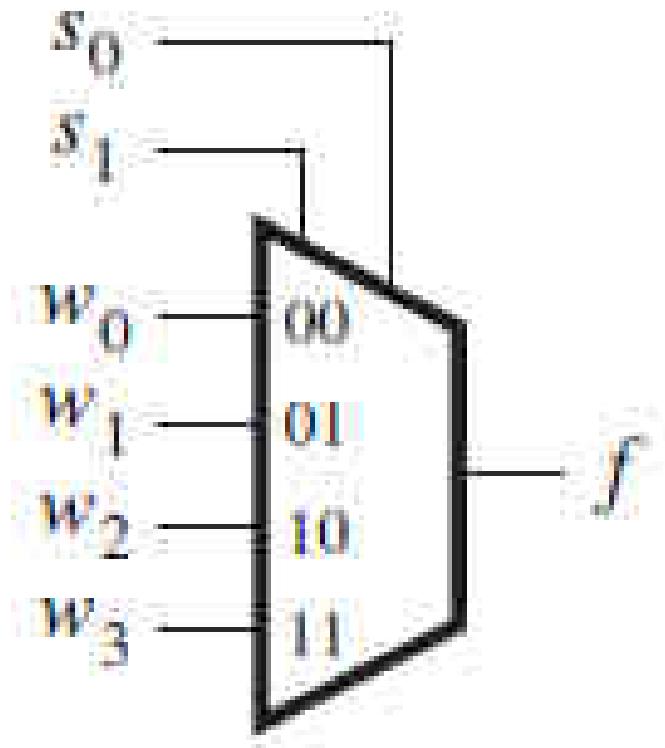
(a) Graphical symbol

$s$	$f$
0	$w_0$
1	$w_1$

(b) Truth table



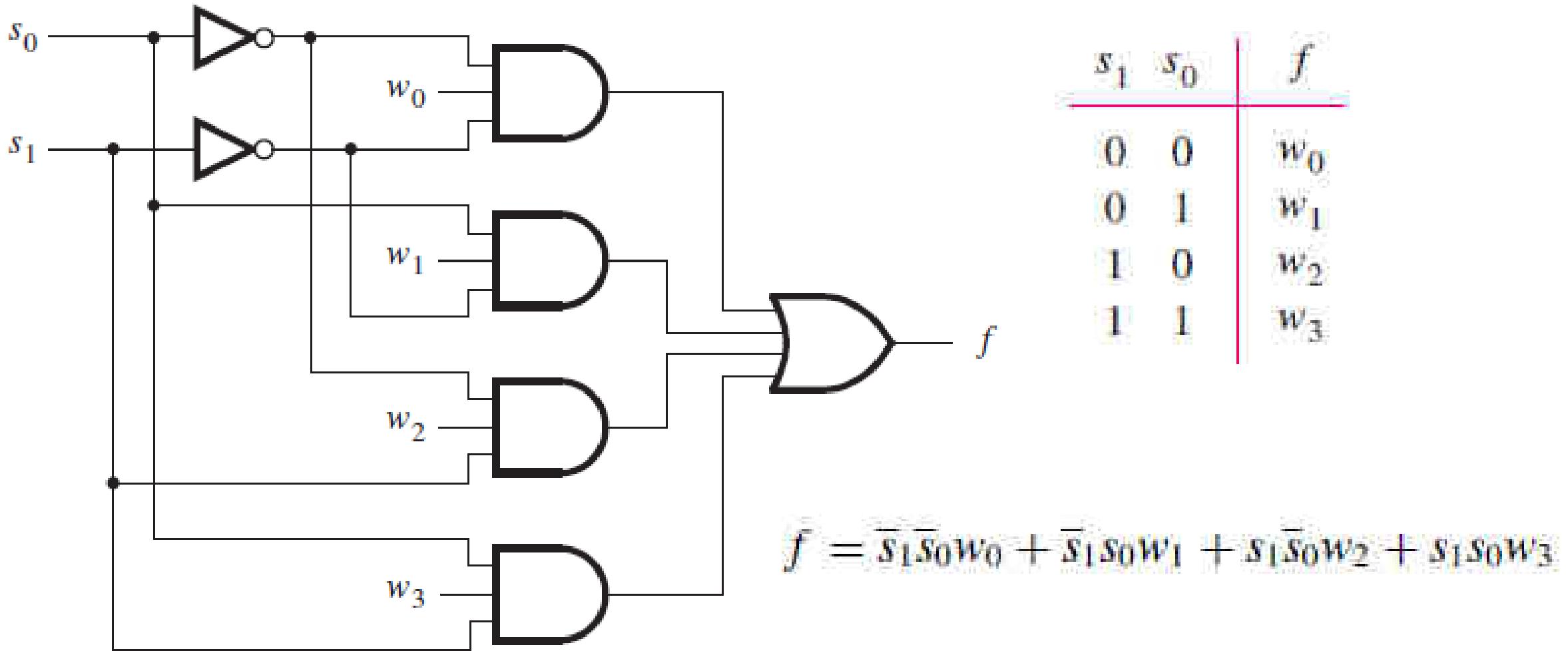
# 4-to-1 multiplexer.



(a) Graphical symbol

$s_1$	$s_0$	$f$	
0	0	$w_0$	
0	1	$w_1$	
1	0	$w_2$	
1	1	$w_3$	

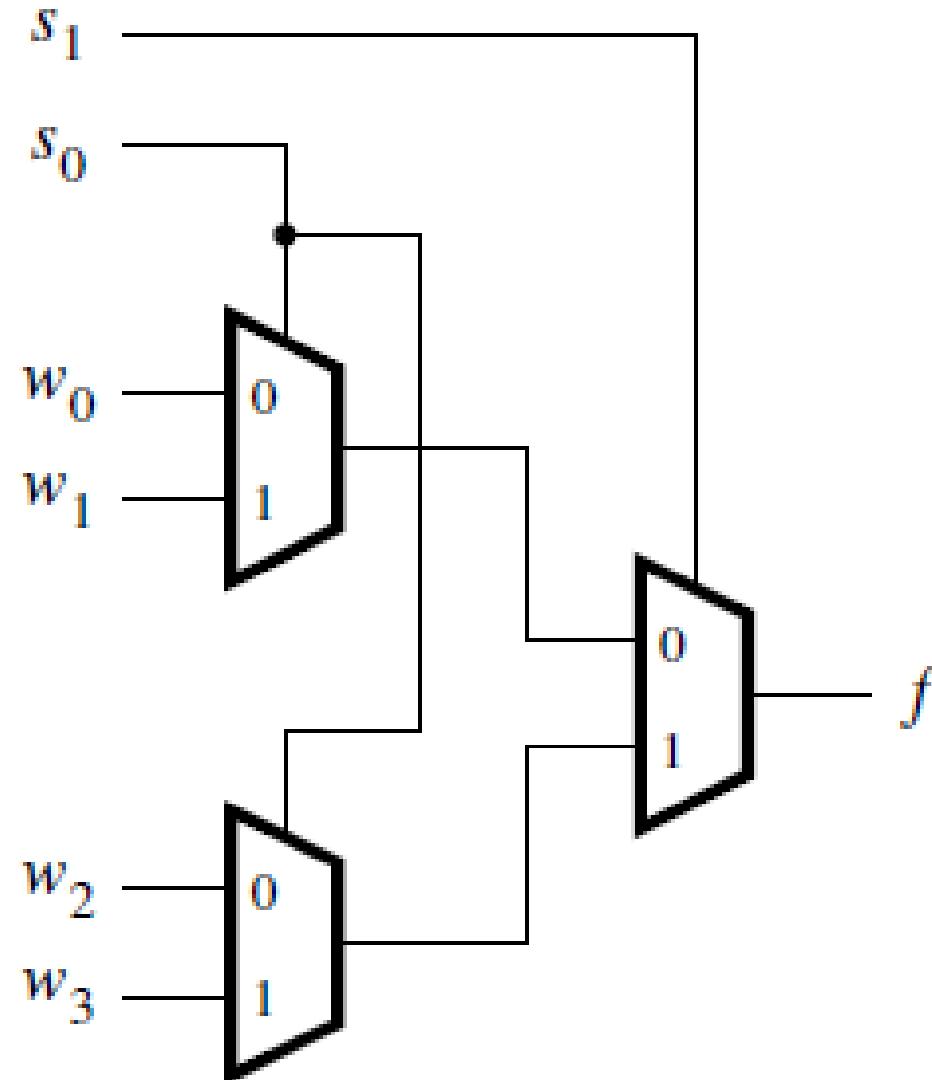
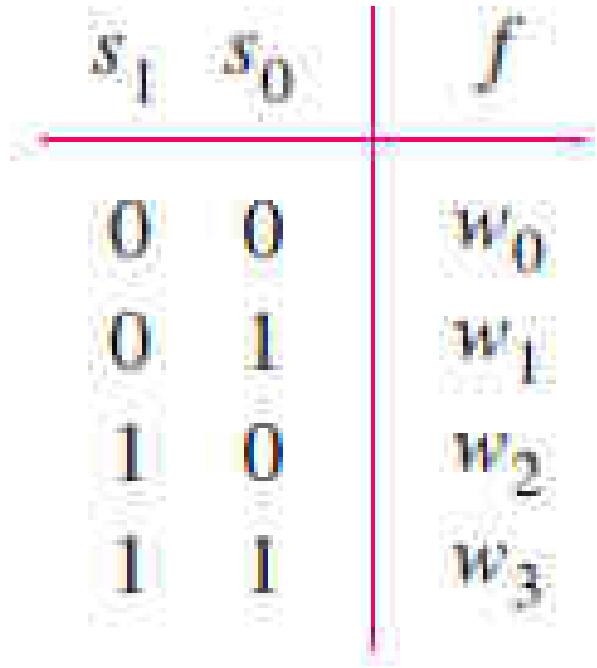
(b) Truth table



(c) Circuit

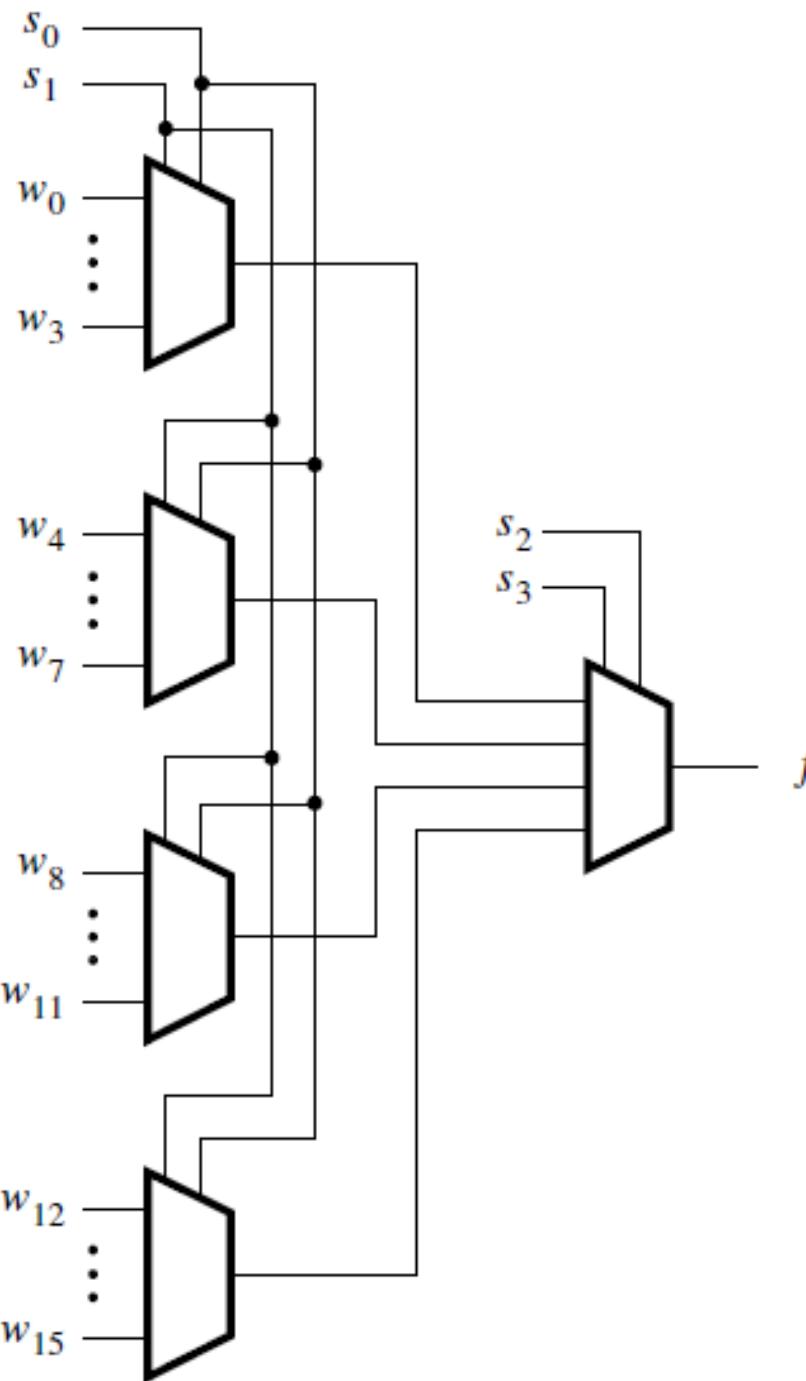
Usually, the number of data inputs,  $n$ , is an integer power of two. A multiplexer that has  $n$  data inputs,  $w_0, \dots, w_{n-1}$ , requires  $\lceil \log_2 n \rceil$  select inputs and it has one output.

Larger multiplexers can also be constructed from smaller multiplexers. For example, the 4-to-1 multiplexer can be built using three 2-to-1 multiplexers as shown below:



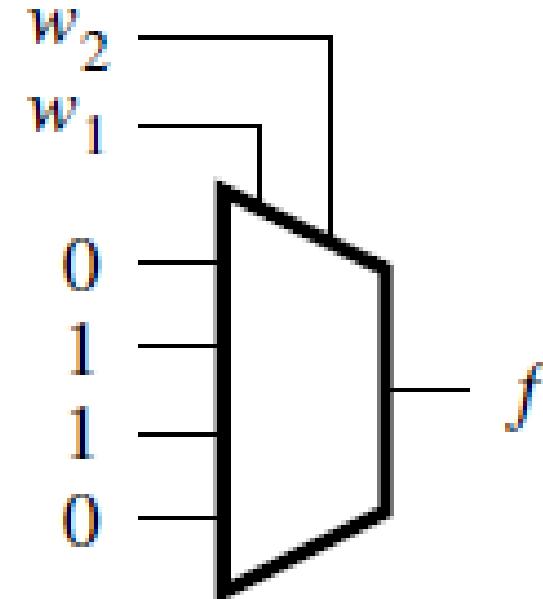
Using 2-to-1 multiplexers to build a 4-to-1 multiplexer.

Similarly, 16-to-1 multiplexer can be constructed with five 4-to-1 multiplexers.



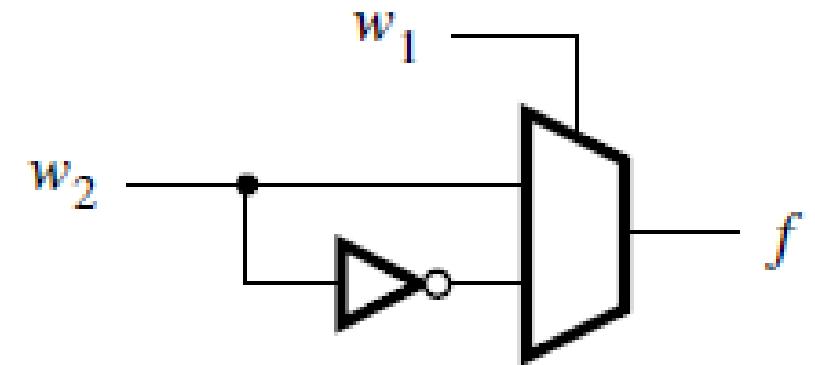
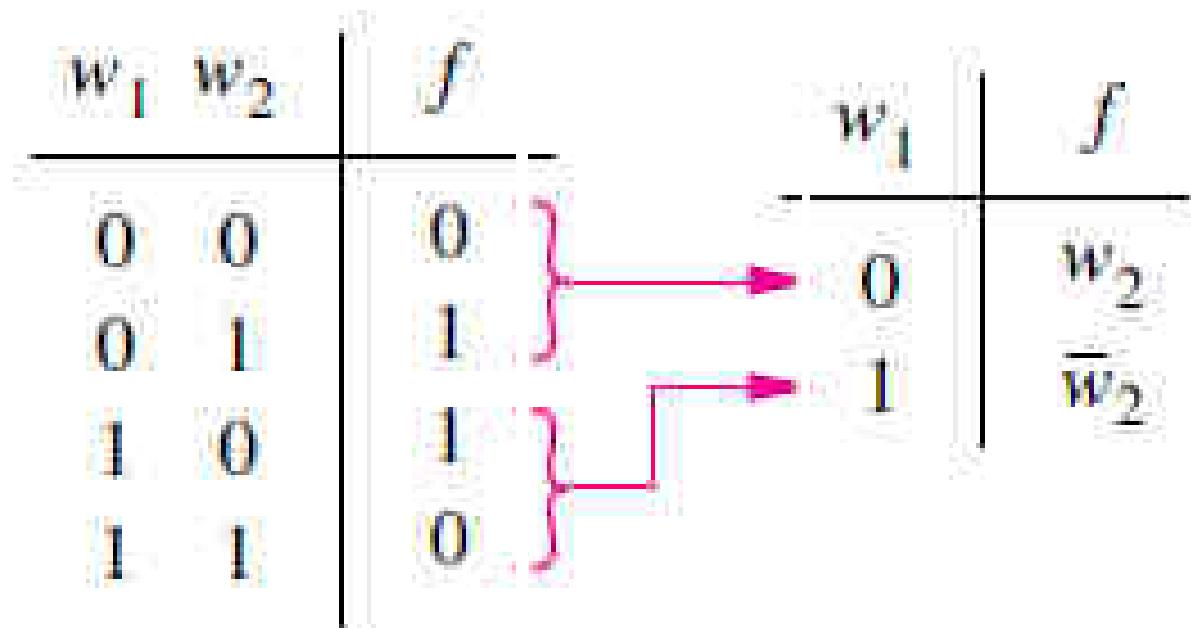
# Synthesis of Logic Functions Using Multiplexers

$w_1$	$w_2$	$f$
0	0	0
0	1	1
1	0	1
1	1	0



(a) Implementation using a 4-to-1 multiplexer

## Example 1 : Two input XOR function using 2 to 1 multiplexer

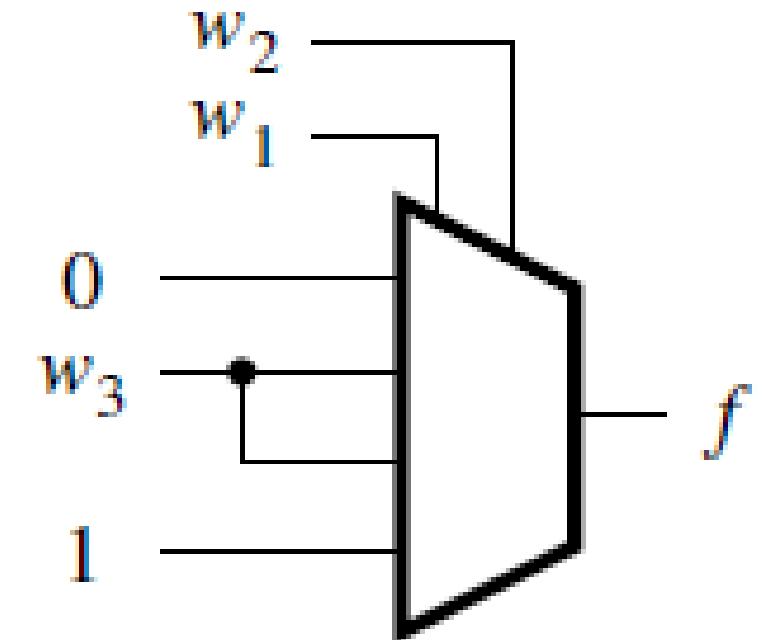


(c) Circuit

## Example 2 : Three input majority function using 4 to 1 multiplexer

$w_1$	$w_2$	$w_3$	$f$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

A logic circuit diagram showing a 4-to-1 multiplexer (MUX) implementation of a three-input majority function. The MUX has three data inputs ( $w_1$ ,  $w_2$ ,  $w_3$ ) and one control input ( $w_3$ ). The outputs are labeled  $w_1$ ,  $w_2$ , and  $w_3$ . The control input  $w_3$  is connected to the  $S$  (Select) input of the MUX. The data inputs  $w_1$  and  $w_2$  are connected to the  $D_0$  and  $D_1$  inputs respectively. The output  $w_1$  is connected to the  $D_2$  input, and the output  $w_2$  is connected to the  $D_3$  input. The output  $w_3$  is the sum of the three data inputs ( $w_1 + w_2 + w_3$ ).



(b) Circuit

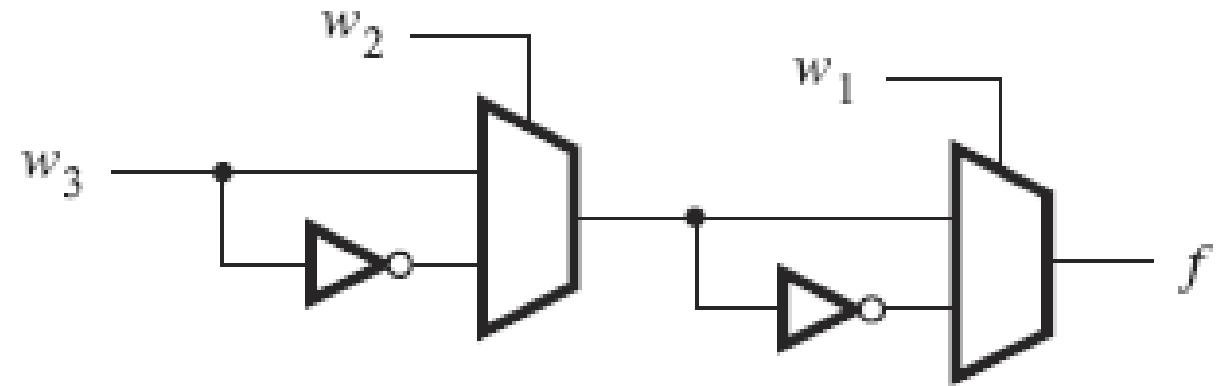
## Example 3: Implement a three-input XOR gate using 2-to-1 multiplexer.

$w_1$	$w_2$	$w_3$	$f$
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

$$w_2 \oplus w_3$$

$$\overline{w_2 \oplus w_3}$$

(a) Truth table

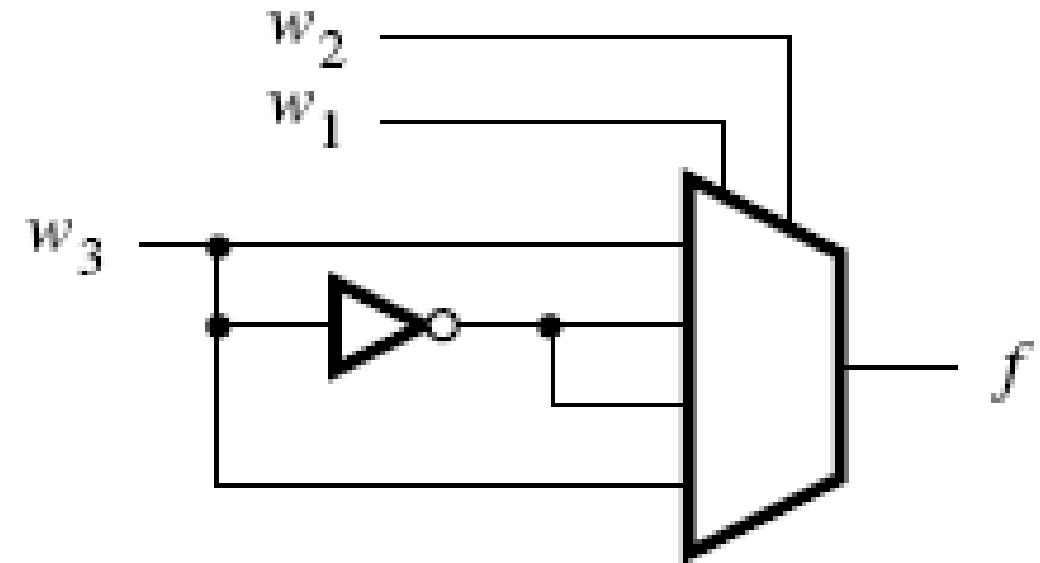


(b) Circuit

Example 4: Implement a three-input XOR gate using 4-to-1 multiplexer.

$w_1$	$w_2$	$w_3$	$f$
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

(a) Truth table



(b) Circuit

# Multiplexer Synthesis Using Shannon's Expansion

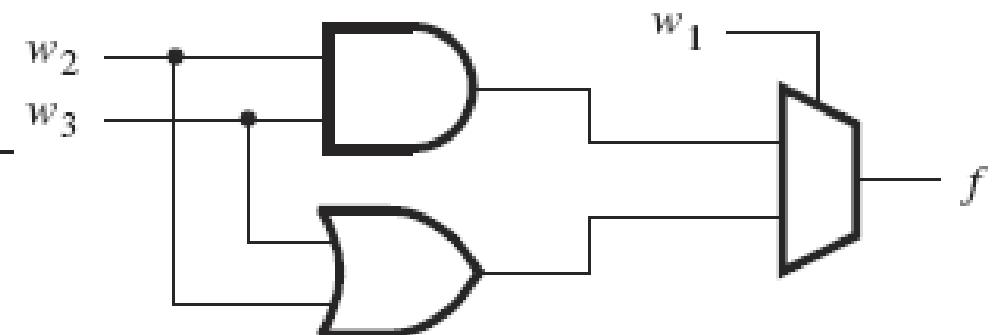
The inputs to the multiplexers are the constants 0 and 1, or some variable or its complement. Besides using such simple inputs, it is possible to connect more complex circuits as inputs to a multiplexer, allowing functions to be synthesized using a combination of multiplexers and other logic gates.

## Example 5 : Three input majority function using 2 to 1 multiplexer

$w_1$	$w_2$	$w_3$	$f$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(a) Truth table

$w_1$	$f$
0	$w_2 w_3$
1	$w_2 + w_3$



(b) Circuit

# Shannon's Expansion

This implementation can be derived using algebraic manipulation as follows. The function is expressed in sum-of-products form as

$w_1$	$w_2$	$w_3$	$f$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$$f = \overline{w}_1 w_2 w_3 + w_1 \overline{w}_2 w_3 + w_1 w_2 \overline{w}_3 + w_1 w_2 w_3$$

It can be manipulated into

$$\begin{aligned} f &= \overline{w}_1 (w_2 w_3) + w_1 (\overline{w}_2 w_3 + w_2 \overline{w}_3 + w_2 w_3) \\ &= \overline{w}_1 (w_2 w_3) + w_1 (w_2 + w_3) \end{aligned}$$

Multiplexer implementations of logic functions require that a given function be decomposed in terms of the variables that are used as the select inputs. This can be accomplished by means of a theorem proposed by Claude Shannon

**Shannon's Expansion Theorem** Any Boolean function  $f(w_1, \dots, w_n)$  can be written in the form

$$f(w_1, w_2, \dots, w_n) = \bar{w}_1 \cdot f(0, w_2, \dots, w_n) + w_1 \cdot f(1, w_2, \dots, w_n)$$

In Shannon's expansion the term  $f(0, w_2, \dots, w_n)$  is called the *cofactor* of  $f$  with respect to  $\bar{w}_1$ ; it is denoted in shorthand notation as  $f_{\bar{w}_1}$ . Similarly, the term  $f(1, w_2, \dots, w_n)$  is called the cofactor of  $f$  with respect to  $w_1$ , written  $f_{w_1}$ . Hence we can write

$$f = \bar{w}_1 f_{\bar{w}_1} + w_1 f_{w_1}$$

In general, if the expansion is done with respect to variable  $w_i$ , then  $f_{\bar{w}_i}$  denotes  $f(w_1, \dots, w_{i-1}, 0, w_{i+1}, \dots, w_n)$ ,  $f_{w_i}$  denotes  $f(w_1, \dots, w_{i-1}, 1, w_{i+1}, \dots, w_n)$ , and

$$f(w_1, \dots, w_n) = \bar{w}_i f_{\bar{w}_i} + w_i f_{w_i}$$

## Example 1

$$f(w_1, w_2, w_3) = w_1w_2 + w_1w_3 + w_2w_3$$

Expanding this function in terms of  $w_1$  gives

$$\begin{aligned} f &= \bar{w}_1(0 \cdot w_2 + 0 \cdot w_3 + w_2w_3) + w_1(1 \cdot w_2 + 1 \cdot w_3 + w_2w_3) \\ &= \bar{w}_1(w_2w_3) + w_1(w_2 + w_3) \end{aligned}$$

## Example 2

For the three-input XOR function, we have

$$\begin{aligned}f &= w_1 \oplus w_2 \oplus w_3 \\&= \overline{w}_1(0 \oplus w_2 \oplus w_3) + w_1(1 \oplus w_2 \oplus w_3) \\&= \overline{w}_1 \cdot (w_2 \oplus w_3) + w_1 \cdot (\overline{w_2 \oplus w_3})\end{aligned}$$

### Example 3

$$f = \overline{w}_1 w_3 + w_2 \overline{w}_3.$$

Decomposition using  $w_1$  gives

$$\begin{aligned} f &= \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1} \\ &= \overline{w}_1(w_3 + w_2) + w_1(w_2 \overline{w}_3) \end{aligned}$$

Decomposition using  $w_3$  gives

$$\begin{aligned} f &= \overline{w}_3 f_{\overline{w}_3} + w_3 f_{w_3} \\ &= \overline{w}_3(w_2) + w_3(\overline{w}_1) \end{aligned}$$

Using  $w_2$  instead of  $w_1$  gives

$$\begin{aligned} f &= \overline{w}_2 f_{\overline{w}_2} + w_2 f_{w_2} \\ &= \overline{w}_2(\overline{w}_1 w_3) + w_2(\overline{w}_1 w_3 + \overline{w}_3) \\ &= \overline{w}_2(\overline{w}_1 w_3) + w_2(\overline{w}_1 + \overline{w}_3) \end{aligned}$$

Shannon's expansion can be done in terms of more than one variable. For example, expanding a function in terms of  $w_1$  and  $w_2$  gives

$$\begin{aligned}f(w_1, \dots, w_n) &= \bar{w}_1 \bar{w}_2 \cdot f(0, 0, w_3, \dots, w_n) + \bar{w}_1 w_2 \cdot f(0, 1, w_3, \dots, w_n) \\&\quad + w_1 \bar{w}_2 \cdot f(1, 0, w_3, \dots, w_n) + w_1 w_2 \cdot f(1, 1, w_3, \dots, w_n)\end{aligned}$$

Example 4: Implement the following function using

i) 2-to-1 multiplexers

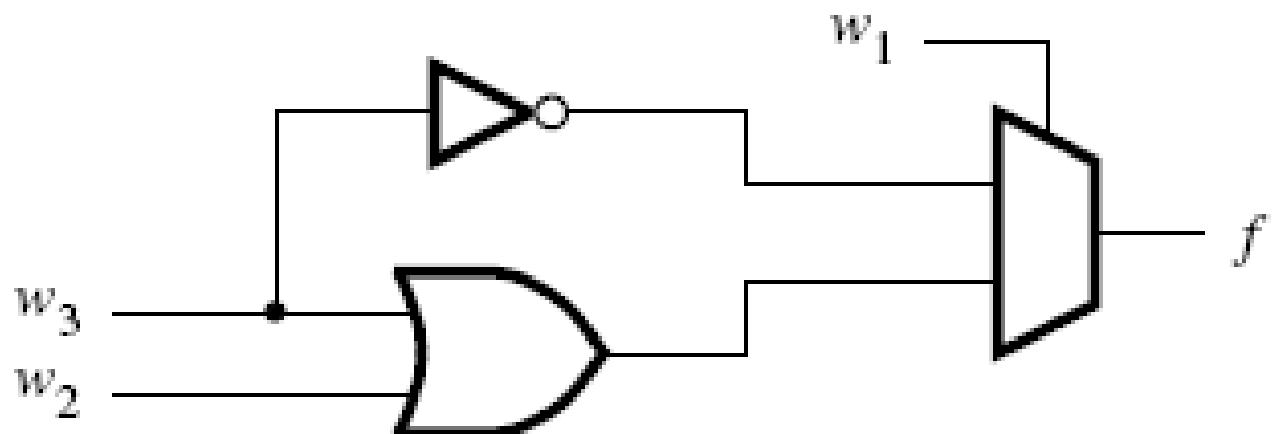
ii) 4-to-1 multiplexers

and any other necessary gates

$$f = \overline{w_1} \overline{w_3} + w_1 w_2 + w_1 w_3$$

Shannon's expansion using  $w_1$  gives

$$\begin{aligned} f &= \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1} \\ &= \overline{w}_1 (\overline{w}_3) + w_1 (w_2 + w_3) \end{aligned}$$

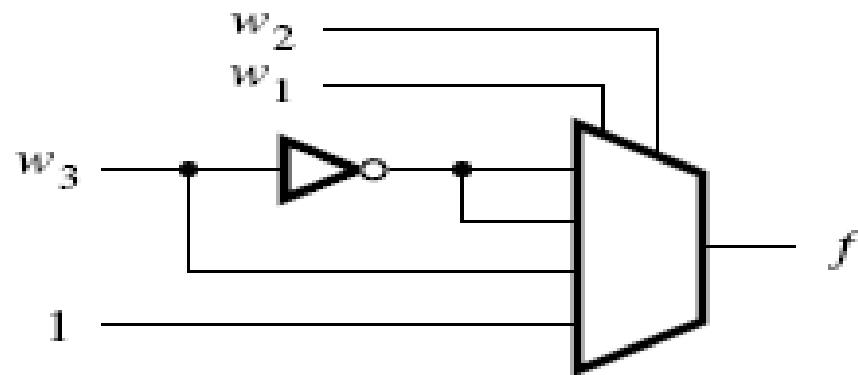


(a) Using a 2-to-1 multiplexer

## Example 4 (Contd...)

Shannon's expansion using both  $w_1$  and  $w_2$  gives

$$\begin{aligned}f &= \overline{w_1} \overline{w_2} f_{\overline{w_1} \overline{w_2}} + \overline{w_1} w_2 f_{\overline{w_1} w_2} + w_1 \overline{w_2} f_{w_1 \overline{w_2}} + w_1 w_2 f_{w_1 w_2} \\&= \overline{w_1} \overline{w_2} (\overline{w_3}) + \overline{w_1} w_2 (\overline{w_3}) + w_1 \overline{w_2} (w_3) + w_1 w_2 (1)\end{aligned}$$



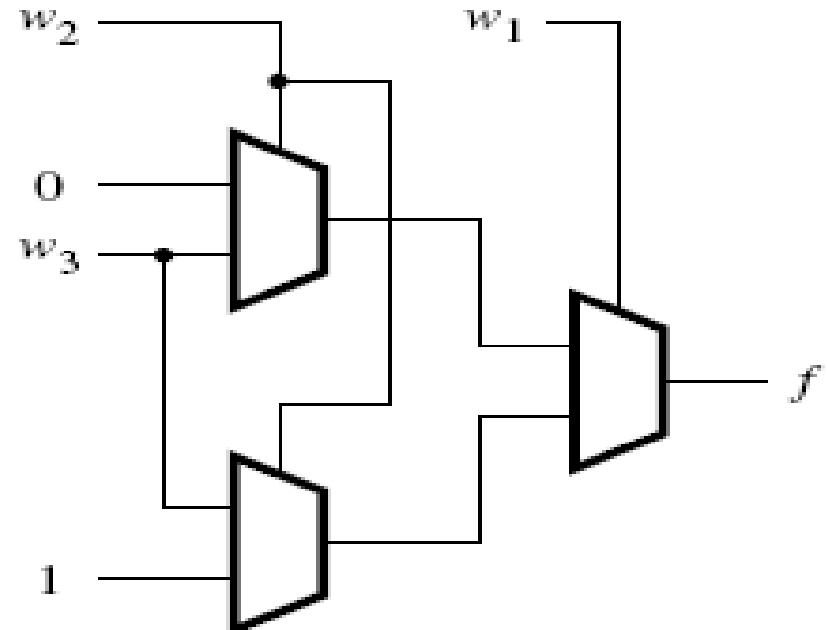
(b) Using a 4-to-1 multiplexer

## Example 5: Implement the following function using only 2-to-1 multiplexers

$$f = w_1 w_2 + w_1 w_3 + w_2 w_3$$

Shannon's expansion using  $w_1$  gives

$$\begin{aligned} f &= \overline{w}_1(w_2 w_3) + w_1(w_2 + w_3 + w_2 w_3) \\ &= \overline{w}_1(w_2 w_3) + w_1(w_2 + w_3) \end{aligned}$$



Let  $g = w_2 w_3$  and  $h = w_2 + w_3$ . Expansion of both  $g$  and  $h$  using  $w_2$  gives

$$g = \overline{w}_2(0) + w_2(w_3)$$

$$h = \overline{w}_2(w_3) + w_2(1)$$

Consider the function  $f = \overline{w_1}\overline{w_3} + w_2\overline{w_3} + \overline{w_1}w_2$ . Use the truth table to derive a circuit for  $f$  that uses a 2-to-1 multiplexer.

Repeat Problem 4.3 for the function  $f = \overline{w_2}\overline{w_3} + w_1w_2$ .

For the function  $f(w_1, w_2, w_3) = \sum m(0, 2, 3, 6)$ , use Shannon's expansion to derive an implementation using a 2-to-1 multiplexer and any other necessary gates.

Repeat Problem 4.5 for the function  $f(w_1, w_2, w_3) = \sum m(0, 4, 6, 7)$ .

# Decoders

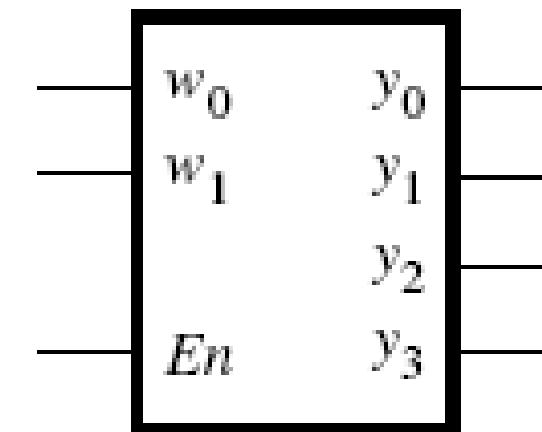
- A binary decoder is a logic circuit with  $n$  inputs and  $2^n$  outputs.
- Only one output is asserted at a time, and each output corresponds to one valuation of the inputs.
- The decoder also has an enable input  $En$
- If  $En=1$ , the valuation of  $w_{n-1} \dots w_1 w_0$  determines which of the outputs is asserted.
- If  $En=0$ , then none of the decoder outputs is asserted.

A  $k$ -bit binary code in which exactly one of the bits is set to 1 at a time is referred to as *one-hot encoded*, meaning that the single bit that is set to 1 is deemed to be “hot.” The outputs of an enabled binary decoder are one-hot encoded.

# 2-to-4 Decoders

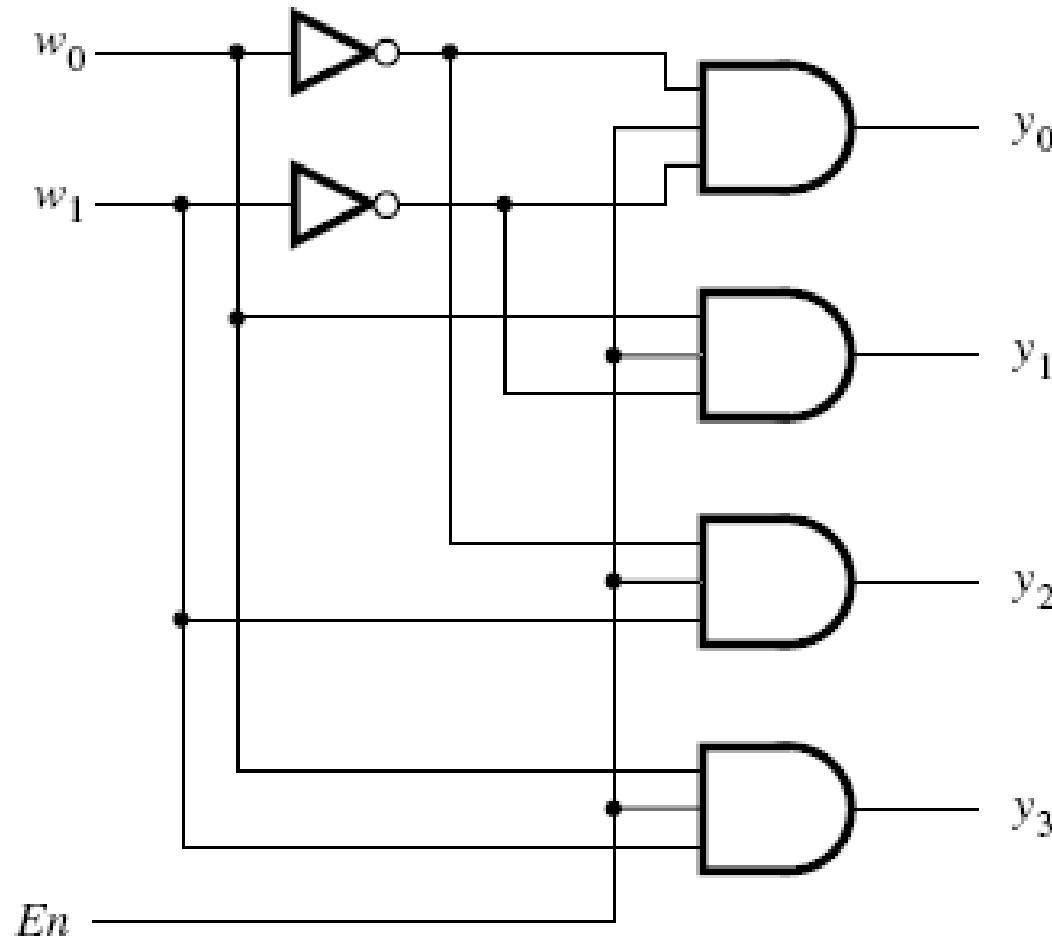
$En$	$w_1$	$w_0$	$y_0$	$y_1$	$y_2$	$y_3$
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

(a) Truth table



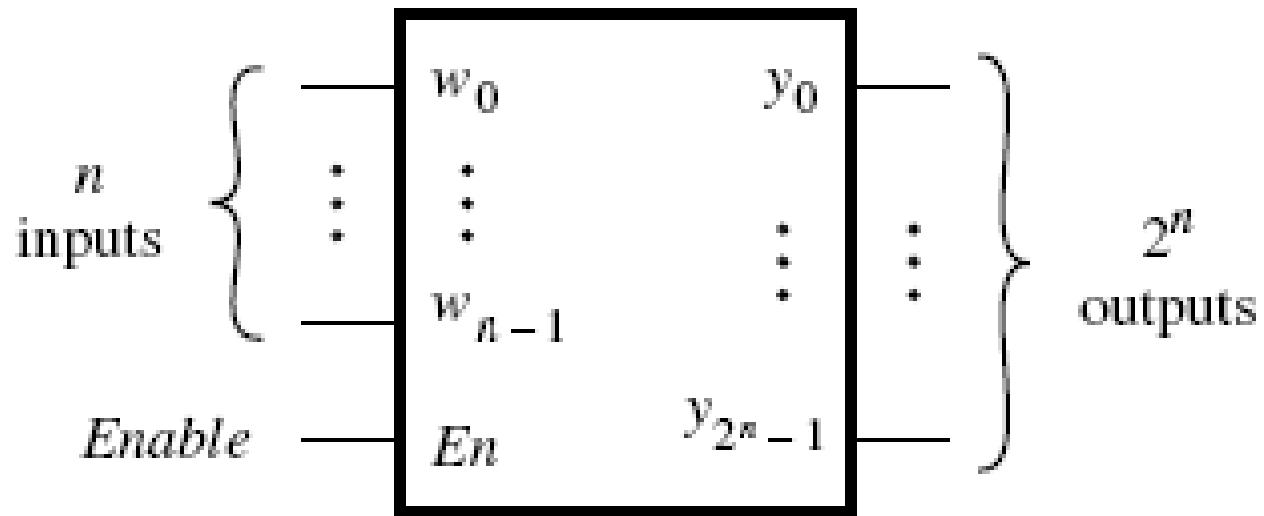
(b) Graphical symbol

# 2-to-4 Decoders



(c) Logic circuit

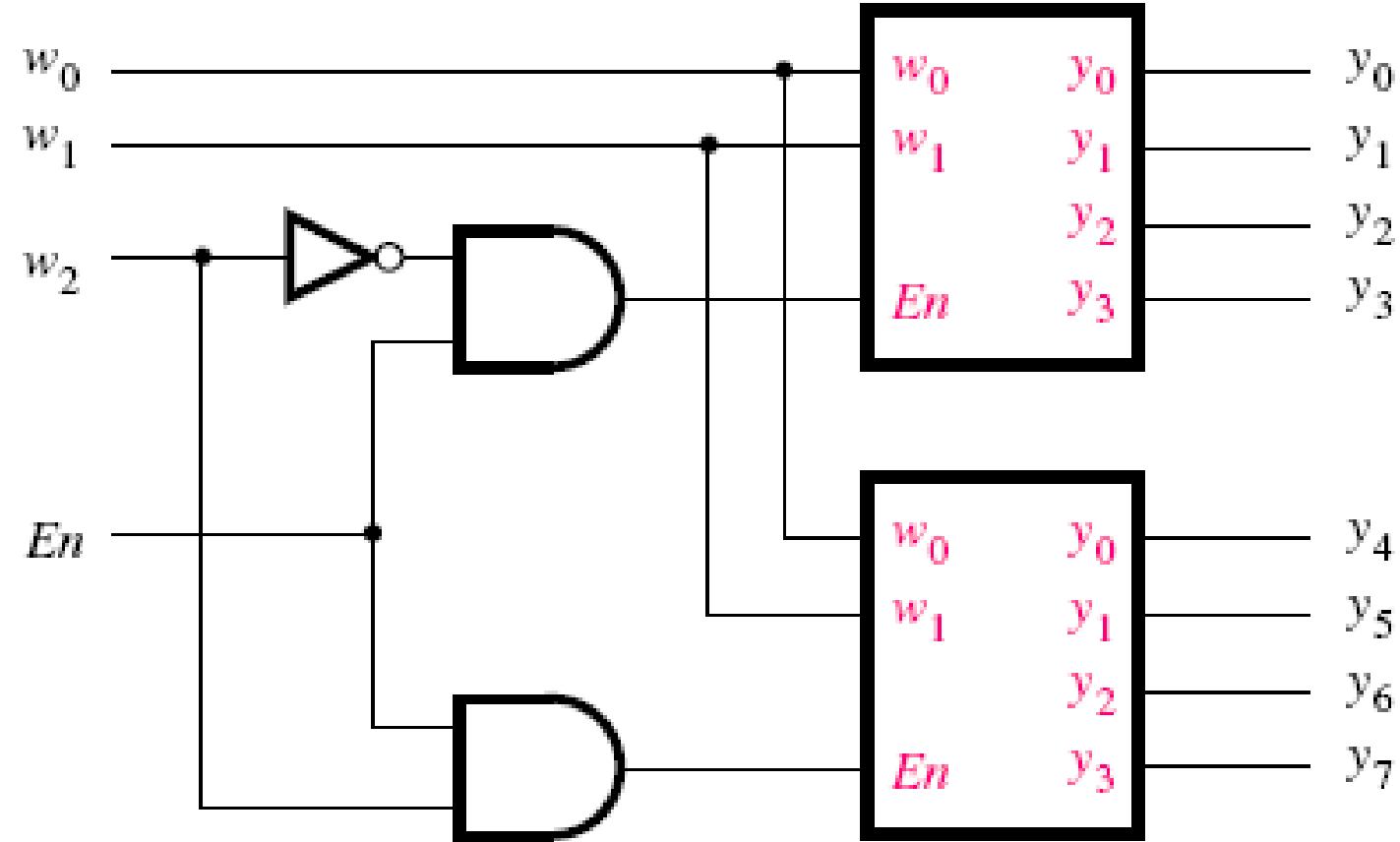
# Decoders



(d) An  $n$ -to- $2^n$  decoder

# Decoders

Larger decoders can be built using the sum-of-products structure or else they can be constructed from smaller decoders.

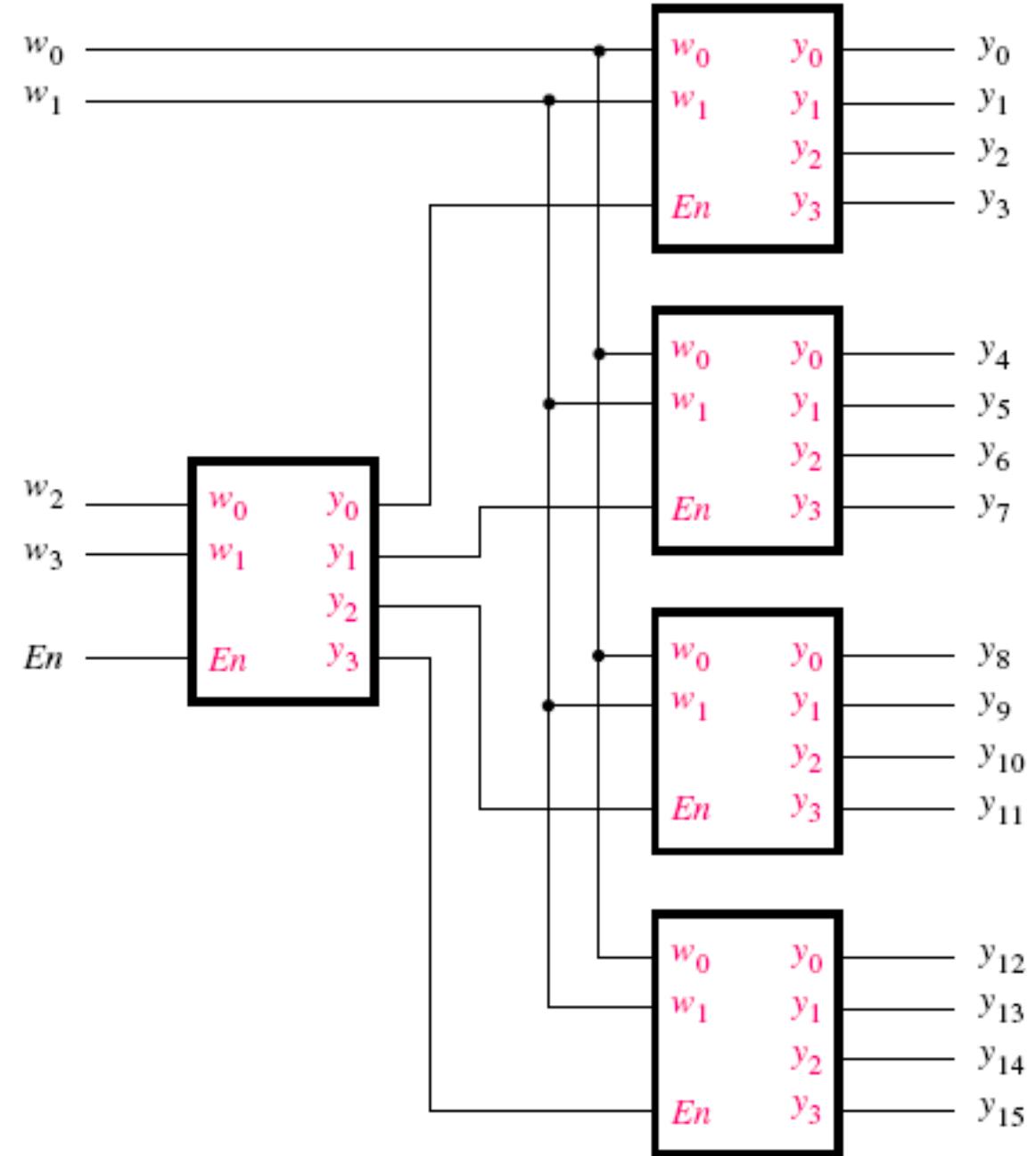


A 3-to-8 decoder using two 2-to-4 decoders.

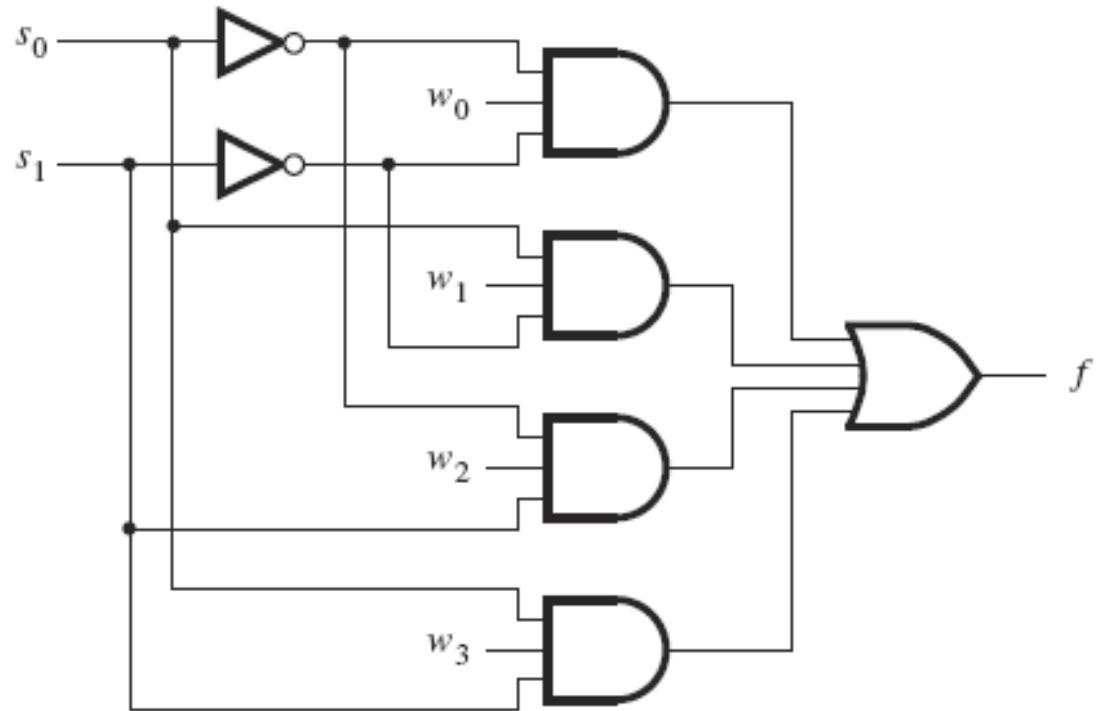
The  $w_2$  input drives the enable input of the two decoders.

# 4-to-16 Decoder

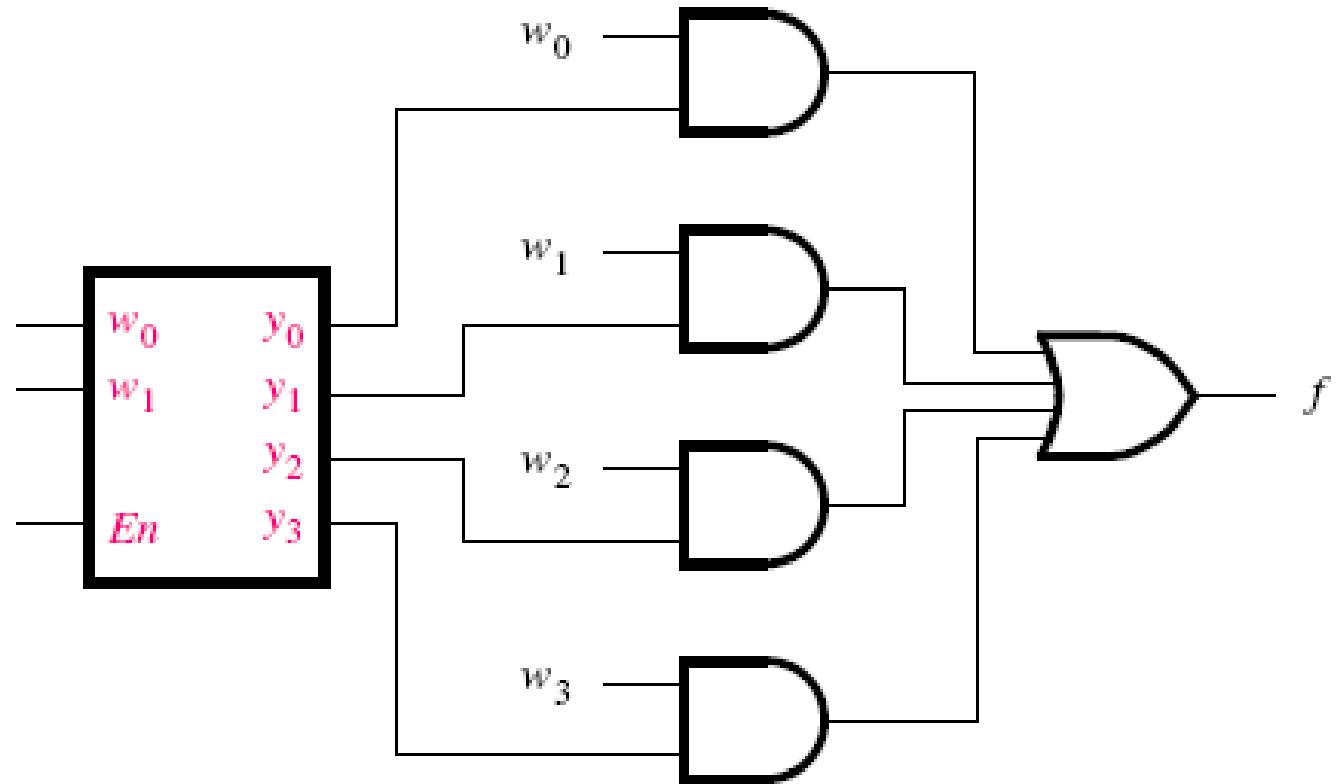
*This type of circuit is referred to as a decoder tree.*



# 4-to-1 Multiplexer using Decoder



4-to-1 Multiplexer



4-to-1 Multiplexer using decoder

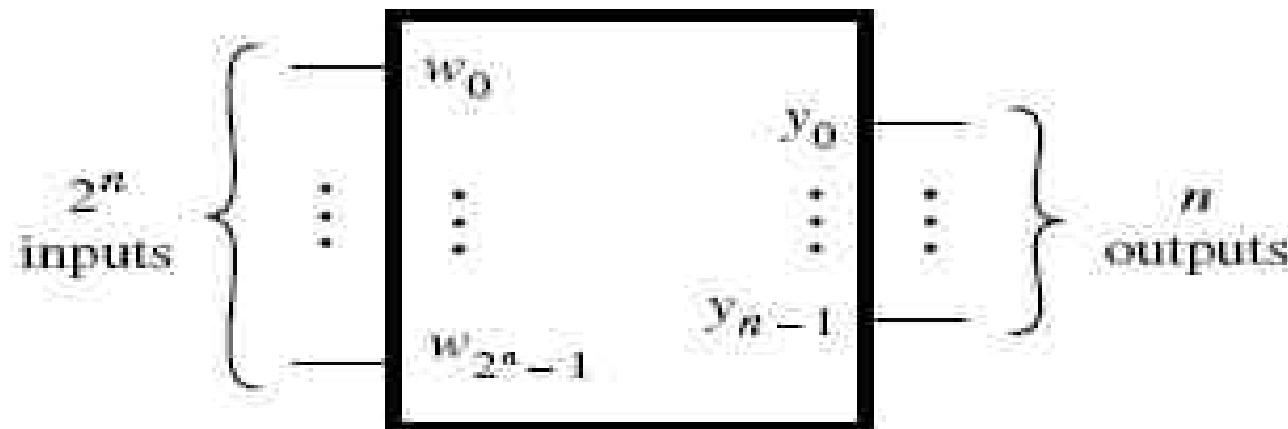
# Encoders

An encoder performs the opposite function of a decoder.

## Binary Encoders

A *binary encoder* encodes information from  $2^n$  inputs into an  $n$ -bit code.

Exactly one of the input signals should have a value of 1, and the outputs present the binary number that identifies which input is equal to 1.

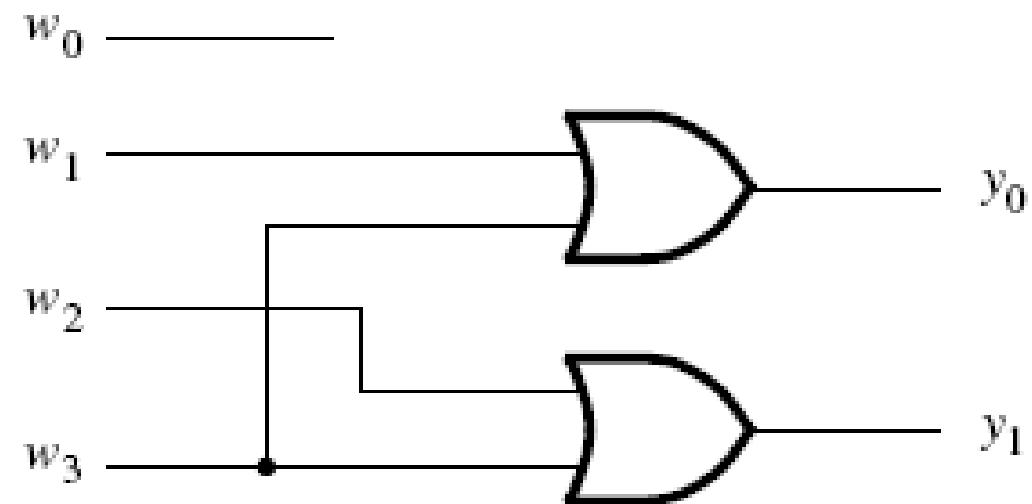


A  $2^n$ -to- $n$  binary encoder.

# 4-to-2 Binary Encoders

$w_3$	$w_2$	$w_1$	$w_0$	$y_1$	$y_0$
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

(a) Truth table



(b) Circuit

The inputs are one-hot encoded. The input patterns that have multiple inputs set to 1 are treated as don't care conditions.  
Encoders are used for transmitting information in a digital system.

# Priority Encoders

In a priority encoder each input has a priority level associated with it. The encoder outputs indicate the active input that has the highest priority. When an input with high priority is asserted, the inputs with lower priority are ignored.

$w_3$	$w_2$	$w_1$	$w_0$	$y_1$	$y_0$	$z$
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

$$i_0 = \overline{w_3} \overline{w_2} \overline{w_1} w_0 \quad y_0 = i_1 + i_3$$

$$i_1 = \overline{w_3} \overline{w_2} w_1 \quad y_1 = i_2 + i_3$$

$$i_2 = \overline{w_3} w_2$$

$$i_3 = w_3$$

$$z = i_0 + i_1 + i_2 + i_3$$

$z$  is set to 1 when at least one of the inputs is equal to 1.

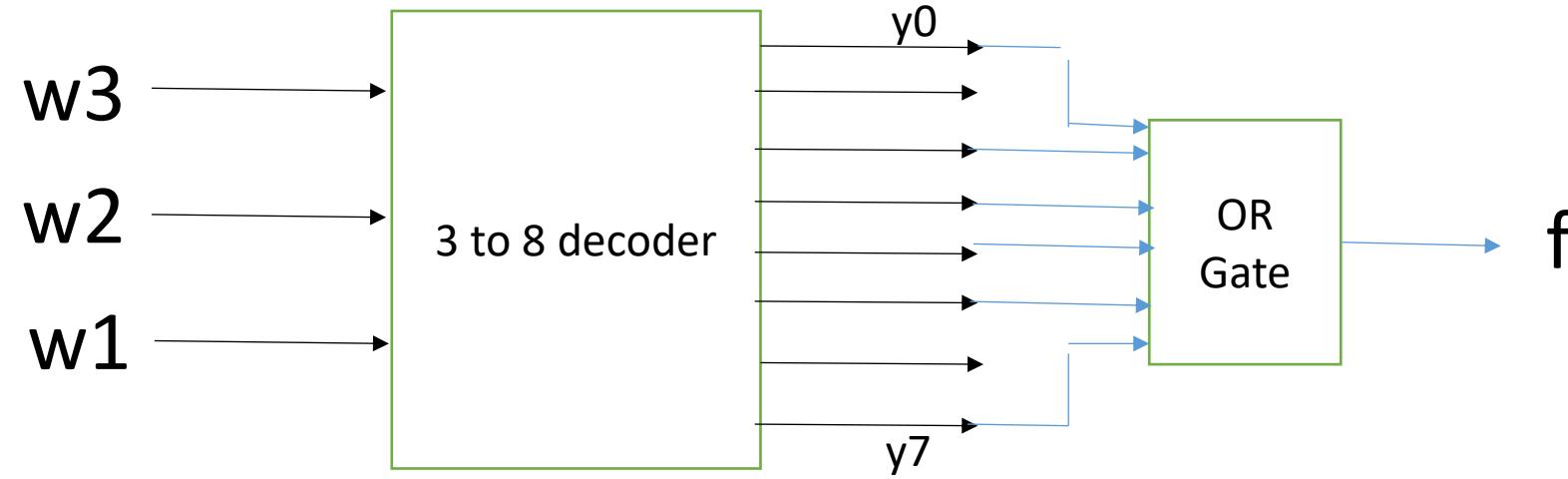
$w_0$  has the lowest priority and  $w_3$  the highest

Priority encoder having w0 highest priority and w3 least priority

w3	w2	w1	w0	y1	y0	z
x	x	x	1	0	0	1
x	x	1	0	0	1	1
x	1	0	0	1	0	1
1	0	0	0	1	1	1
0	0	0	0	x	x	0

## Problems:

1. Show how the function  $f(w_1, w_2, w_3) = \sum m(0, 2, 3, 4, 5, 7)$  can be implemented using a 3-to-8 binary decoder and an OR gate.
2. Show how the function  $f(w_1, w_2, w_3) = \sum m(1, 2, 3, 5, 6)$  can be implemented using a 3-to-8 binary decoder and an OR gate.



# Procedural Statements

Verilog provides *procedural statements* (also called *sequential statements*). Whereas concurrent statements are executed in parallel, procedural statements are evaluated in the order in which they appear in the code. Verilog syntax requires that procedural statements be contained inside an **always** block.

# The Conditional Operator

conditional\_expression ? true\_expression : false\_expression

If the conditional expression evaluates to 1 (true), then the value of true\_expression is chosen; otherwise, the value of false\_expression is chosen.

For example, the statement    A= (B < C) ? (D + 5) : (D + 2); means that if B is less than C, the value of A will be D + 5 or else A will have the value D+2.

Parentheses are not necessary. The conditional operator can be used both in continuous assignment statements and in procedural statements inside an always block.

```
module mux2to1 (w0, w1, s, f);  
input w0, w1, s;  
output f;  
assign f = s ? w1 : w0;  
endmodule
```

```
module mux2to1 (w0, w1, s, f);  
input w0, w1, s;  
output f;  
reg f;  
always @(w0 or w1 or s)  
f = s ? w1 : w0;  
endmodule
```

```
module mux4to1 (w0, w1, w2, w3, S, f);
input w0, w1, w2, w3;
input [1:0] S;
output f;
assign f = S[1] ? (S[0] ? w3 : w2) : (S[0] ? w1 : w0);
endmodule
```

# The If-Else Statement

**if-else** statement has the syntax

```
if (conditional_expression) statement;  
else statement;
```

If the expression is evaluated to true then the first statement (or a block of statements delineated by **begin** and **end** keywords) is executed or else the second statement (or a block of statements) is executed.

```
module mux2to1 (w0, w1, s, f);
input w0, w1, s;
output f;
reg f;
always @(w0 or w1 or s)
if (s == 0)
f = w0;
else
f = w1;
endmodule
```

```
module mux4to1 (w0, w1, w2, w3, S, f);
input w0, w1, w2, w3;
input [1:0] S;
output f;
reg f;
always @ (w0 or w1 or w2 or w3 or S)
if (S == 2'b00)
f = w0;
else if (S == 2'b01)
f = w1;
else if (S == 2'b10)
f = w2;
else if (S == 2'b11)
f = w3;
endmodule
```

```
module mux4to1 (W, S, f);
input [0:3] W;
input [1:0] S;
output f;
reg f;
always @(W or S)
if (S == 0)
f = W[0];
else if (S == 1)
f = W[1];
else if (S == 2)
f = W[2];
else if (S == 3)
f = W[3];
endmodule
```

```
module mux16to1 (W, S16, f);
input [0:15] W;
input [3:0] S16;
output f;
wire [0:3] M;
mux4to1 Mux1 (W[0:3], S16[1:0], M[0]);
mux4to1 Mux2 (W[4:7], S16[1:0], M[1]);
mux4to1 Mux3 (W[8:11], S16[1:0], M[2]);
mux4to1 Mux4 (W[12:15], S16[1:0], M[3]);
mux4to1 Mux5 (M[0:3], S16[3:2], f);
endmodule
```

## The Case Statement

```
case (expression)
alternative1: statement;
alternative2: statement;
·
·
·
alternativej: statement;
[default: statement;]
endcase
```

The controlling expression and each alternative are compared bit by bit. When there is one or more matching alternative, the statement(s) associated with the first match (only) is executed. When the specified alternatives do not cover all possible valuations of the controlling expression, the optional **default** clause should be included.

```
module mux4to1 (W, S, f);
input [0:3] W;
input [1:0] S;
output f;
reg f;
always @(W or S)
case (S)
 0: f = W[0];
 1: f = W[1];
 2: f = W[2];
 3: f = W[3];
endcase
endmodule
```

The four values that the select vector  $S$  can have are given as decimal numbers, but they could also be given as binary numbers.

```
module dec2to4 (W, Y, En);
    input [1:0] W;
    input En;
    output reg [0:3] Y;

    always @(W, En)
        case ({En, W})
            3'b100: Y = 4'b1000;
            3'b101: Y = 4'b0100;
            3'b110: Y = 4'b0010;
            3'b111: Y = 4'b0001;
            default: Y = 4'b0000;
        endcase

    endmodule
```

```
module dec4to16 (W, En, Y);
    input [3:0] W;
    input En;
    output [0:15] Y;
    wire [0:3] M;

    dec2to4 Dec1 (W[3:2], M[0:3], En);
    dec2to4 Dec2 (W[1:0], Y[0:3], M[0]);
    dec2to4 Dec3 (W[1:0], Y[4:7], M[1]);
    dec2to4 Dec4 (W[1:0], Y[8:11], M[2]);
    dec2to4 Dec5 (W[1:0], Y[12:15], M[3]);

    endmodule
```

```
module dec2to4 (W, En, Y);
    input [1:0] W;
    input En;
    output reg [0:3] Y;

    always @(W, En)
    begin
        if (En == 0)
            Y = 4'b0000;
        else
            case (W)
                0: Y = 4'b1000;
                1: Y = 4'b0100;
                2: Y = 4'b0010;
                3: Y = 4'b0001;
            endcase
    end
endmodule
```

```
module dec2to4 (W, En, Y);
    input [1:0] W;
    input En;
    output reg [0:3] Y;
    integer k;

    always @(W, En)
        for (k = 0; k <= 3; k = k+1)
            if ((W == k) && (En == 1))
                Y[k] = 1;
            else
                Y[k] = 0;
endmodule
```

# The Casex and Casez Statements

Verilog provides two variants of the **case** statement that treat the *z* and *x* values in a different way. The **casez** statement treats all *z* values in the case alternatives and the controlling expression as don't cares. The **casex** statement treats all *z* and *x* values as don't cares.

```
module priority (W, Y, z);
    input [3:0] W;
    output reg [1:0] Y;
    output reg z;

    always @ (W)
    begin
        z = 1;
        casex (W)
            4'b1xxx: Y = 3;
            4'b01xx: Y = 2;
            4'b001x: Y = 1;
            4'b0001: Y = 0;
            default: begin
                z = 0;
                Y = 2'bx;
            end
        endcase
    end

endmodule
```

```
module priority (W, Y, z);
    input [3:0] W;
    output reg [1:0] Y;
    output reg z;
    integer k;

    always @(W)
    begin
        Y = 2'bxx;
        z = 0;
        for (k = 0; k < 4; k = k+1)
            if (W[k])
                begin
                    Y = k;
                    z = 1;
                end
        end
    endmodule
```

# Verilog Operators

## Bitwise Operators

Bitwise operators operate on individual bits of operands. The  $\sim$  operator forms the 1's complement of the operand such that the statement  $C = \sim A$ ; produces the result  $c_2 = a_2'$ ,  $c_1 = a_1'$ , and  $c_0 = a_0'$ , where  $a_i$  and  $c_i$  are the bits of the vectors  $A$  and  $C$ .

Other bitwise operators operate on pairs of bits. The statement  
 $C = A \& B$ ;  
generates  $c_2 = a_2 \& b_2$ ,  $c_1 = a_1 \& b_1$ , and  $c_0 = a_0 \& b_0$ .

A scalar function may be assigned a value as a result of a bitwise operation on two vector operands. In this case, it is only the least-significant bits of the operands that are involved in the operation. Hence the statement  $f = A \wedge B$ ; yields  $f = a_0 \oplus b_0$ .

The bitwise operations may involve operands that include the unknown logic value  $x$ .

if  $P = 4'b101x$  and  $Q = 4'b1001$ , then  $P \& Q = 4'b100x$  while  $P | Q = 4'b1011$ .

$\wedge$	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

$\sim \wedge$	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

## Logical Operators

The ! operator has the same effect on a scalar operand as the ~ operator. Thus,  $f = !w = \sim w$ .

But the effect on a vector operand is different, namely if  $f = !A$ ; then  $f = (a_2 + a_1 + a_0)'$ .

The && operator implements the AND operation such that  $f = A \&\& B$ ; produces  $f = (a_2 + a_1 + a_0) \& (b_2 + b_1 + b_0)$ . Similarly, using the || operator in  $f = A || B$ ; gives  $f = (a_2 + a_1 + a_0) | (b_2 + b_1 + b_0)$ .

## Reduction Operators

The reduction operators perform an operation on the bits of a single vector operand and produce a one-bit result. Using the & operator in  $f = \&A;$  produces  $f = a_2 \& a_1 \& a_0.$

## Arithmetic Operators

$C = A + B;$  puts the three-bit sum of  $A$  plus  $B$  into  $C.$

The operation  $C = -A;$  places the 2's complement of  $A$  into  $C.$

The addition, subtraction, and multiplication operations are supported by most CAD synthesis tools. However, the division operation is often not supported. When the Verilog compiler encounters an arithmetic operator, it usually synthesizes it by using an appropriate module from a library.

## Relational Operators

The relational operators are typically used as conditions in **if-else** and **for** statements.

These operators function in the same way as the corresponding operators in the C programming language. An expression that uses the relational operators returns the value 1 if it is evaluated as true, and the value 0 if evaluated as false. If there are any x (unknown) or z bits in the operands, then the expression takes the value x.

```
module compare (A, B, AeqB, AgtB, AltB);
input [3:0] A, B;
output AeqB, AgtB, AltB;
reg AeqB, AgtB, AltB;
always @(A or B)
begin
    AeqB = 0;
    AgtB = 0;
    AltB = 0;
    if (A == B)
        AeqB = 1;
    else if (A > B)
        AgtB = 1;
    else
        AltB = 1;
end
endmodule
```

## Equality Operators

The expression ( $A == B$ ) is evaluated as true if  $A$  is equal to  $B$  and false otherwise.

The  $!=$  operator has the opposite effect. The result is ambiguous ( $x$ ) if either operand contains  $x$  or  $z$  values.

## Shift Operators

A vector operand can be shifted to the right or left by a number of bits specified as a constant. When bits are shifted, the vacant bit positions are filled with 0s. For example,

$B = A << 1$ ; results in  $b_2 = a_1$ ,  $b_1 = a_0$ , and  $b_0 = 0$ . Similarly,

$B = A >> 2$ ; yields  $b_2 = b_1 = 0$  and  $b_0 = a_2$ .

## Concatenate Operator

This operator concatenates two or more vectors to create a larger vector. For example,  $D = \{A, B\}$ ; defines the six-bit vector  $D = a_2a_1a_0b_2b_1b_0$ . Similarly, the concatenation  $E = \{3'b111, A, 2'b00\}$ ; produces the eight-bit vector  $E = 111a_2a_1a_000$ .

## Replication Operator

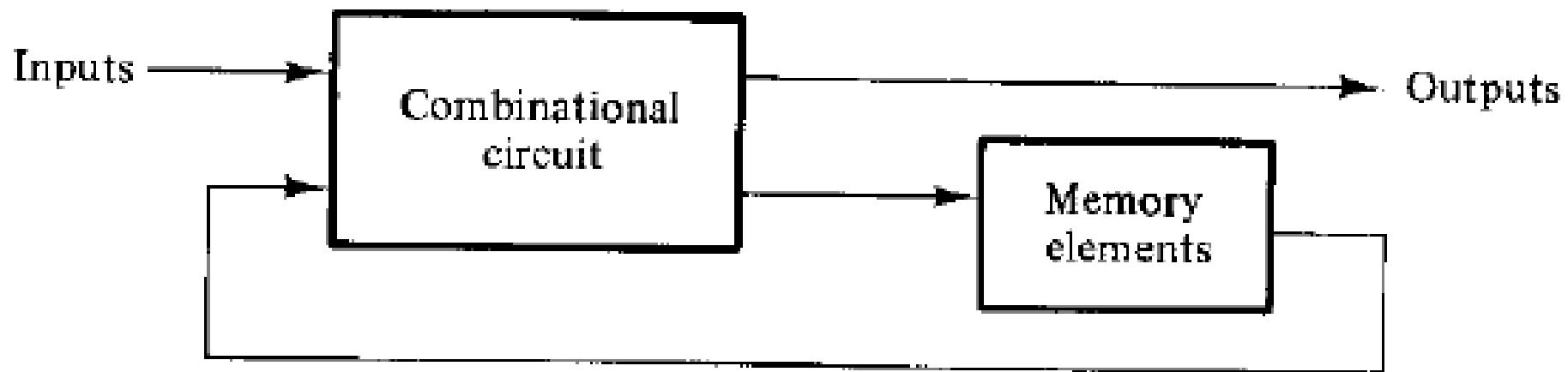
This operator allows repetitive concatenation of the same vector, which is replicated the number of times indicated in the replication constant. For example,  $\{3\{A\}\}$  is equivalent to writing  $\{A, A, A\}$ . The specification  $\{4\{2'b10\}\}$  produces the eight-bit vector  $10101010$ .

The replication operator may be used in conjunction with the concatenate operator. For instance,  $\{2\{A\}, 3\{B\}\}$  is equivalent to  $\{A, A, B, B, B\}$ .

# Synchronous Sequential Circuits

Text book: Digital Design by Morris Mano, 2<sup>nd</sup> edition, 6<sup>th</sup> Chapter

- The digital circuits considered thus far have been combinational, i.e., the outputs at any instant of time are entirely dependent upon the inputs present at that time.
- Although every digital system is likely to have combinational circuits, most systems encountered in practice also include memory elements, which require that the system be described in terms of sequential.



**FIGURE 6-1**  
Block diagram of a sequential circuit

- In the diagram, the memory elements are devices capable of storing binary information within them.
  - The binary information stored in the memory elements at any given time defines the state of the sequential circuit.
  - The sequential circuit receives binary information from external inputs. These inputs, together with the present state of the memory elements, determine the binary value at the output terminals.
  - They also determine the condition for changing the state in the memory elements.
  - The block diagram demonstrates that the external outputs in a sequential circuit are a function not only of external inputs, but also of the present state of the memory elements. The next state of the memory 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.

Two main types of sequential circuits. :

- Synchronous sequential circuit
- Asynchronous sequential circuit

A synchronous sequential circuit is a system whose behavior can be defined from the knowledge of its signals at discrete instants of time. Synchronization is achieved by a timing device called a clock generator, which provides a clock signal having the form of a periodic train of clock pulses. The activity within the circuit and the resulting updating of stored values is synchronized to the occurrence of clock pulses.

The behavior of an asynchronous sequential circuit depends upon the order in which its input signals change and can be affected at any instant of time.

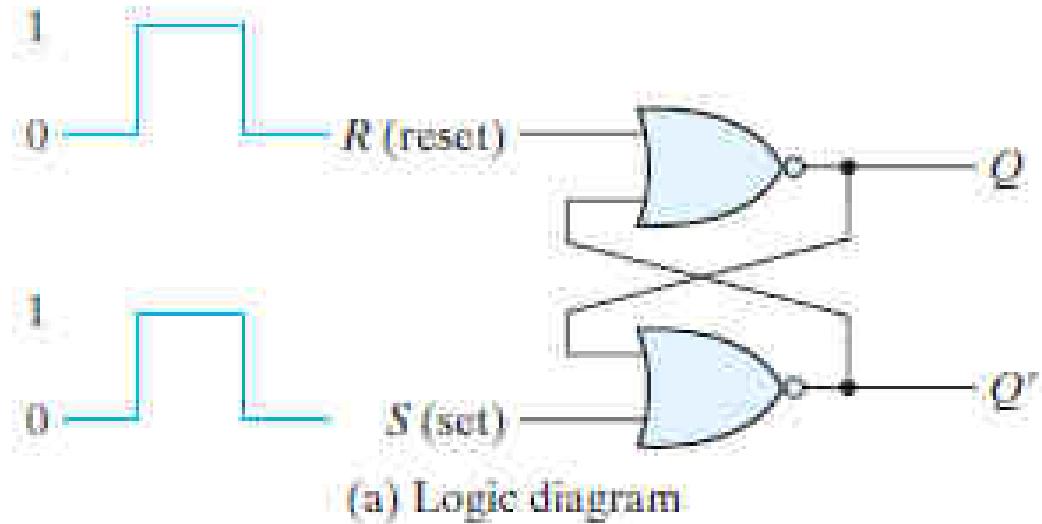
# Flip Flop

A flip-flop 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 flip-flops are in the number of inputs they possess and in the manner in which the inputs affect the binary state. The most common types of flip-flops are discussed in what follows.

A flip-flop is a binary storage device capable of storing one bit of information. In a stable state, the output of a flip-flop is either 0 or 1. A sequential circuit may use many flip-flops to store as many bits as necessary.

# Basic Flip Flop Circuit

A FF circuit can be constructed from two cross-coupled NOR gates or two cross-coupled NAND gates, and two inputs labeled *S* for set and *R* for reset. Each FF has two outputs, *Q* and *Q'*. This type of FF is also called as SR latch.



$S$	$R$	$Q$	$Q'$
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)

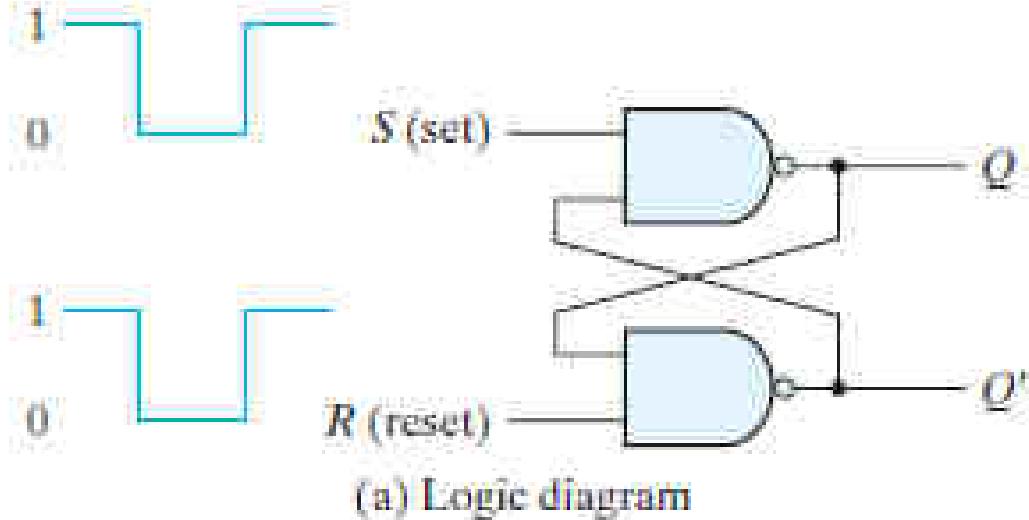
(b) Function table

### Basic FF circuit with NOR gates

- The output of a NOR gate is 0 if any input is 1, and that the output is 1 only when all the inputs are 0.

- It has two useful states. When output  $Q = 1$  and  $Q' = 0$ , it 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. The binary state of the FF is taken to be the value of the normal output.
- 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 remain at 0 unless the state has to be changed. The application of a momentary 1 to the S input causes the FF 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.
- When both inputs S and R are equal to 0, the FF 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, 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

<i>S</i>	<i>R</i>	<i>Q</i>	<i>Q'</i>
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)

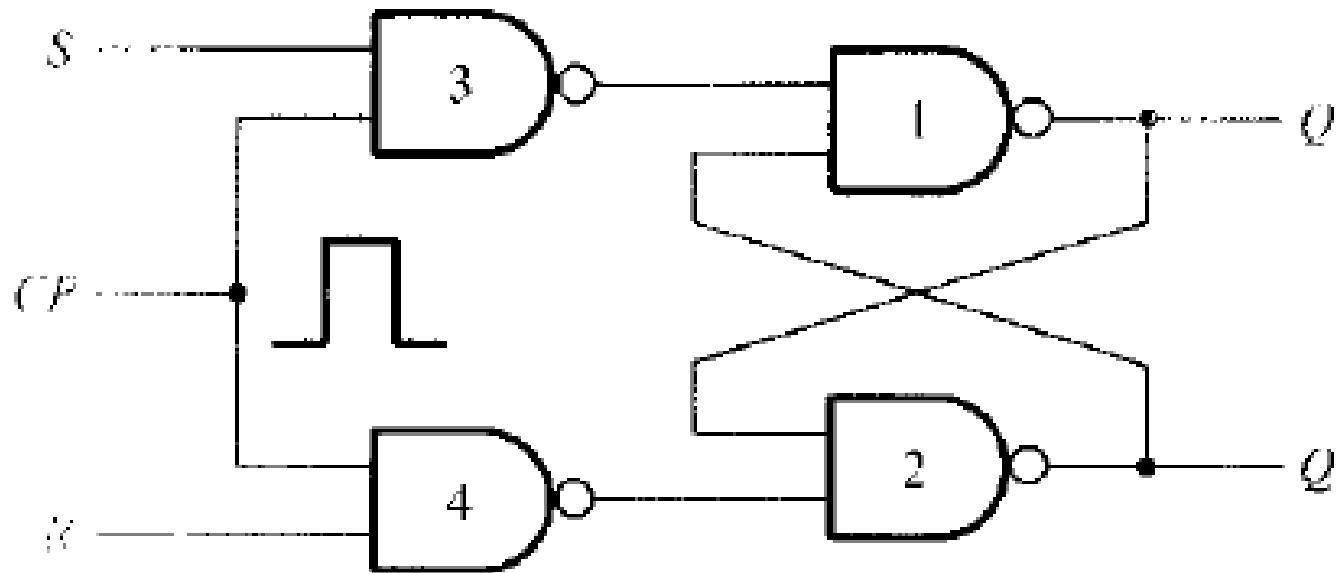
(b) Function table

### Basic FF circuit with NAND gates

- The output of a NAND gate is 1 if any input is 0, and that the output is 0 only when all the inputs are 1.

- It operates with both inputs normally at 1, unless the state has to be changed. The application of 0 to the  $S$  input causes output  $Q$  to go to 1, putting the FF in the set state.
- When the  $S$  input goes back to 1, the circuit remains in the set state. After both inputs go back to 1, we are allowed to change the state of the FF 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 FF is both inputs being equal to 0 at the same time, an input combination that should be avoided.

# SR FF with NAND gates and Clock

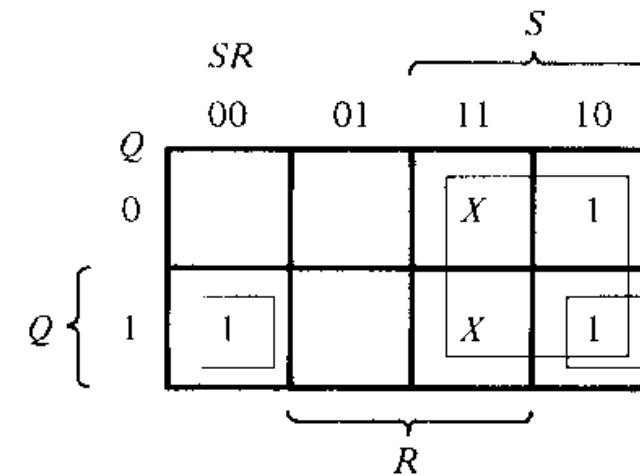


(a) Logic diagram

- FFs with clock signal are called clocked FFs or simply FFs

$Q$	$S$	$R$	$Q_{t+1}$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	Indeterminate
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	Indeterminate

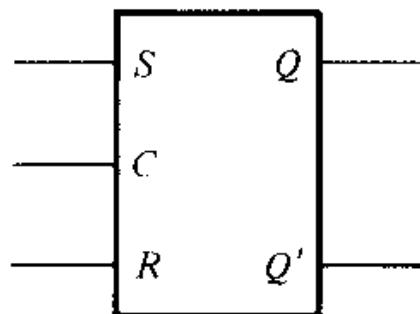
b) Characteristic Table



$$Q(t + 1) = S + R'Q$$

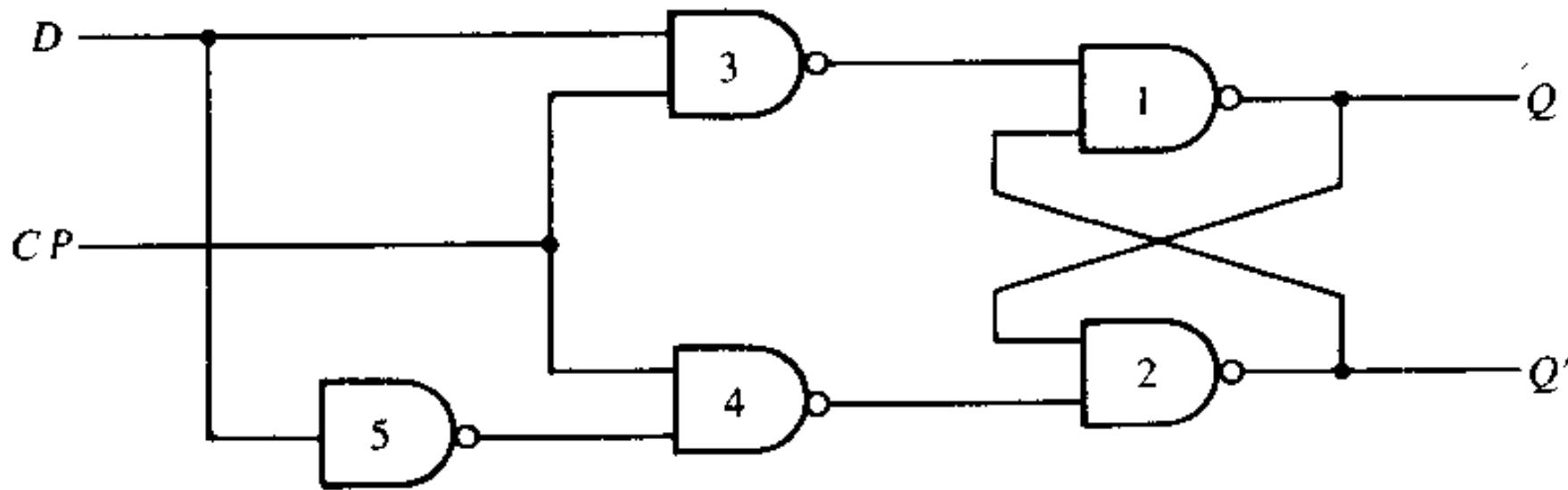
$$SR = 0$$

(c) Characteristic equation



c) Graphical Symbol

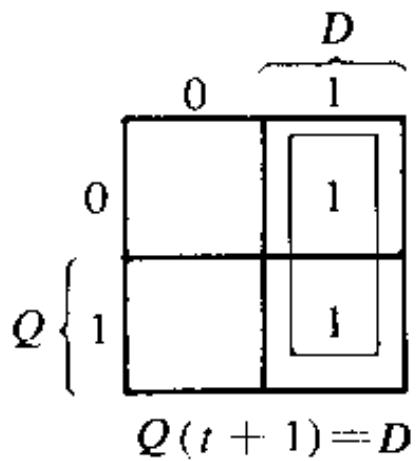
# D Flip Flop



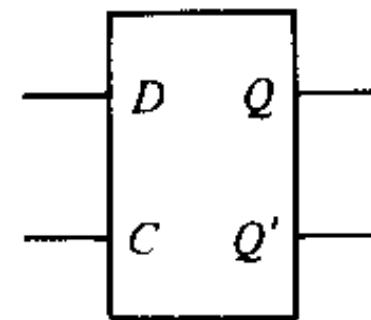
(a) Logic diagram

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

(b) Characteristic table



(c) Characteristic equation

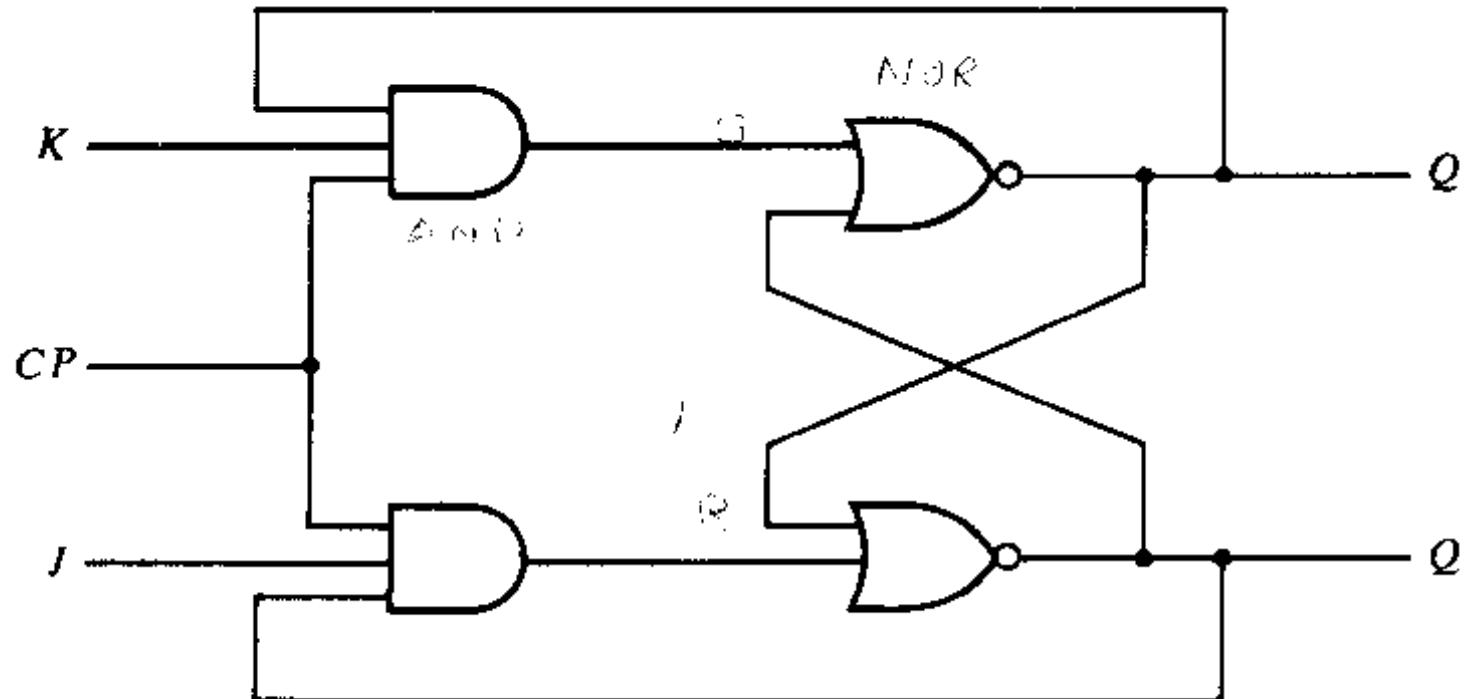


(d) Graphic symbol

**FIGURE 6-5**

*D* flip-flop

# JK Flip flop



(a) Logic diagram

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

(b) Characteristic table

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

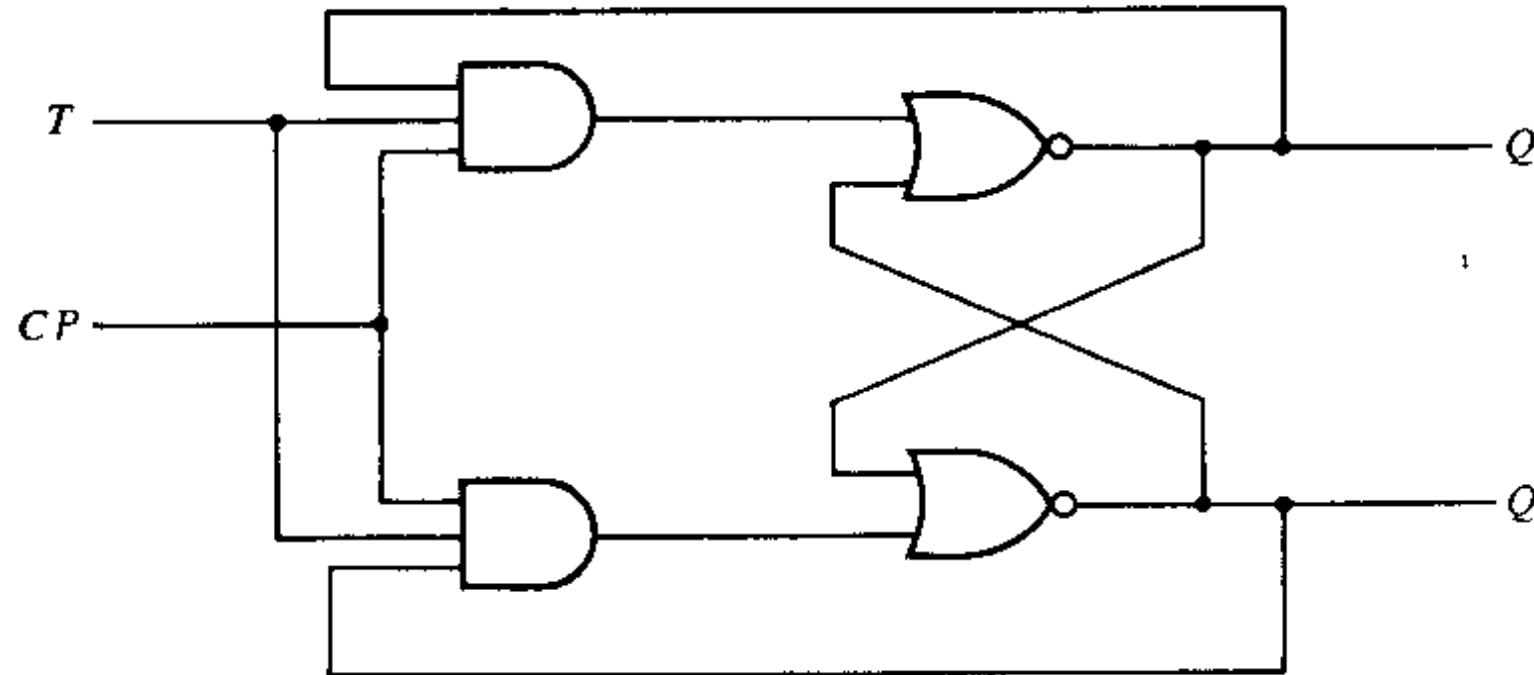
(b) Characteristic table

$Q$	$JK$	00	01	$\overbrace{11}^J$	$\overbrace{10}^K$
0				1	1
1		1			1

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

(c) Characteristic equation

# T Flip flop



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

(b) Characteristic table

T FF Logic Diagram

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

(b) Characteristic table

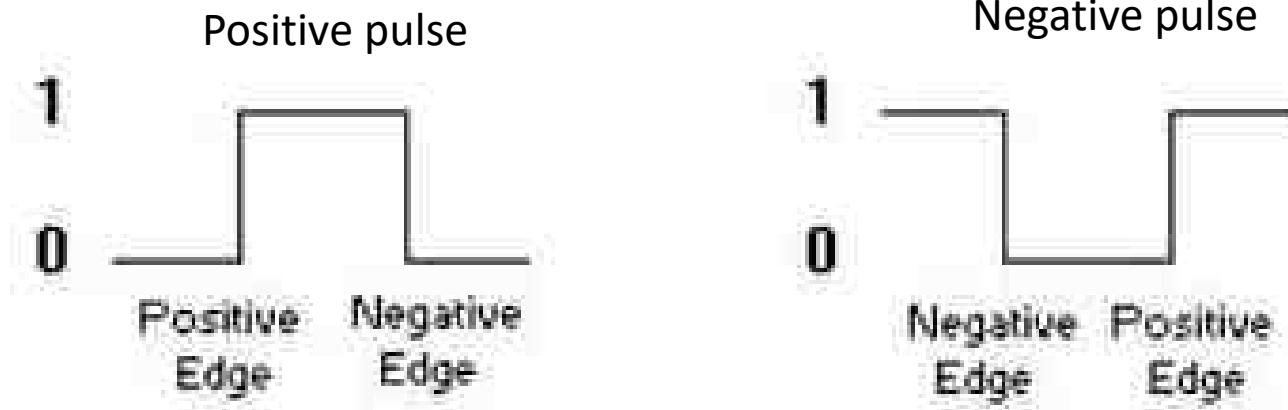
$$Q \left\{ \begin{array}{cc|c} & 0 & T \\ 0 & & 1 \\ \hline 1 & 1 & \end{array} \right.$$

$$Q(t+1) = TQ' + T'Q$$

(c) Characteristic equation

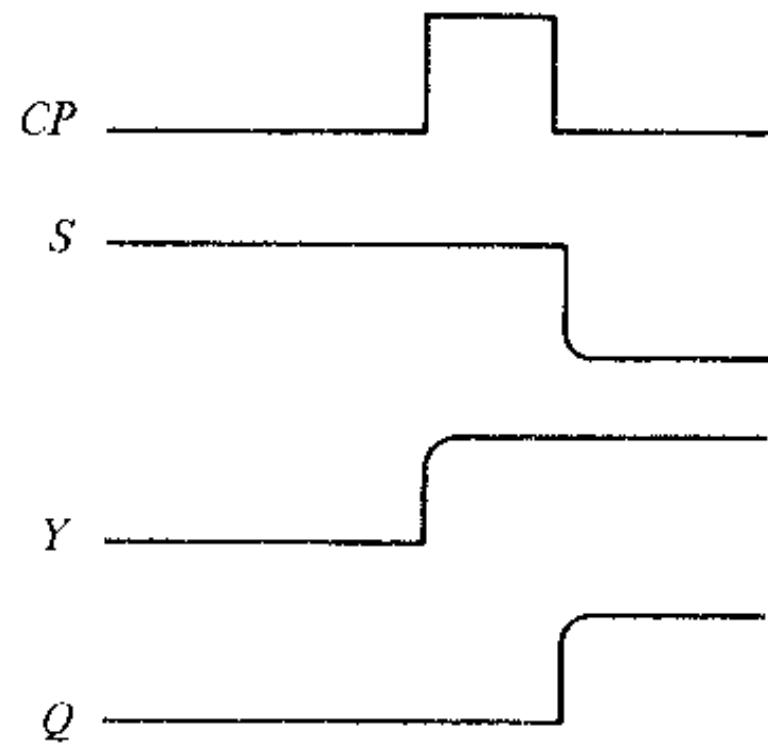
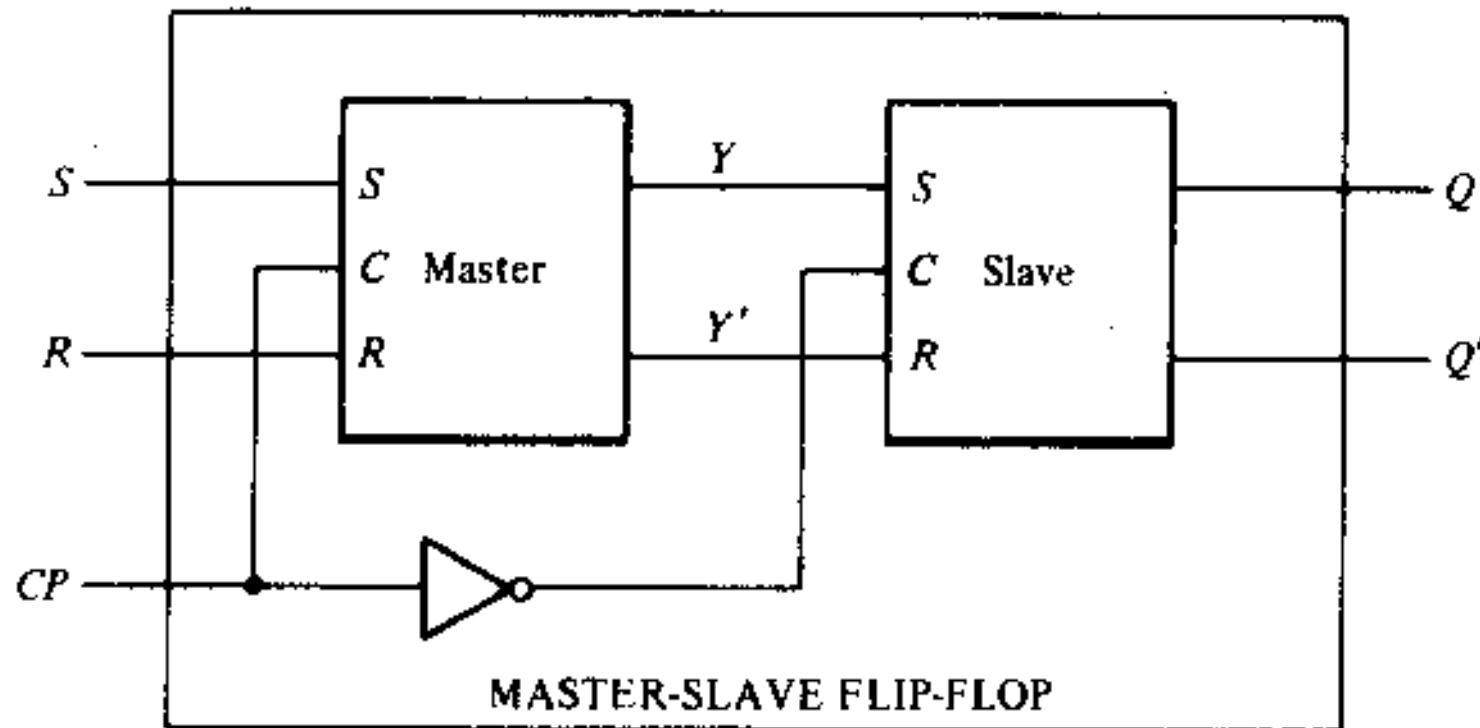
# Triggering of flip flops

- The state of a flipflop is switched by a momentary change in the input signal. This momentary change is called a trigger and the transition it causes is said to trigger the flipflop.
- Clocked FFs are triggered by clock pulses
- **Clock is a signal** that oscillates between a high and a low state

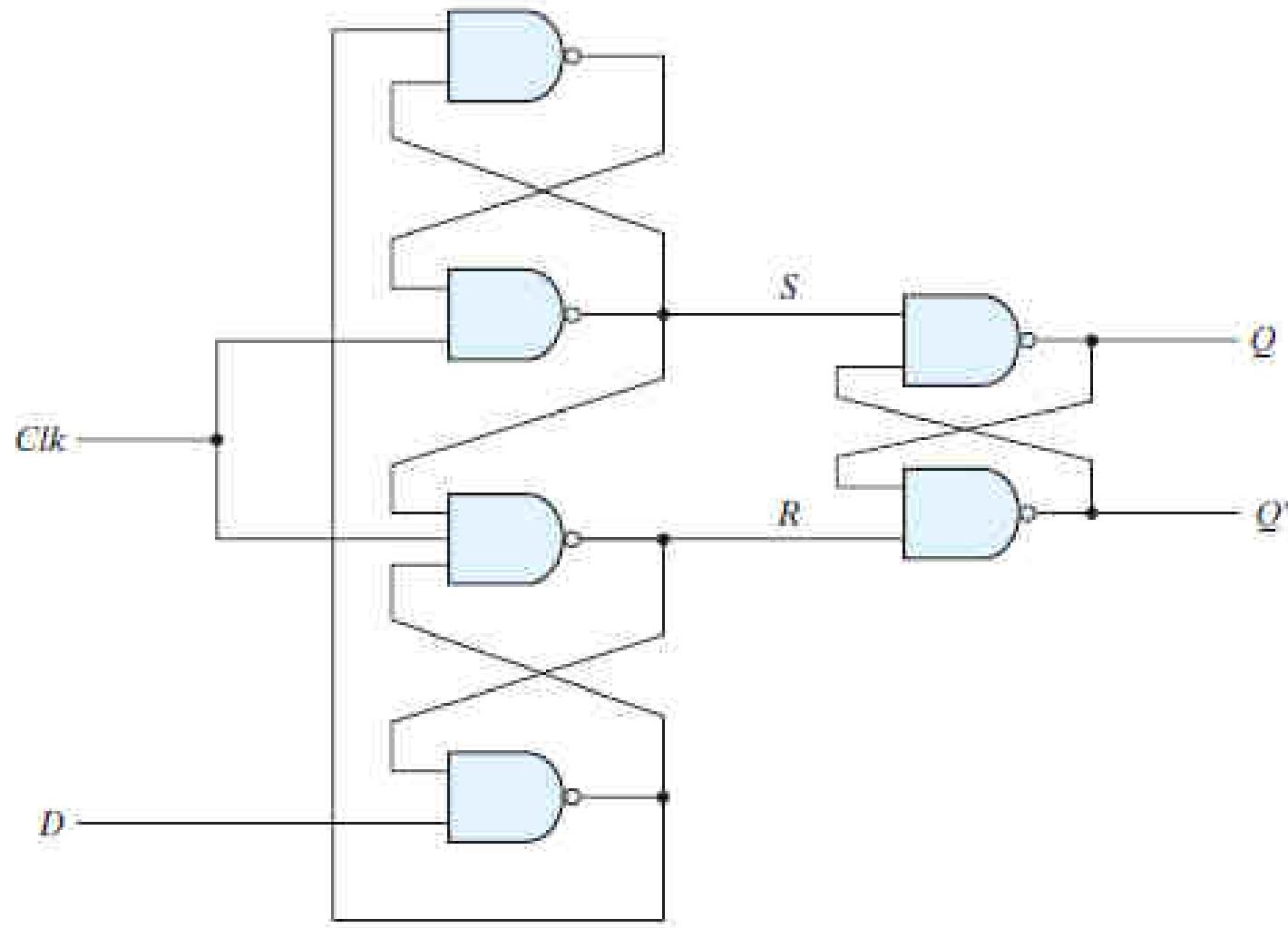


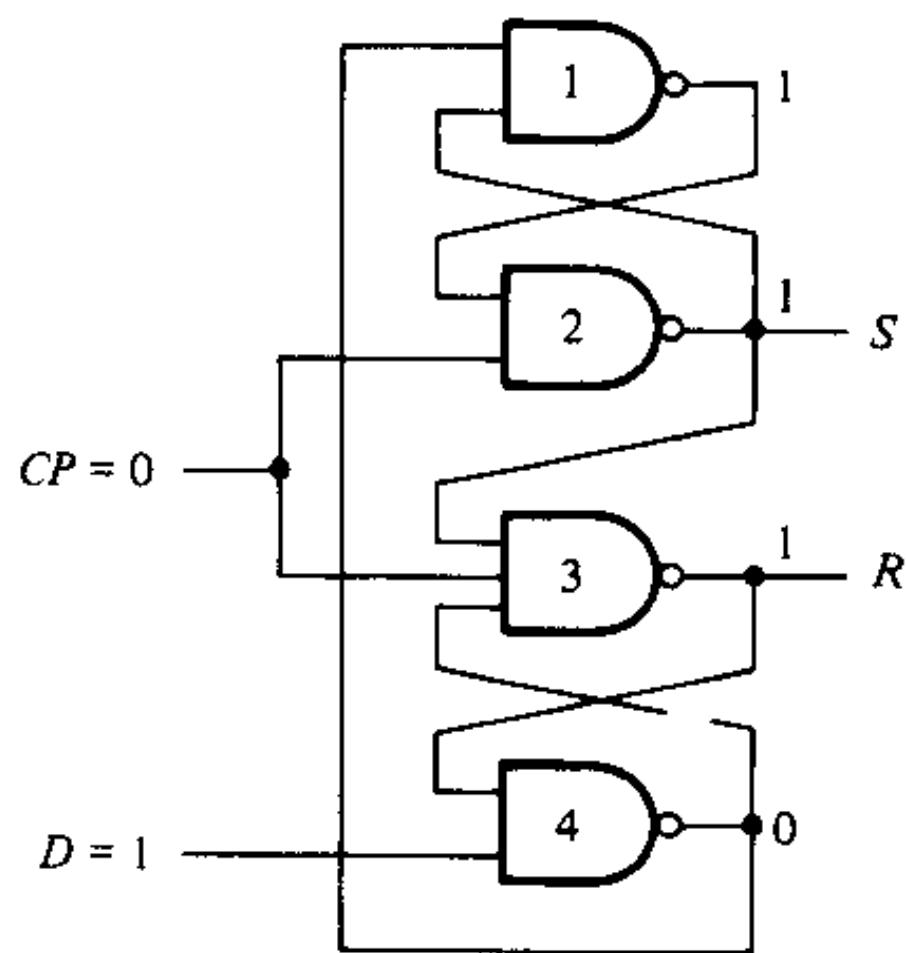
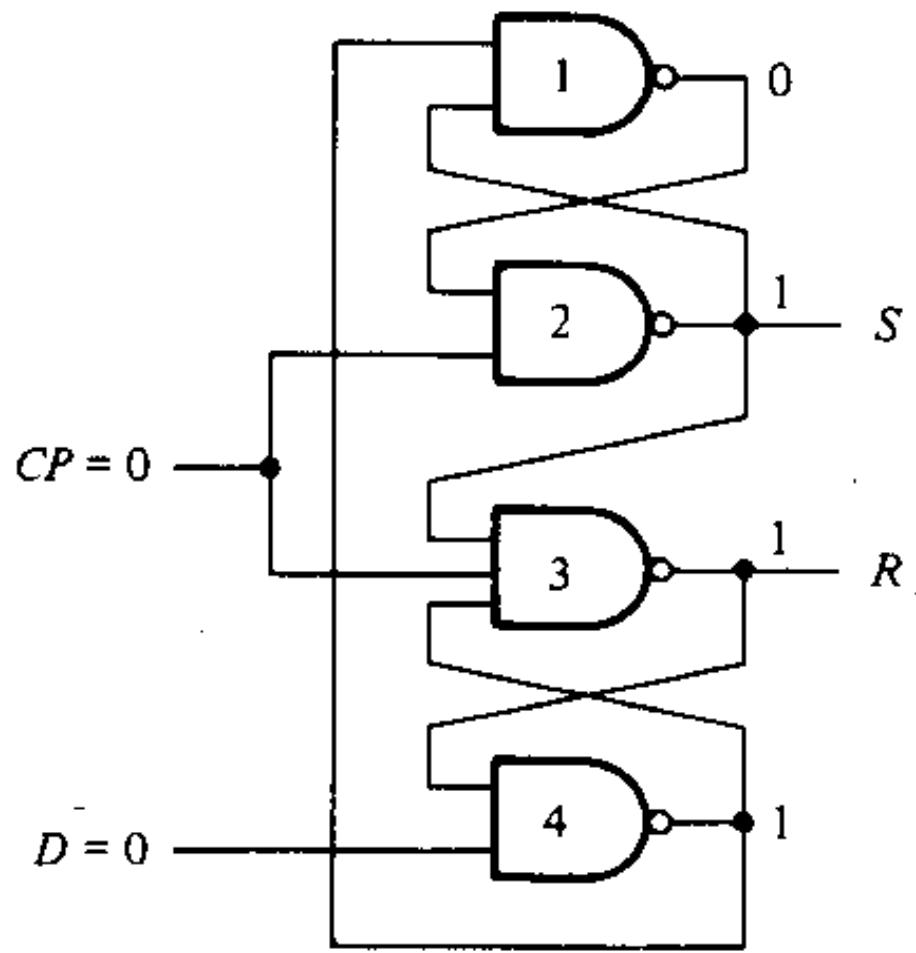
- A positive clock source remains at 0 during the interval between pulses and goes to 1 during the occurrence of a pulse
- The pulse transition from 0 to 1 is called positive edge and that from 1 to 0 is called negative edge.

# Master Slave FF

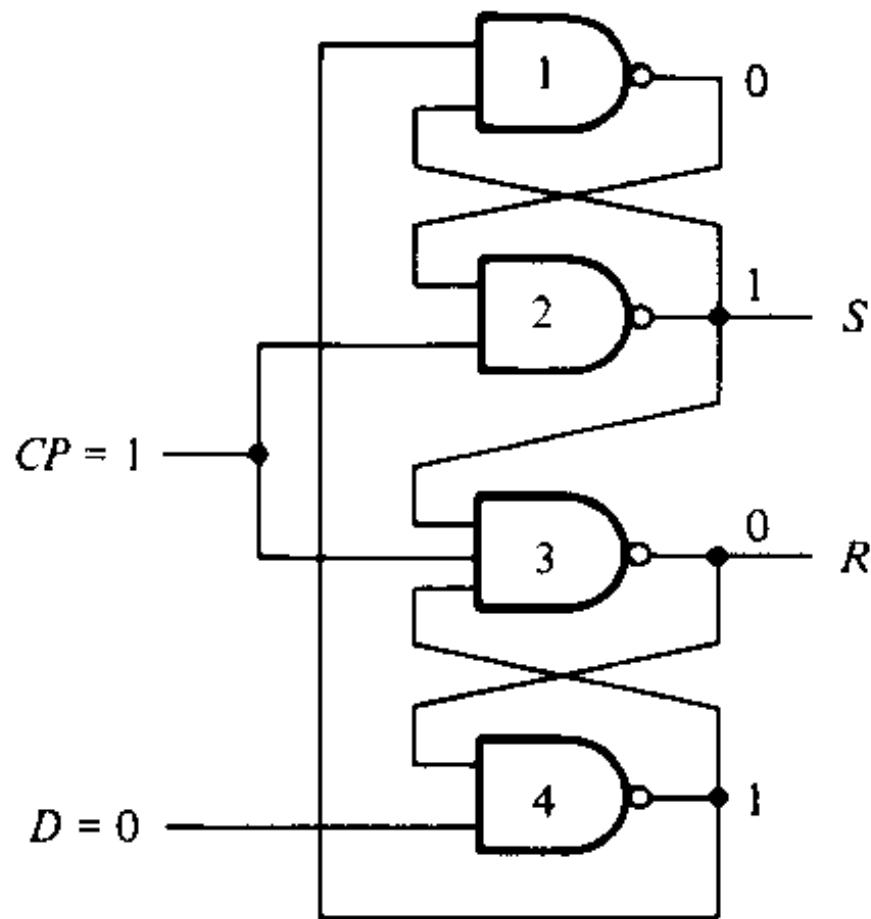


# Edge Triggered FF

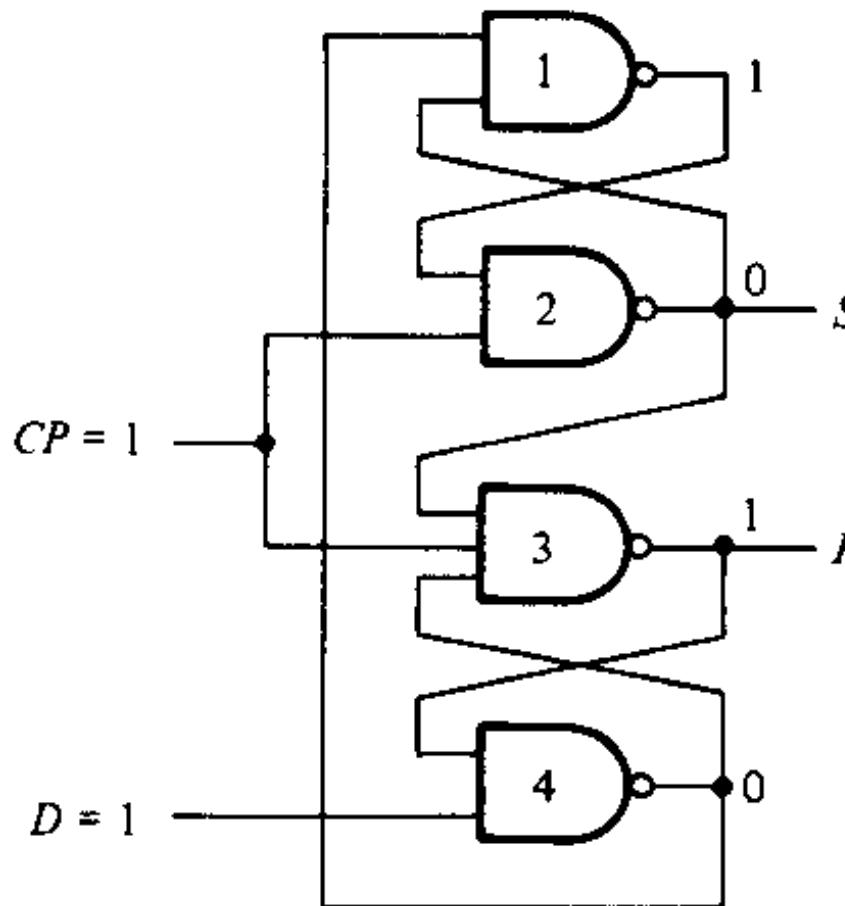




(a) With  $CP = 0$



(b) With  $CP = 1$

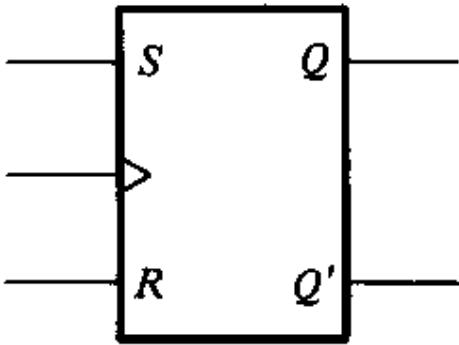


$S$	$R$	$Q$	$Q'$
1	0	0	1
1	1	0	1 (after $S = 1, R = 0$ )
0	1	1	0
1	1	1	0 (after $S = 0, R = 1$ )
0	0	1	1 (forbidden)

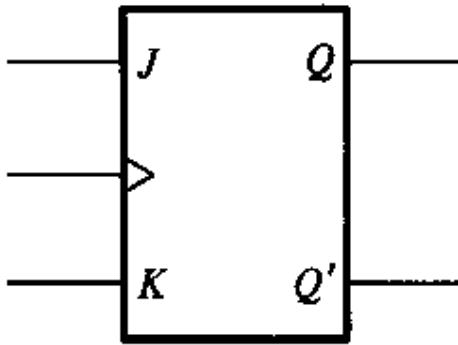
(b) Function table

- When the input clock in the positive-edge-triggered flip-flop makes a positive transition, the value of  $D$  is transferred to  $Q$ . A negative transition of the clock (i.e., from 1 to 0) does not affect the output, nor is the output affected by changes in  $D$  when  $Clock$  is in the steady logic-1 level or the logic-0 level. Hence, this type of flip-flop responds to the transition from 0 to 1 and nothing else.
- There is a minimum time called the *setup time* during which the  $D$  input must be maintained at a constant value prior to the occurrence of the clock transition. It is equal to the propagation delay through the gates 4 and 1 since a change in  $D$  causes a change in the outputs of these two gates
- Similarly, there is a minimum time called the *hold time* during which the  $D$  input must not change after the application of the positive transition of the clock. When  $D=0$ , it is the propagation delay of gate 3, since it must be ensured that  $R$  becomes 0 in order to maintain the output of gate 4 at 1, regardless the value of  $D$ . When  $D=1$ , it is the propagation delay of gate 2, since it must be ensured that  $S$  becomes 0 in order to maintain the output of gate 1 at 1, regardless the value of  $D$
- The propagation delay time of the flip-flop is defined as the interval between the trigger edge and the stabilization of the output to a new state

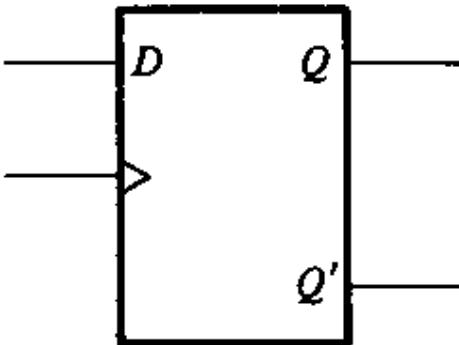
# Graphic Symbols



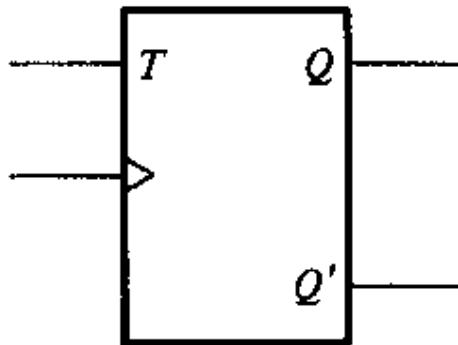
(a) *RS*



(b) *JK*

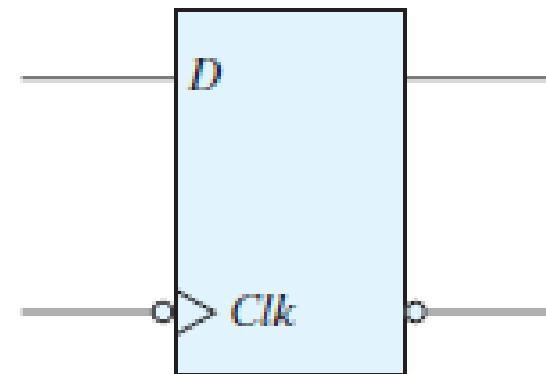


(c) *D*



(d) *T*

Positive edge triggered FFs



Negative edge triggered D FF

# Excitation Tables

$Q(t)$	$Q(t = 1)$	$J$	$K$
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

(a) JK Flip-Flop

$J$	$K$	$Q_{t+1}$
0	0	$Q_t$
0	1	0
1	0	1
1	1	$Q_t'$

$Q(t)$	$Q(t = 1)$	$T$
0	0	0
0	1	1
1	0	1
1	1	0

(b) T Flip-Flop

$T$	$Q_{t+1}$
0	$Q_t$
1	$Q_t'$

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

$D$	$Q_{t+1}$
0	0
1	1

(c)  $D$

$Q(t)$	$Q(t + 1)$	$S$	$R$
0	0	0	$X$
0	1	1	0
1	0	0	1
1	1	$X$	0

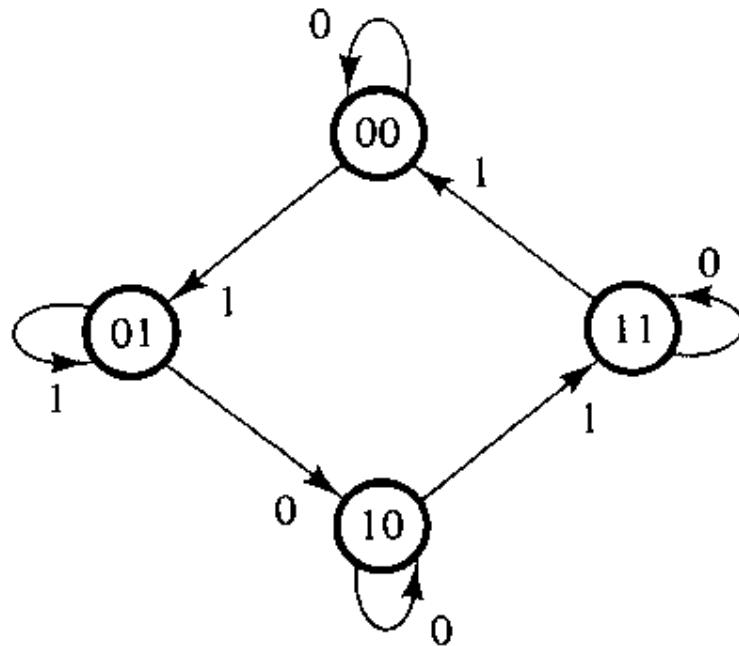
$S$	$R$	$Q_{t+1}$
0	0	$Q_t$
0	1	0
1	0	1
1	1	?

(d) SR FF

## **DESIGN PROCEDURE**

1. The problem may be given by the word description or state diagram or state table
2. From the given information obtain the state table
3. Reduce the number of states if necessary.
4. Assign binary values to the states.
5. Determine the number of FFs needed and assign a letter symbol to each
6. Choose the type of flip-flops to be used.
7. Derive the circuit excitation and output tables
8. Derive the simplified flip-flop input equations and output equations.
9. Draw the logic diagram.

Example 1: Design the synchronous sequential circuit for the following state diagram using JK FF



Solution:

**State Table**

Present State		Next State			
A	B	x = 0	x = 1	A	B
0	0	0	0	0	1
0	1	1	0	0	1
1	0	1	0	1	1
1	1	1	1	0	0

## **Excitation Table**

Inputs of Combinational Circuit			Outputs of Combinational Circuit					
Present State		Input	Next State		Flip-Flop Inputs			
A	B	x	A	B	JA	KA	JB	KB
0	0	0	0	0	0	X	0	X
0	0	1	0	1	0	X	1	X
0	1	0	1	0	1	X	X	1
0	1	1	0	1	0	X	X	0
1	0	0	1	0	X	0	0	X
1	0	1	1	1	X	0	1	X
1	1	0	1	1	X	0	X	0
1	1	1	0	0	X	1	X	1

		$B_x$	00	01	11	$B$
		A	0			10
A	1	X	X	X	X	

$\overbrace{\hspace{1cm}}$   
 $x$

$$JA = Bx'$$

X	X	X	X
		1	

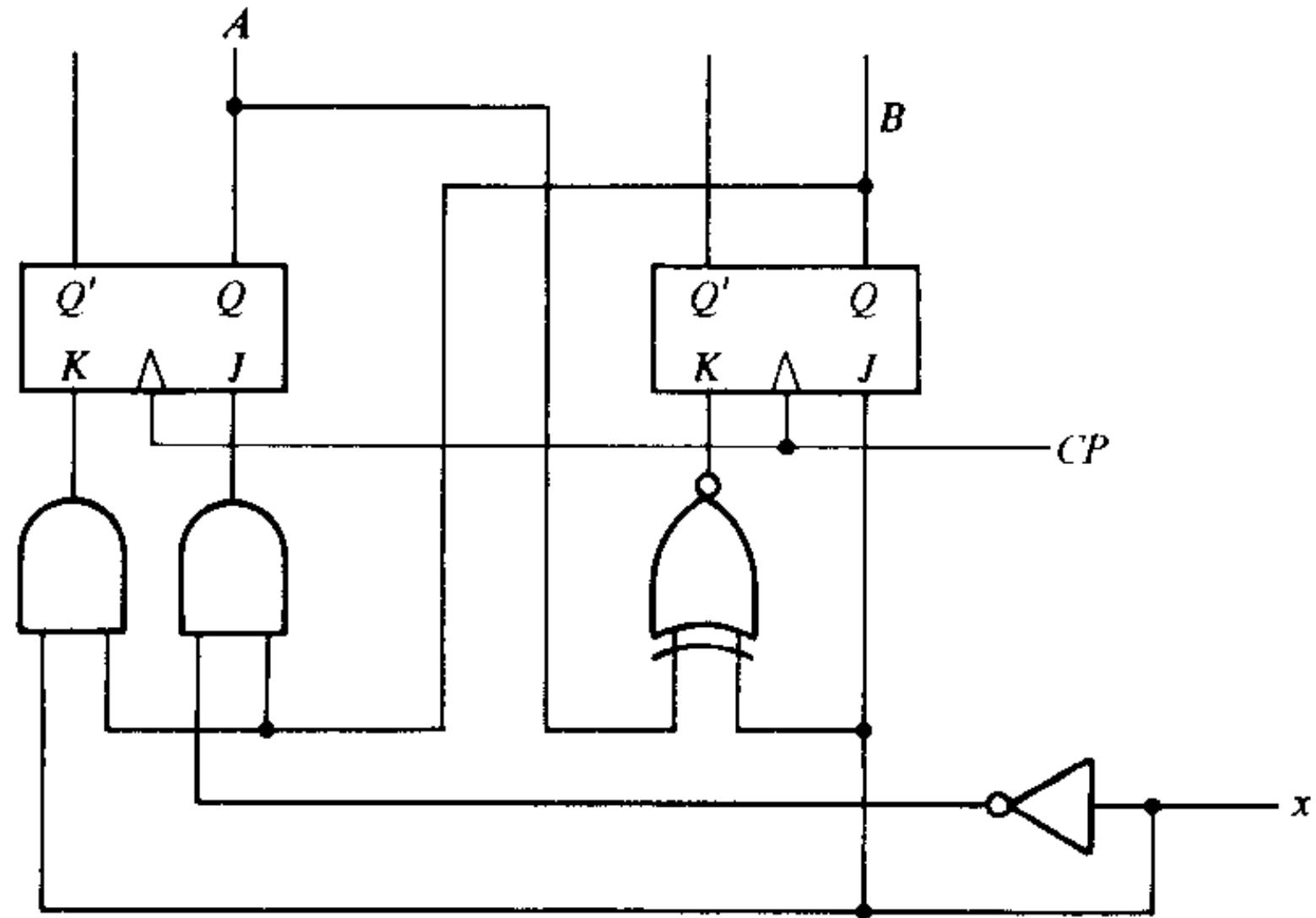
$$KA = Bx$$

	1	X	X
	1	X	X

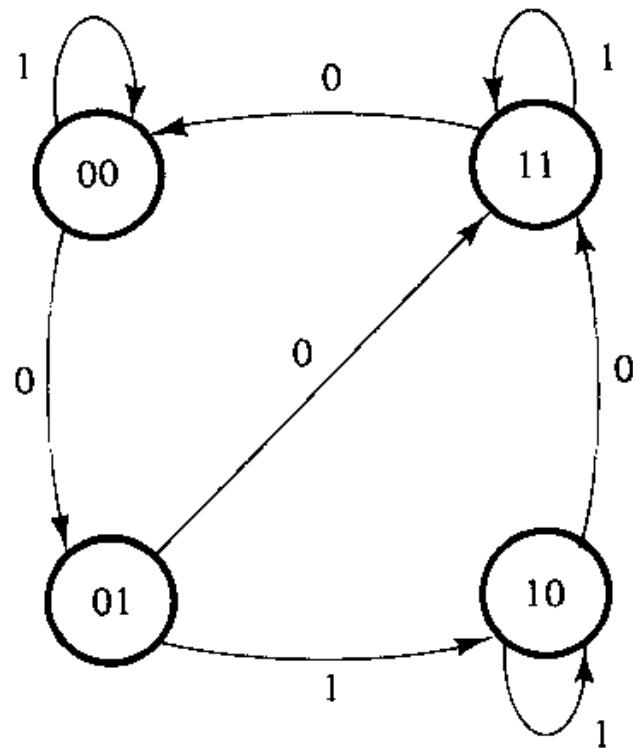
$$JB = x$$

X	X		1
X	X	1	

$$KB = (A \oplus x)'$$



Example 2: Design the synchronous sequential circuit for the following state diagram using T FF-(Exercise for the students to solve)



Solution:

$$T_a = A'B + Bx'$$

$$T_b = B'x' + Ax' + A'Bx$$

# Design with D FFs

- Design if

$$A(t+1) = DA(A, B, x) = \Sigma (2, 4, 5, 6)$$

$$B(t+1) = DB(A, B, x) = \Sigma (1, 3, 5, 6)$$

$$v(A, B, x) = \Sigma (1, 5)$$

Solution:

		$Bx$		$B$	
		00	01	11	10
$A$	0				1
	1	1	1		1
		$x$			

		1	1	
				1
		1		
				1

$$DB = A'x + B'x + ABx'$$

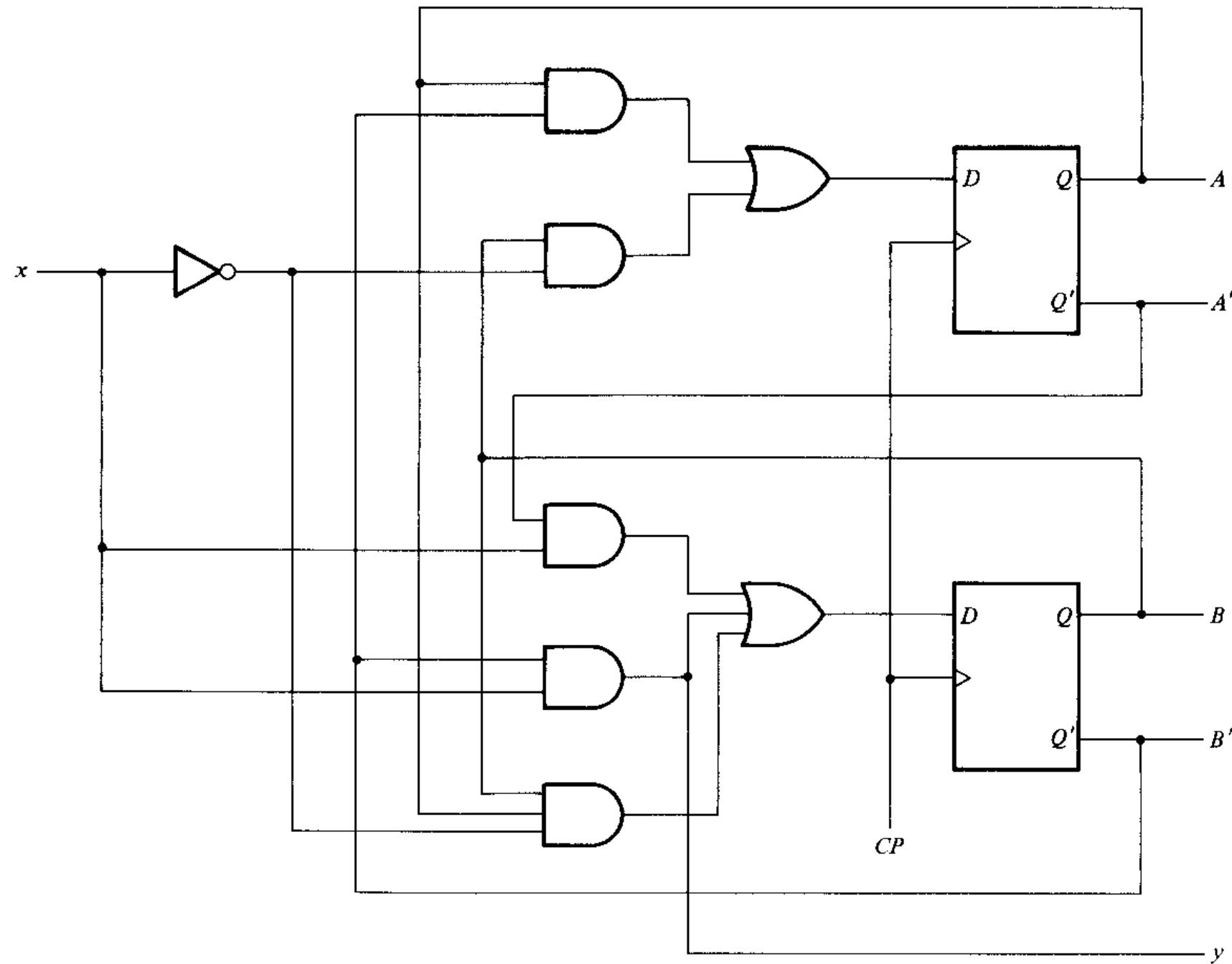
$$DA = AB' + Bx'$$

	1			
		1		

$$y = B'x$$

## **State Table for Design with $D$ Flip-Flops**

<u>Present State</u>		<u>Input</u>	<u>Next State</u>		<u>Output</u>
<i>A</i>	<i>B</i>	<i>x</i>	<i>A</i>	<i>B</i>	<i>y</i>
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	0
1	1	1	0	0	0

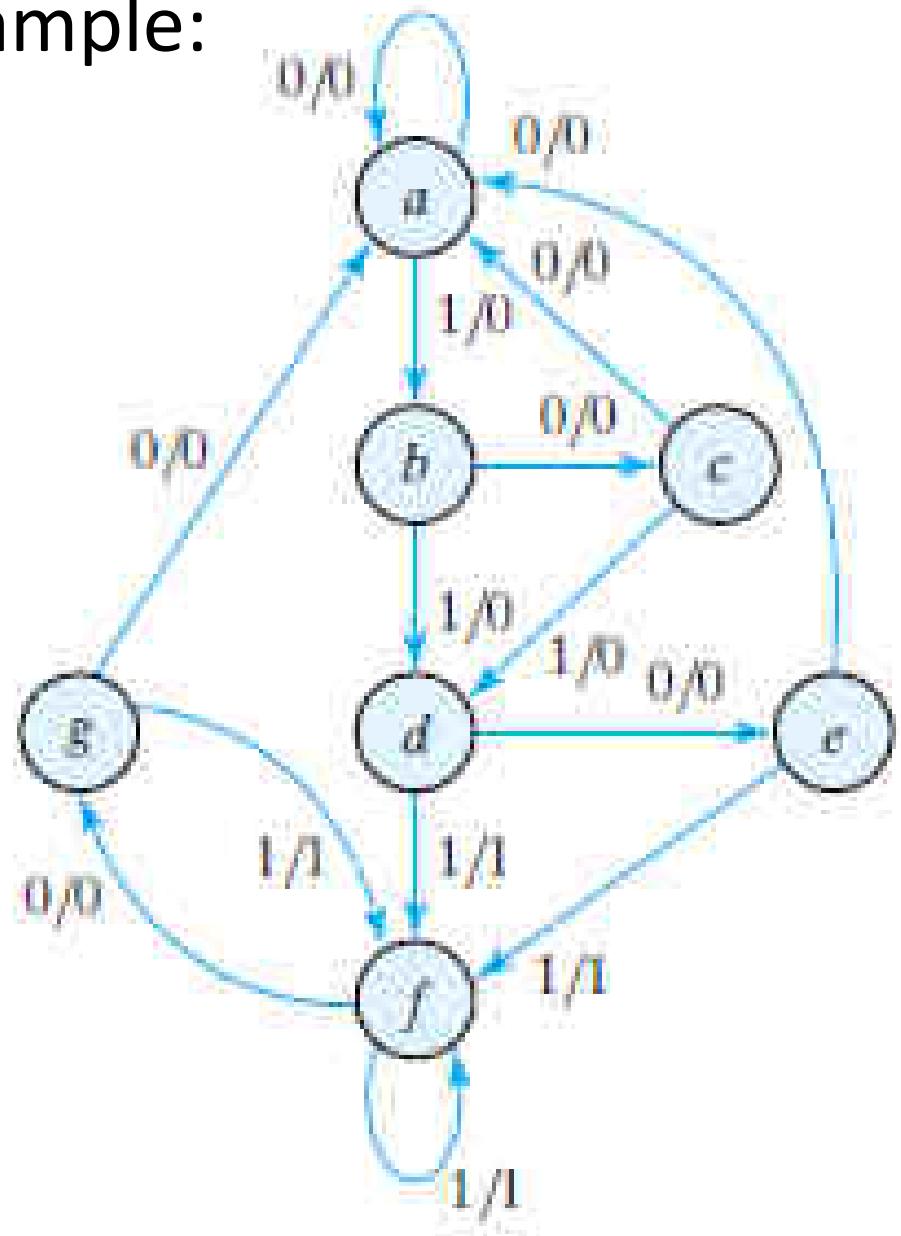


# **STATE REDUCTION AND ASSIGNMENT**

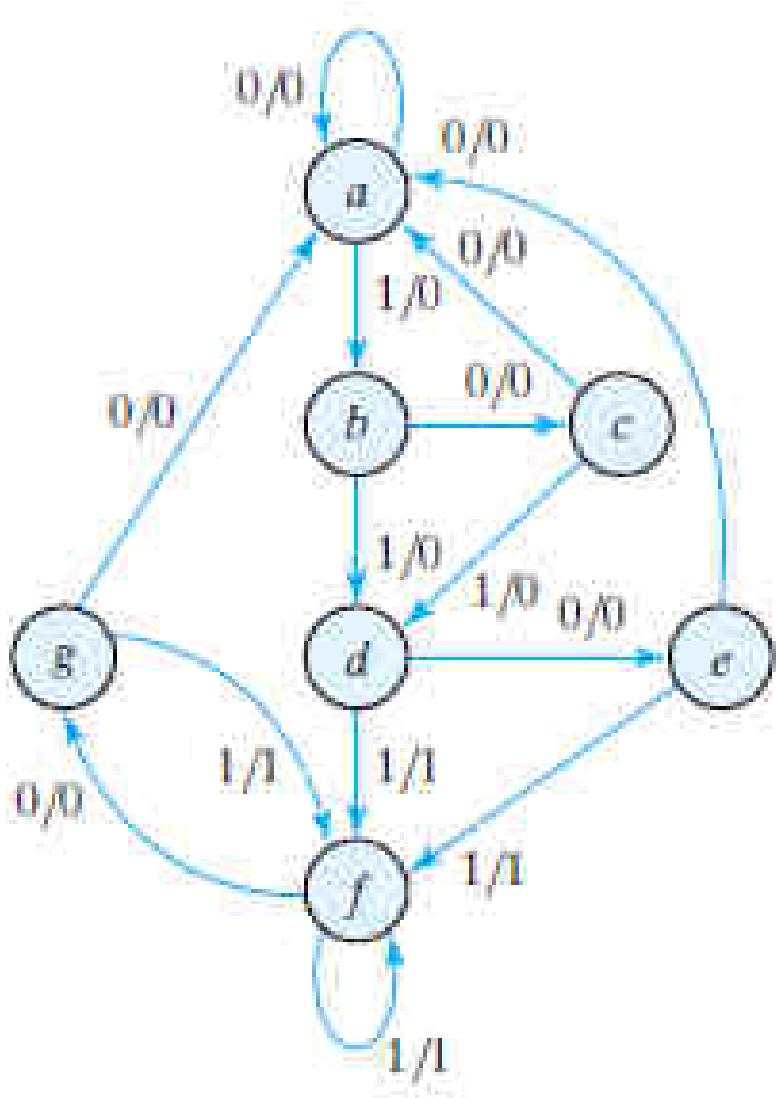
## **State Reduction**

- The reduction in the number of flip-flops in a sequential circuit is referred to as the *state-reduction* problem.
- Since  $m$  flip-flops produce  $2^m$  states, a reduction in the number of states may (or may not) result in a reduction in the number of flip-flops.
- An unpredictable effect in reducing the number of flip-flops is that sometimes the equivalent circuit (with fewer flip-flops) may require more combinational gates to realize its next state and output logic.

Example:



There are an infinite number of input sequences that may be applied to the circuit; each results in a unique output sequence. As an example, consider the input sequence 01010110100 starting from the initial state  $a$ . Each input of 0 or 1 produces an output of 0 or 1 and causes the circuit to go to the next state.



<b>State</b>	a	a	b	c	d	e	f	f	g	f	g	a
<b>Input</b>	0	1	0	1	0	1	1	0	1	0	0	
<b>Output</b>	0	0	0	0	0	1	1	0	1	0	0	

In each column, we have the present state, input value, and output value. The next state is written on top of the next column.

## *State Table*

---

<b>Present State</b>	<b>Next State</b>		<b>Output</b>	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$

## *State Table*

<b>Present State</b>	<b>Next State</b>		<b>Output</b>	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
<i>a</i>	<i>a</i>	<i>b</i>	0	0
<i>b</i>	<i>c</i>	<i>d</i>	0	0
<i>c</i>	<i>a</i>	<i>d</i>	0	0
<i>d</i>	<i>e</i>	<i>f</i>	0	1
<i>e</i>	<i>a</i>	<i>f</i>	0	1
<i>f</i>	<i>g</i>	<i>f</i>	0	1
<i>g</i>	<i>a</i>	<i>f</i>	0	1

The following algorithm for the state reduction of a completely specified state table is used: “Two states are said to be equivalent if, for each member of the set of inputs, they give exactly the same output and send the circuit either to the same state or to an equivalent state.” When two states are equivalent, one of them can be removed without altering the input–output relationships.

## 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
$f$	$g$	$f$	0	1
$g$	$a$	$f$	0	1

we look for two present states that go to the same next state and have the same output for both input combinations. States  $e$  and  $g$  are two such states: They both go to states  $a$  and  $f$  and have outputs of 0 and 1 for  $x = 0$  and  $x = 1$ , respectively. Therefore, states  $g$  and  $e$  are equivalent, and one of these states can be removed.

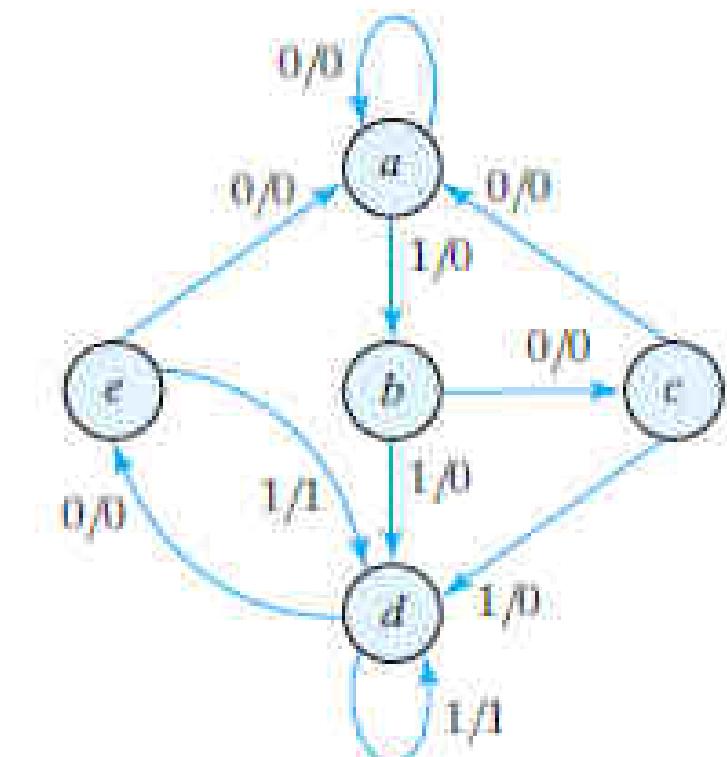
## Reducing the State Table

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
$f$	$g$	$f$	0	1
$g$	$a$	$f$	0	1

### Reduced 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
$d$	$e$	$d$	0	1
$e$	$a$	$d$	0	1

<b>State</b>	a	a	b	c	d	e	d	d	e	a
<b>Input</b>	0	1	0	1	0	1	1	0	1	0
<b>Output</b>	0	0	0	0	0	1	1	0	1	0



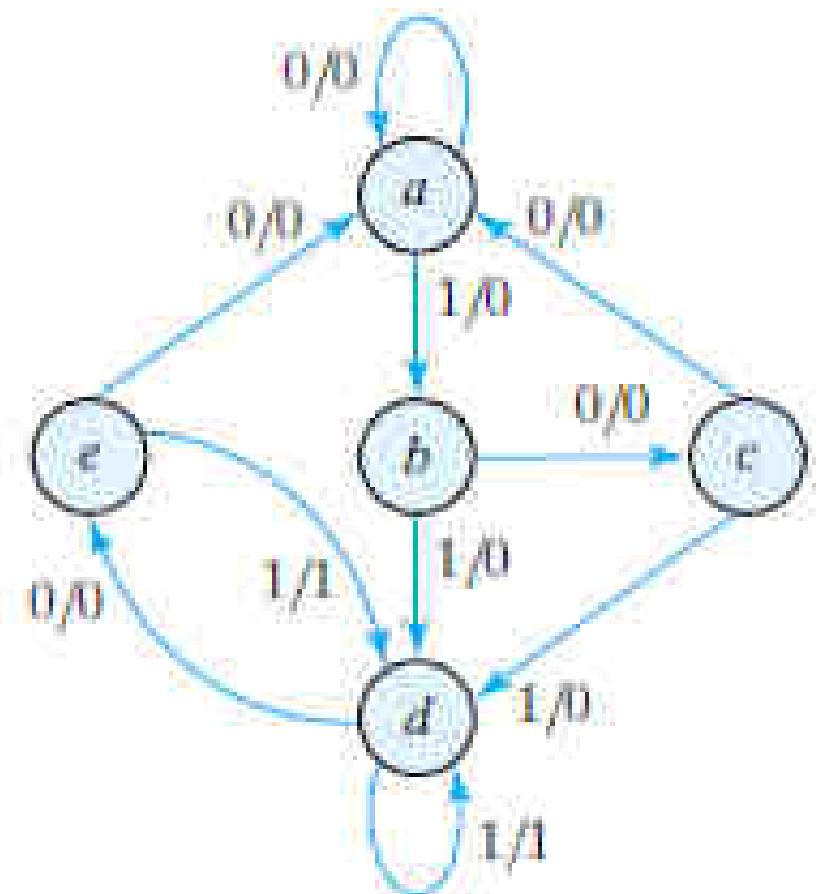
The sequential circuit of this example was reduced from seven to five states but the no. of FFs are not reduced because 5 states require 3 FFs that is same for 7 states too. In general, reducing the number of states in a state table may result in a circuit with less equipment because the unused states are taken as don't cares in the design process. However, the fact that a state table has been reduced to fewer states does not guarantee a saving in the number of flip-flops or the number of gates.

# State Assignment

In order to design a sequential circuit with physical components, it is necessary to assign unique coded binary values to the states. For a circuit with  $m$  states, the codes must contain  $n$  bits, where  $2^n \geq m$ . For example, with three bits, it is possible to assign codes to eight states, denoted by binary numbers 000 through 111. For 5 states, we are left with three unused states. Unused states are treated as don't-care conditions during the design. Since don't-care conditions usually help in obtaining a simpler circuit, it is more likely but not certain that the circuit with five states will require fewer combinational gates than the one with seven states.

The simplest way to code five states is to use the first five integers in binary counting order. Another similar assignment is the Gray code. Here, only one bit in the code group changes when going from one number to the next. This code makes it easier for the Boolean functions to be placed in the map for simplification. There are many possibilities.

State	Assignment 1, Binary	Assignment 2, Gray Code
<i>a</i>	000	000
<i>b</i>	001	001
<i>c</i>	010	011
<i>d</i>	011	010
<i>e</i>	100	110



*Reduced State Table with Binary Assignment 1*

Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
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

## Design with Unused States

Design the synchronous sequential circuit for the following state table:

**State Table with Unused States**

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

Present State	Input	Next State	Flip-Flop Inputs						Output	
			A	B	C	S <sub>A</sub>	R <sub>A</sub>	S <sub>B</sub>	R <sub>B</sub>	
0 0 1	0	0 0 1								0
0 0 1	1	0 1 0								0
0 1 0	0	0 1 1								0
0 1 0	1	1 0 0								0
0 1 1	0	0 0 1								0
0 1 1	1	1 0 0								0
1 0 0	0	1 0 1								0
1 0 0	1	1 0 0								1
1 0 1	0	0 0 1								0
1 0 1	1	1 0 0								1

O(t)	O(t + 1)	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

Present State			Input <u>x</u>	Next State			Flip-Flop Inputs						Output <u>y</u>	
<u>A</u>	<u>B</u>	<u>C</u>		<u>A</u>	<u>B</u>	<u>C</u>	<u>SA</u>	<u>RA</u>	<u>SB</u>	<u>RB</u>	<u>SC</u>	<u>RC</u>		
0	0	1	0	0	0	1	0	X	0	X	X	0		0
0	0	1	1	0	1	0	0	X	1	0	0	1		0
0	1	0	0	0	1	1	0	X	X	0	1	0		0
0	1	0	1	1	0	0	1	0	0	1	0	X		0
0	1	1	0	0	0	1	0	X	0	1	X	0		0
0	1	1	1	1	0	0	1	0	0	1	0	1		0
1	0	0	0	1	0	1	X	0	0	X	1	0		0
1	0	0	1	1	0	0	X	0	0	X	0	X		1
1	0	1	0	0	0	1	0	1	0	X	X	0		0
1	0	1	1	1	0	0	X	0	0	X	0	1		1

$AB$	$Cx$	$C$		
	00	01	11	10
$A$	$X$	$X$		
00	$X$	$X$		
01		1	1	
11	$X$	$X$	$X$	$X$
10	$X$	$X$	$X$	

$\underbrace{\hspace{1cm}}$   
 $x$

$SA = Bx$

$X$	$X$	$X$	$X$
$X$			
$X$	$X$	$X$	$X$
			1

$$RA = Cx'$$

$X$	$X$	1	
$X$			
$X$	$X$	$X$	$X$
			1

$$SB = A'B'x$$

$X$	$X$		
$X$	$X$	$X$	$X$
		1	1

$$y = Ax$$

$X$	$X$		$X$
	1	1	1
$X$	$X$	$X$	$X$
$X$	$X$	$X$	$X$

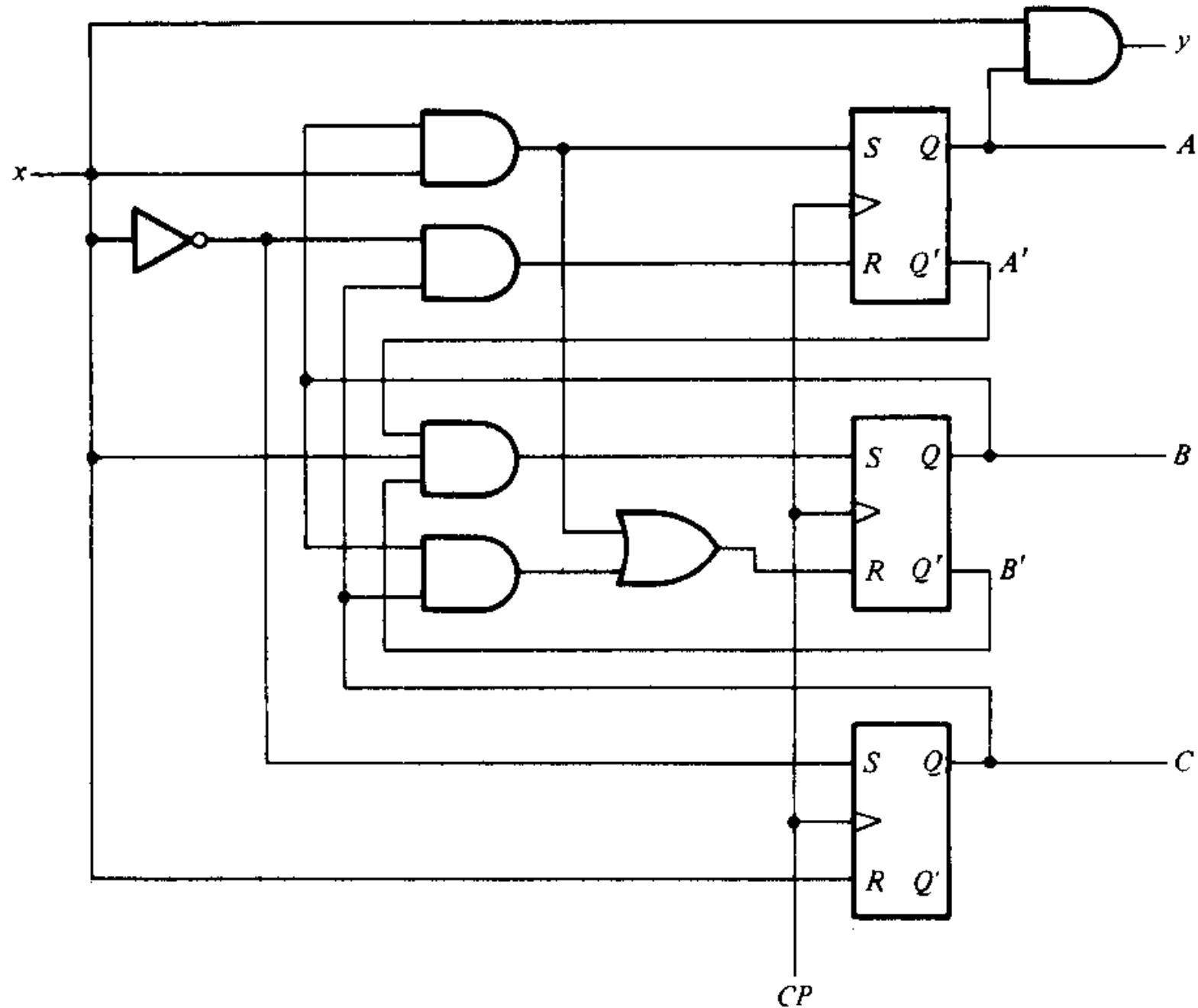
$$RB = BC + Bx$$

$X$	$X$		$X$
1			$X$
$X$	$X$	$X$	$X$
1			$X$

$$SC = x'$$

$X$	$X$	1	
	$X$	1	
$X$	$X$	$X$	$X$
	$X$	1	

$$RC = x$$



Example: Design a sequential circuit with two  $JK$  flip-flops  $A$  and  $B$  and two inputs  $E$  and  $F$ . If  $E = 0$ , the circuit remains in the same state regardless of the value of  $F$ . When  $E = 1$  and  $F = 1$ , the circuit goes through the state transitions from 00 to 01, to 10, to 11, back to 00, and repeats.

When  $E = 1$  and  $F = 0$ , the circuit goes through the state transitions from 00 to 11, to 10, to 01, back to 00, and repeats

Solution:  $JA = KA = (B'F' + BF)E$   
 $JB = KB = E$

## Home work:

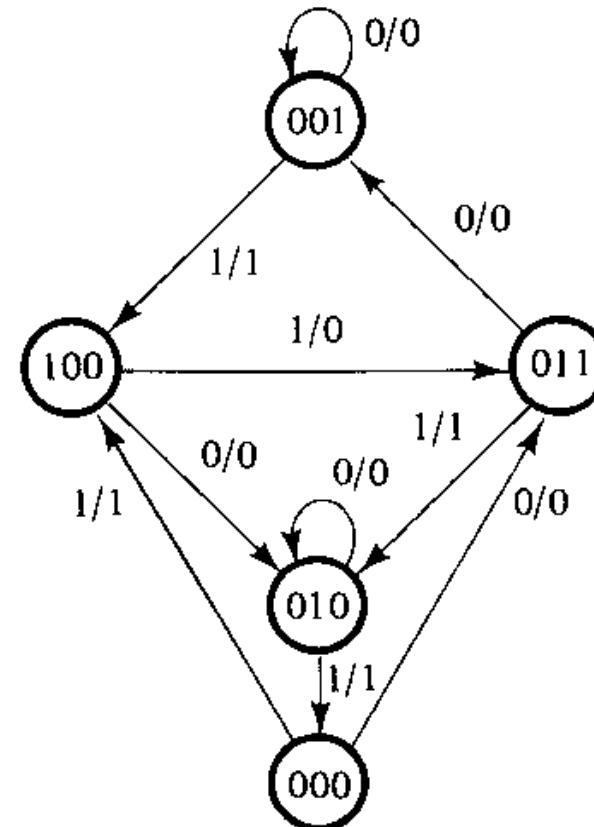
1. Reduce the number of states in the following state table and tabulate the reduced state table.

Present State	Next state		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
$a$	$f$	$b$	0	0
$b$	$d$	$c$	0	0
$c$	$f$	$e$	0	0
$d$	$g$	$a$	1	0
$e$	$d$	$c$	0	0
$f$	$f$	$b$	1	1
$g$	$g$	$h$	0	1
$h$	$g$	$a$	1	0

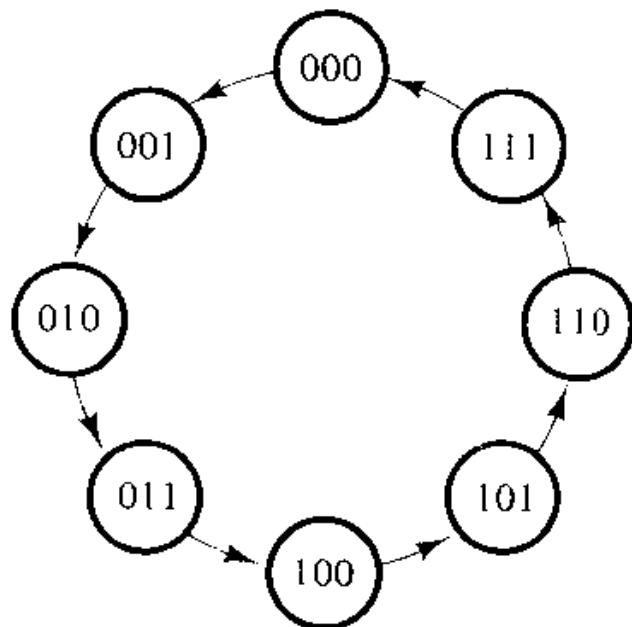
2 Design a sequential circuit with two  $D$  flip-flops,  $A$  and  $B$ , and one input,  $x$ . When  $x = 0$ , the state of the circuit remains the same. When  $x = 1$ , the circuit goes through the state transitions from 00 to 01 to 11 to 10 back to 00, and repeats.

3. Design the sequential circuit for the following state diagram

- (a) Use  $D$  flip-flops in the design.
- (b) Use  $JK$  flip-flops in the design.



# Design of synchronous counters



### **Excitation Table for 3-Bit Counter**

Present State			Next State			Flip-Flop Inputs		
$A_2$	$A_1$	$A_0$	$A_2$	$A_1$	$A_0$	$TA_2$	$TA_1$	$TA_0$
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	0	1
1	1	1	0	0	0	1	1	1

$A_2$	$A_1$	$A_0$	$TA_2$
0	0	1	1
0	1	1	1

$$TA_2 = A_1 A_0$$

$A_2$	$A_1$	$A_0$	$TA_1$
0	0	1	1
0	1	1	1

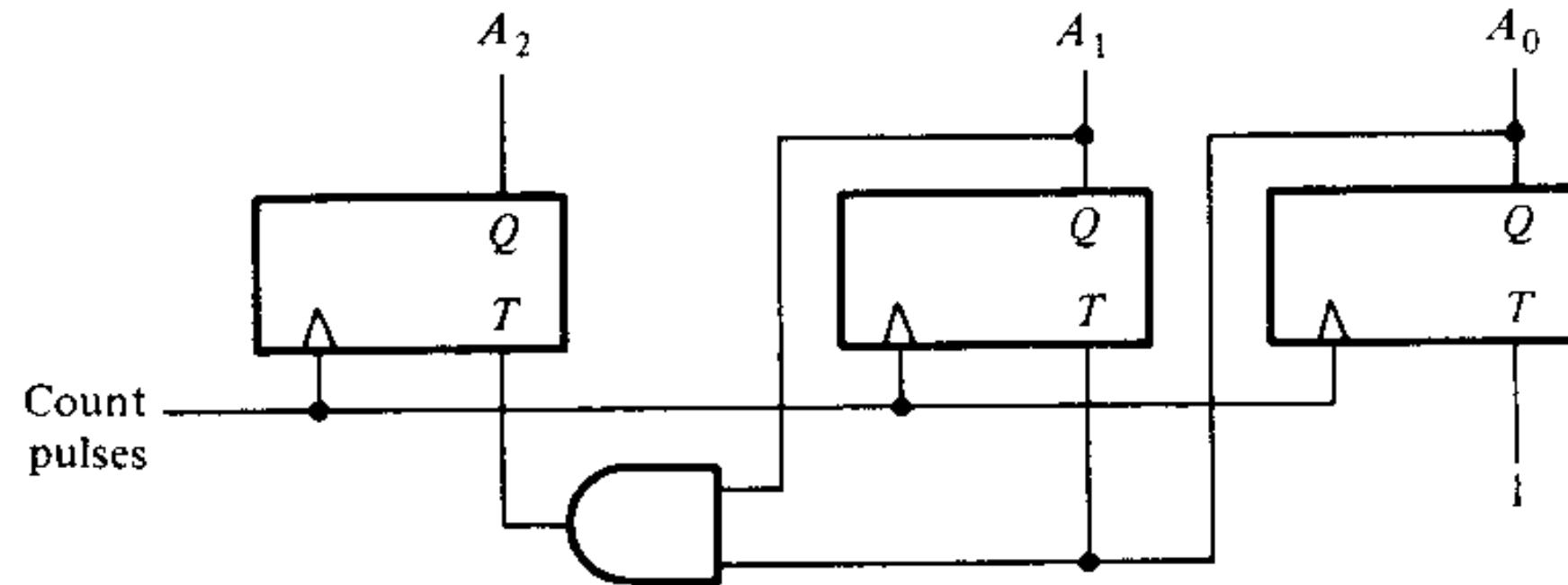
$$TA_1 = A_0$$

$A_2$	$A_1$	$A_0$	$TA_0$
0	0	1	1
0	1	1	1

$$TA_0 = 1$$

### Excitation Table for 3-Bit Counter

Present State			Next State			Flip-Flop Inputs		
$A_2$	$A_1$	$A_0$	$A_2$	$A_1$	$A_0$	$TA_2$	$TA_1$	$TA_0$
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	1	1
1	1	1	0	0	0	1	1	1



## Counter with Nonbinary Sequence

Design a three-bit counter that counts in the sequence  
0, 1, 2, 4, 5, 6, 0, .....

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

(a) JK Flip-Flop

## Excitation Table for Counter

Present State			Next State			Flip-Flop Inputs					
$A$	$B$	$C$	$A$	$B$	$C$	$JA$	$KA$	$JB$	$KB$	$JC$	$KC$

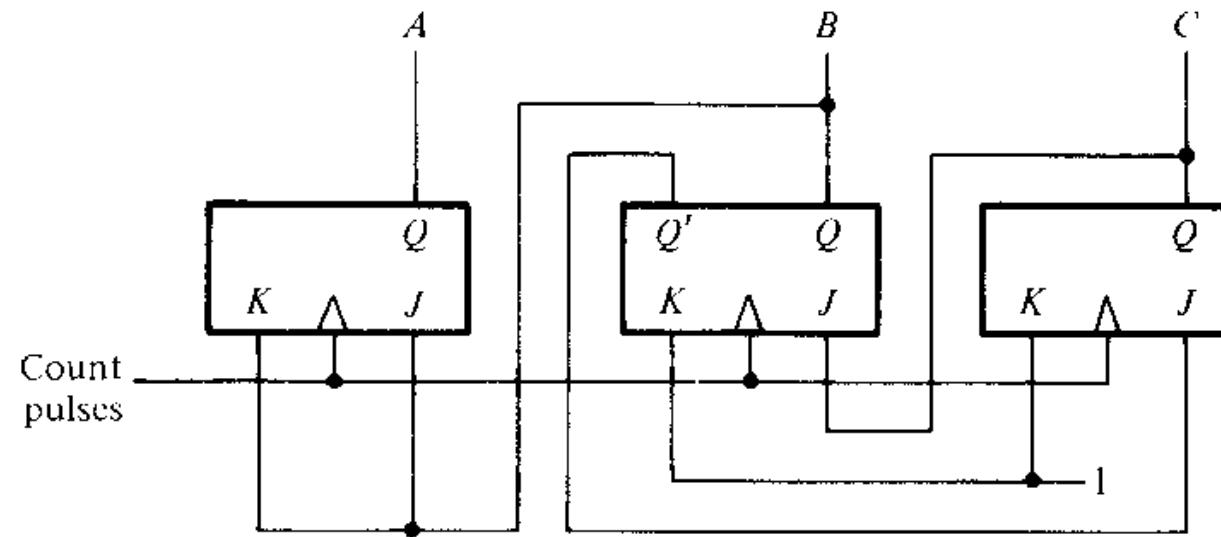
## Excitation Table for Counter

Present State			Next State			Flip-Flop Inputs					
A	B	C	A	B	C	JA	KA	JB	KB	J <sub>C</sub>	K <sub>C</sub>
0	0	0	0	0	1	0	X	0	X	1	X
0	0	1	0	1	0	0	X	1	X	X	1
0	1	0	1	0	0	1	X	X	1	0	X
1	0	0	1	0	1	X	0	0	X	1	X
1	0	1	1	1	0	X	0	1	X	X	1
1	1	0	0	0	0	X	1	X	1	0	X

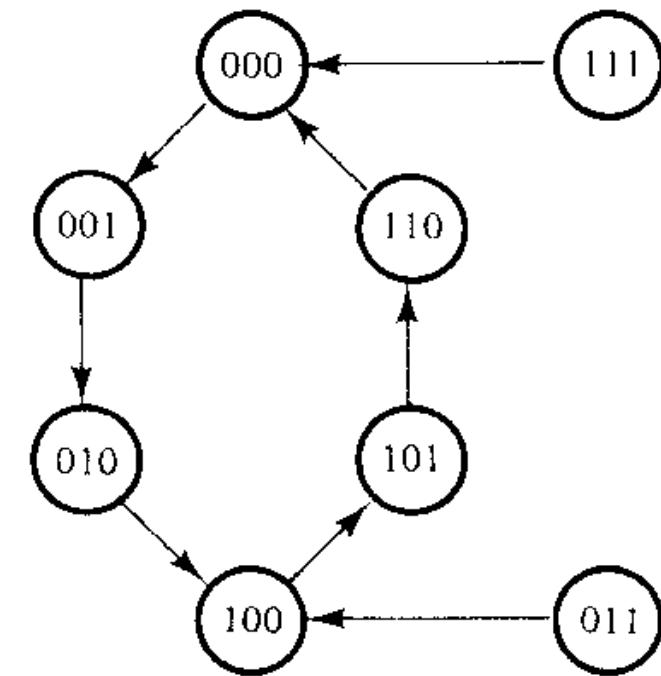
$$JA = B \quad KA = B$$

$$JB = C \quad KB = 1$$

$$J_C = B' \quad K_C = 1$$



(a) Logic diagram of counter



(b) State diagram of counter

## Problem

A  $JN$  flip-flop has two inputs,  $J$  and  $N$ . Input  $J$  behaves like the  $J$  input of a  $JK$  flip-flop and input  $N$  behaves like the complement of the  $K$  input of a  $JK$  flip-flop (that is,  $N = K'$ ).

- (a) Tabulate the characteristic table of the flip-flop (as in Table 6-3).
- (b) Tabulate the excitation table of the flip-flop (as in Table 6-10).
- (c) Show that by connecting the two inputs together, one obtains a  $D$  Flip-flop.

Problem:

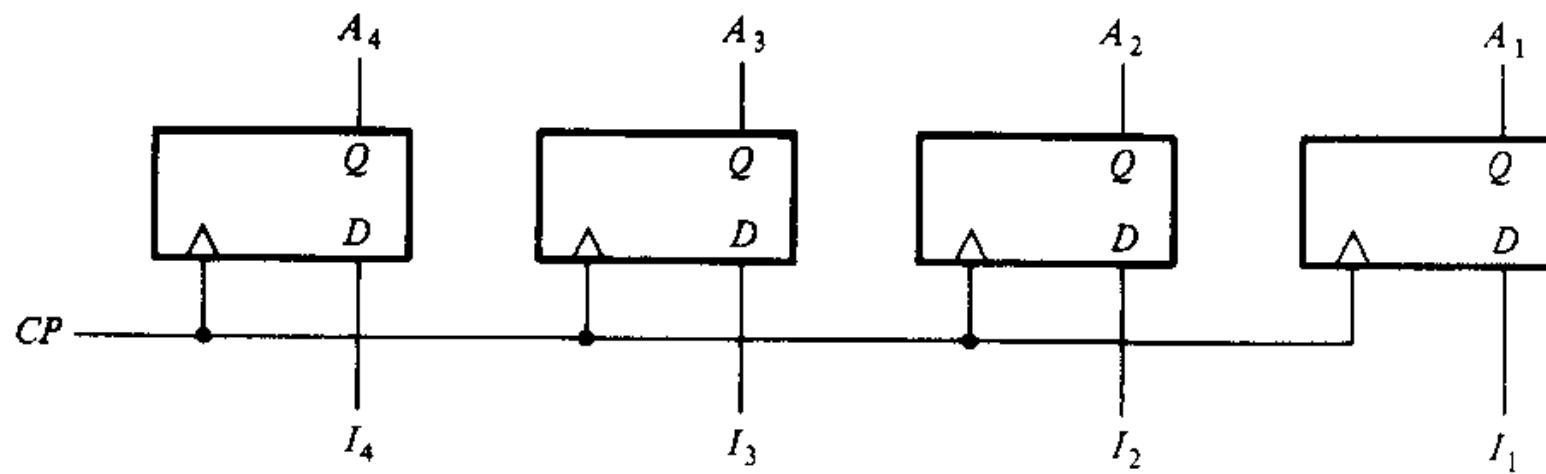
Design with the following repeated binary sequence 0, 1, 3, 7, 6, 4.  
Use T FF. Check if it is self correcting

## **Home work**

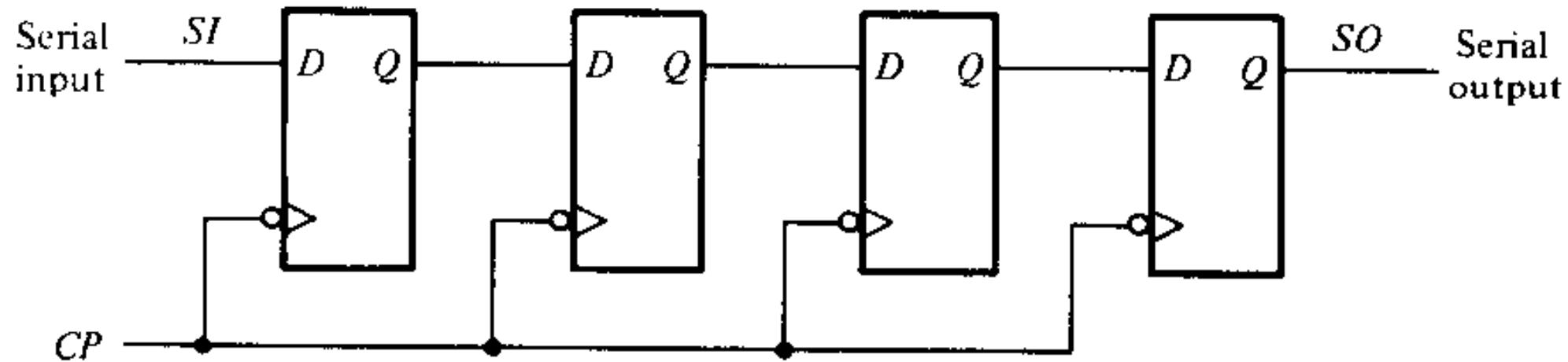
- (a) Design a counter with the following repeated binary sequence: 0, 1, 2, 3, 4, 5, 6. Use *JK* flip-flops.
- (b) Design a counter with the following repeated binary sequence: 0, 1, 2, 4, 6. Use *D* flip-flops.
- (c) Design a counter with the following repeated binary sequence: 0, 1, 3, 5, 7. Use *T* flip-flops.

**Check all of your design to see if it is self correcting or not**

# Registers



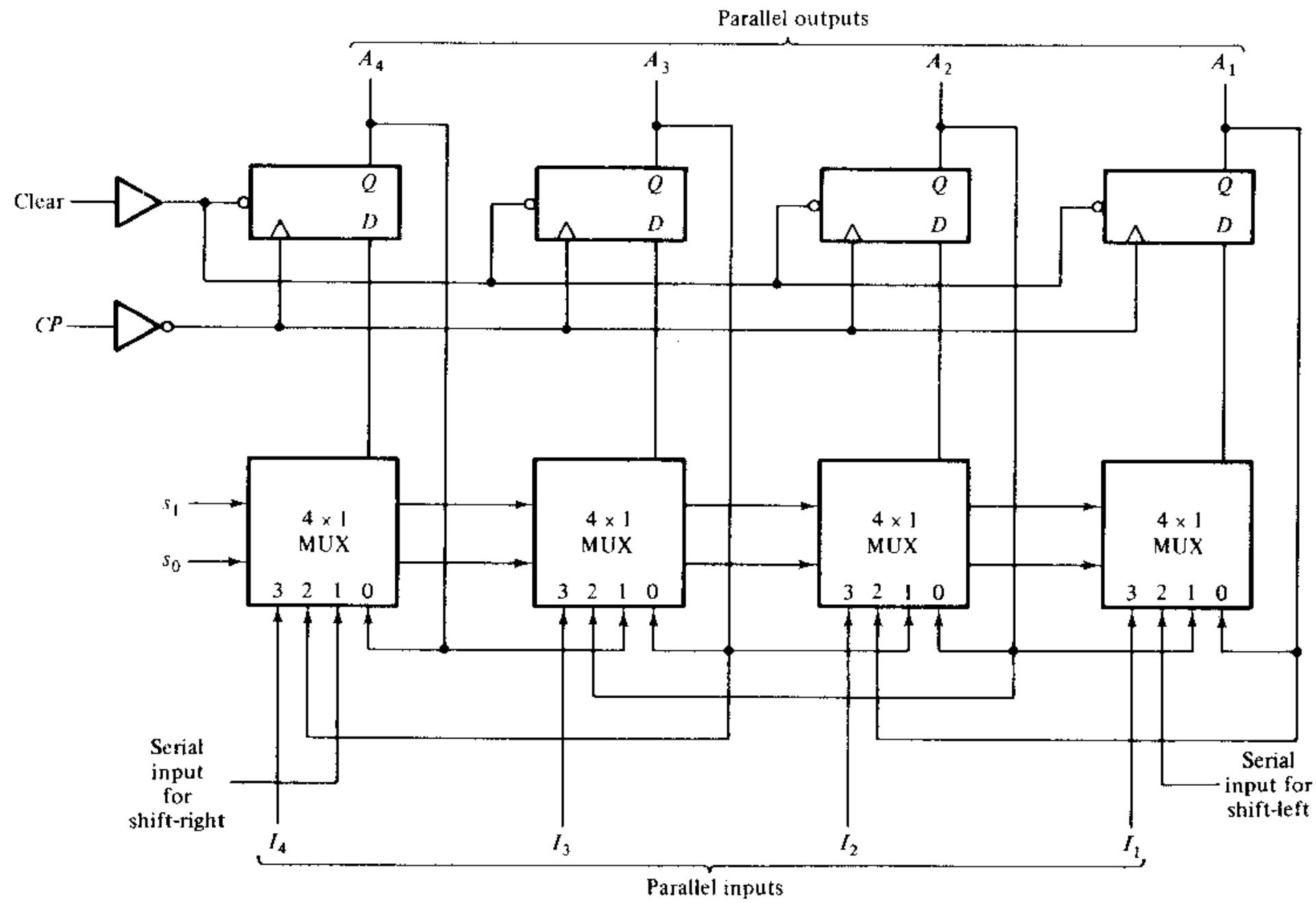
# Shift Register



## **Bidirectional Shift Register with Parallel Load**

**Function Table for the Register of Fig. 7-9**

Mode Control		Register Operation
$s_1$	$s_0$	
0	0	No change
0	1	Shift right
1	0	Shift left
1	1	Parallel load



## Home Work

1. Draw the logic diagram of a four-bit register with four  $D$  flip-flops and four  $4 \times 1$  multiplexers with mode selection inputs  $s_1$  and  $s_0$ . The register operates according to the following function table.

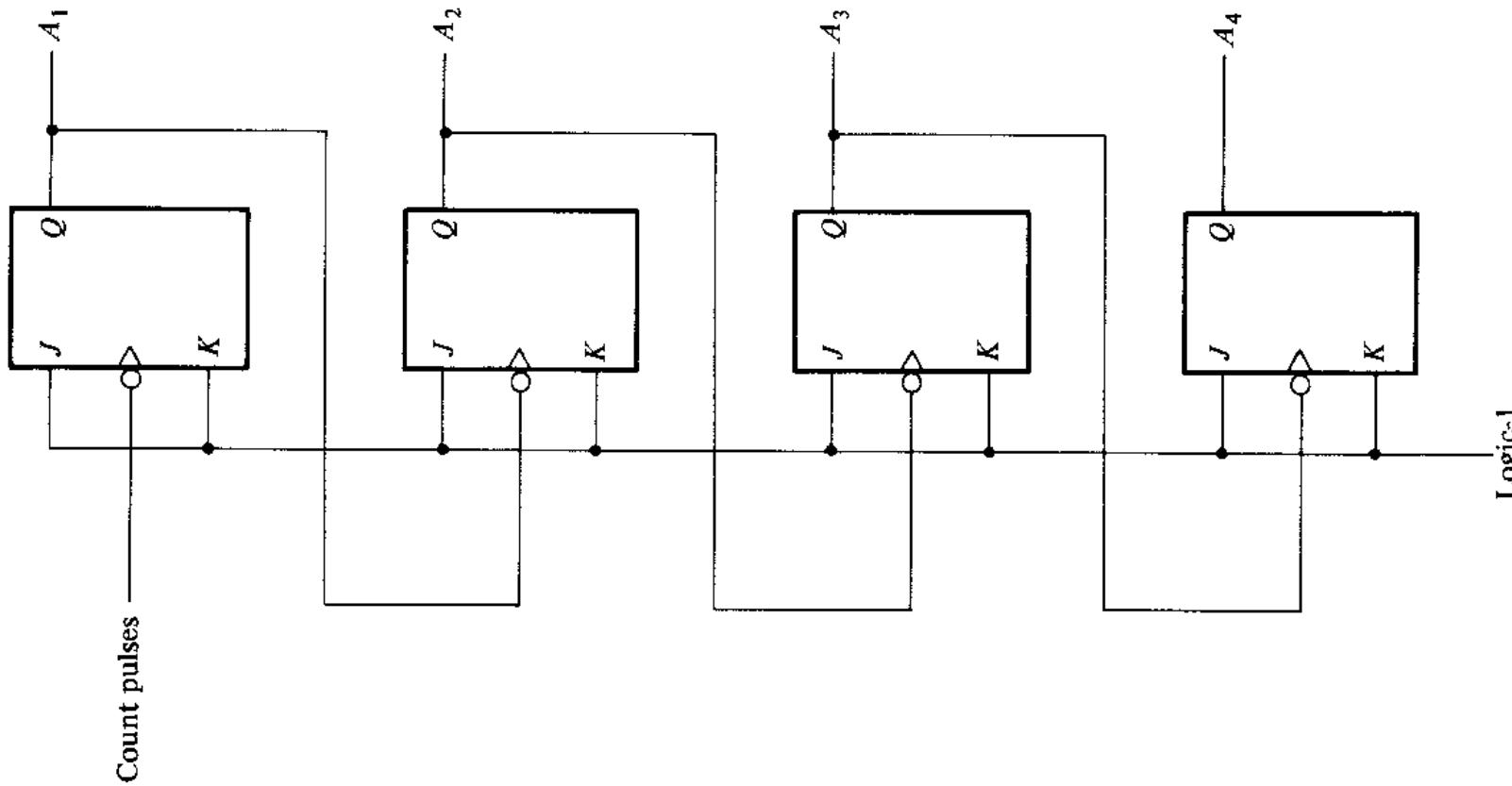
$s_1$	$s_0$	Register Operation
0	0	No change
1	0	Complement the four outputs
0	1	Clear register to 0 (synchronous with the clock)
1	1	Load parallel data

2.

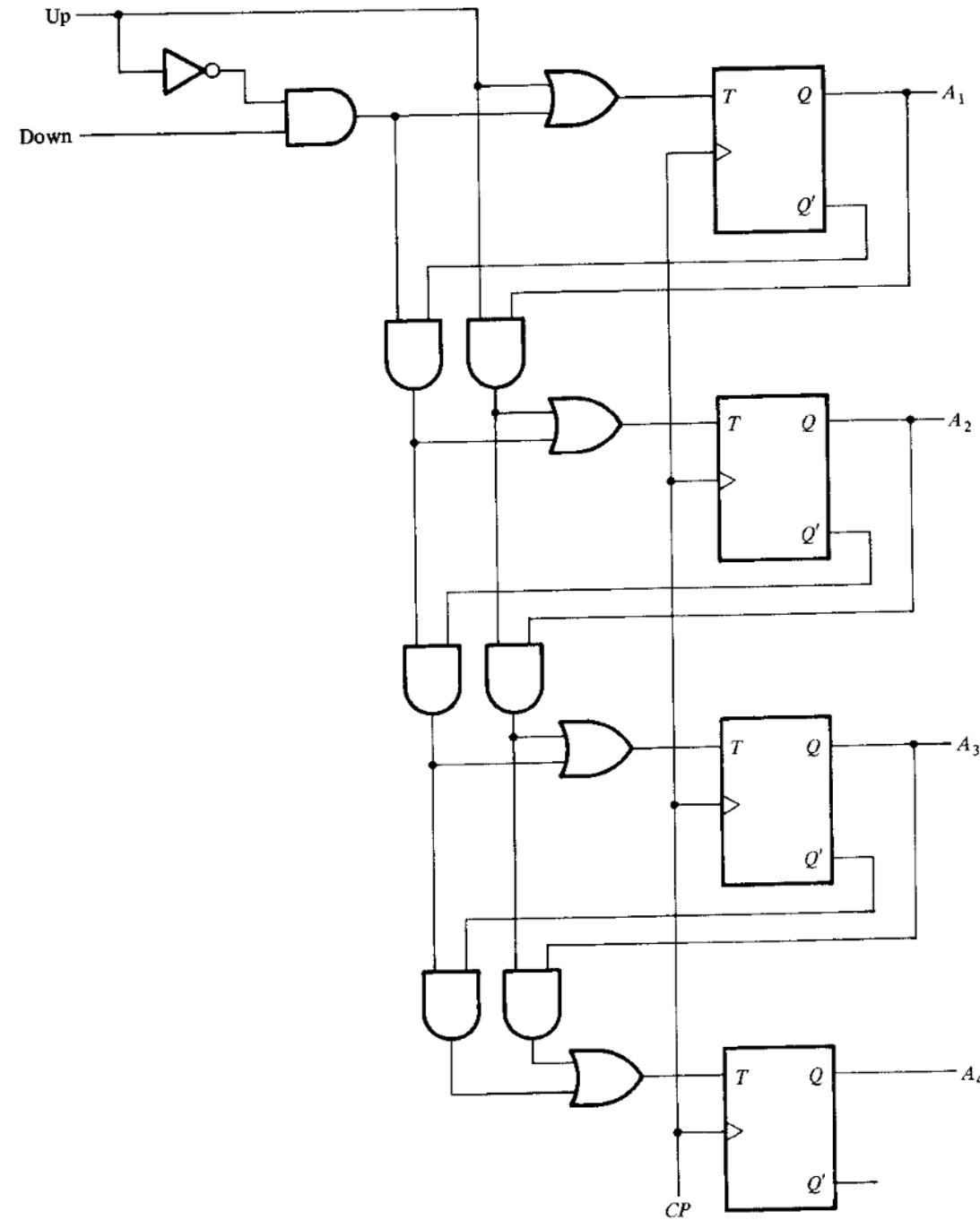
Design a shift register with parallel load that operates according to the following function table:

Shift	Load	Register Operation
0	0	No change
0	1	Load parallel data
1	X	Shift right

# Ripple Counters



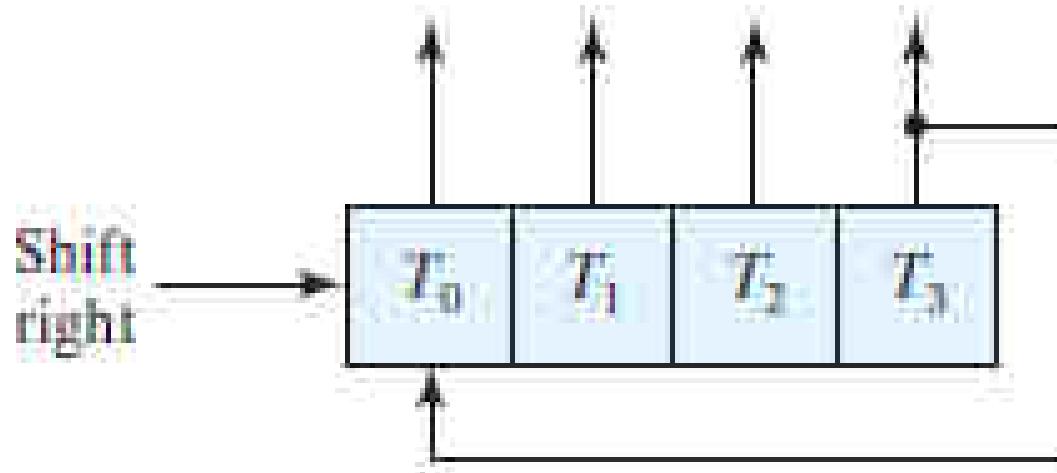
# Synchronous up/down counter



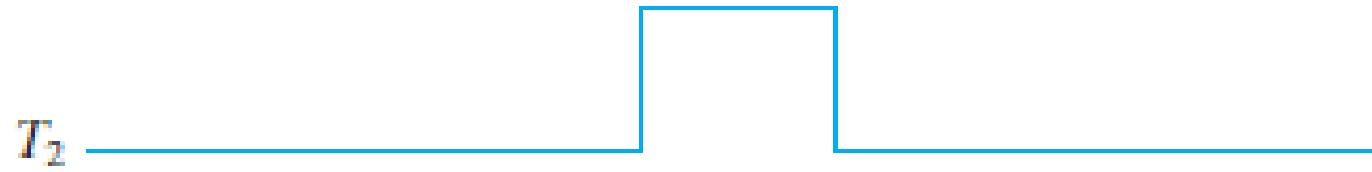
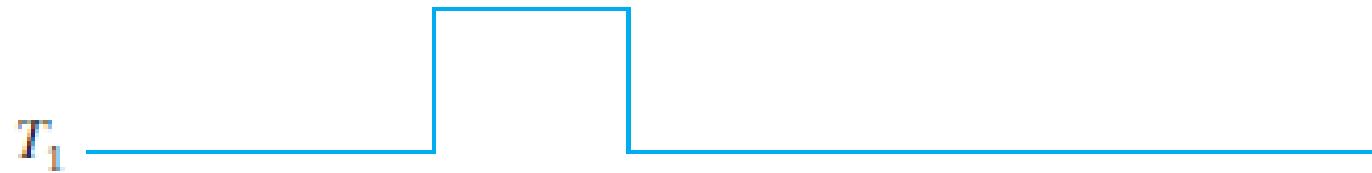
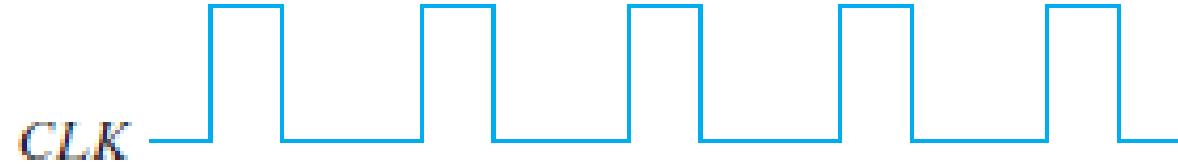


## Ring Counter

Timing signals that control the sequence of operations in a digital system can be generated by a shift register or by a counter with a decoder. A *ring counter* is a circular shift register with only one flip-flop being set at any particular time; all others are cleared. The single bit is shifted from one flip-flop to the next to produce the sequence of timing signals.

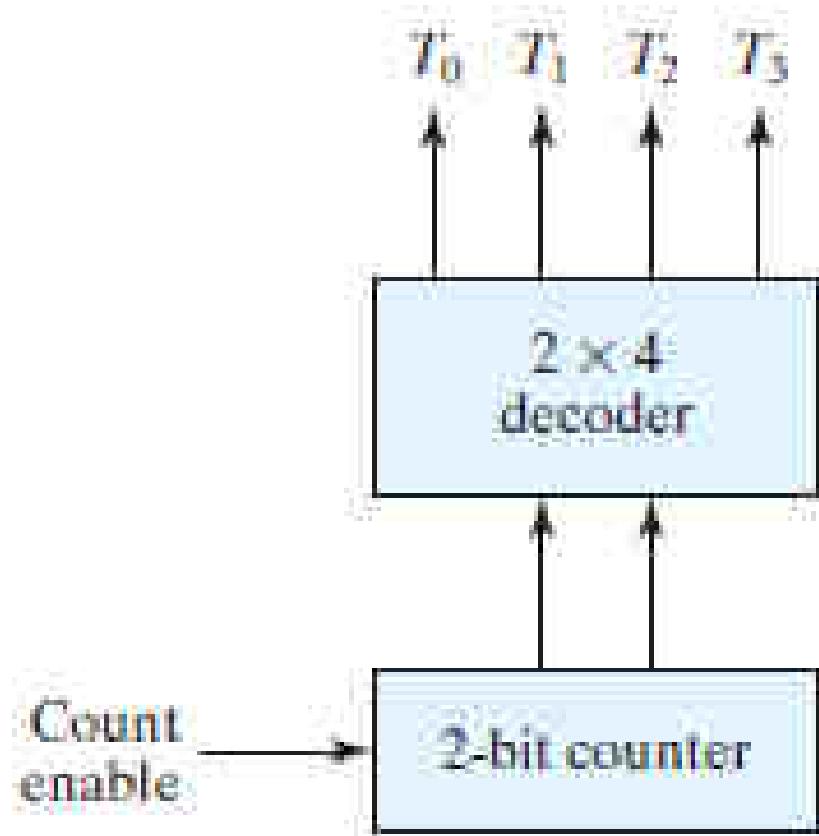


This Figure shows a four-bit shift register connected as a ring counter. The initial value of the register is 1000 which produces the variable  $T_0$ . The single bit is shifted right with every clock pulse and circulates back from  $T_3$  to  $T_0$ . Each flip-flop is in the 1 state once every four clock cycles and produces one of the four timing signals. Each output becomes a 1 after the negative-edge transition of a clock pulse and remains 1 during the next clock cycle.



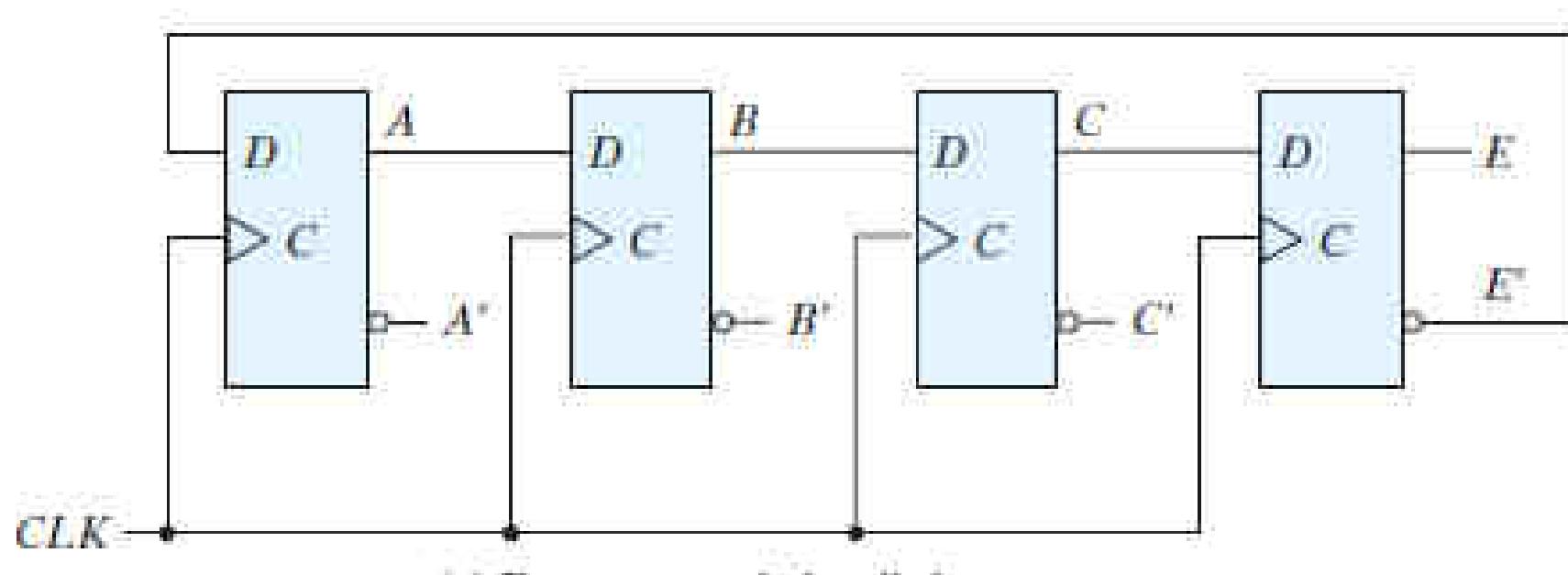
For an alternative design, the timing signals can be generated by a two-bit counter that goes through four distinct states and a decoder. The decoder decodes the four states of the counter and generates the required sequence of timing signals.

To generate  $2^n$  timing signals, we need either a shift register with  $2^n$  flip-flops or an  $n$ -bit binary counter together with an  $n$ -to- $2^n$  -line decoder.



# Johnson Counter

A Johnson Counter or a switch-tail ring counter is a circular shift register with the complemented output of the last flip-flop connected to the input of the first flip-flop.



Sequence number	Flip-flop outputs				AND gate required for output
	A	B	C	E	
1	0	0	0	0	$A'E'$
2	1	0	0	0	$AB'$
3	1	1	0	0	$BC'$
4	1	1	1	0	$CE'$
5	1	1	1	1	$AE'$
6	0	1	1	1	$A'B'$
7	0	0	1	1	$B'C'$
8	0	0	0	1	$CE$

Starting from a cleared state, the Johnson counter goes through a sequence of eight states. In general, a  $k$ -bit Johnson counter will go through a sequence of  $2k$  states. Starting from all 0's, each shift operation inserts 1's from the left until the register is filled with all 1's. In the next sequences, 0's are inserted from the left until the register is again filled with all 0's.

A  $k$ -bit Johnson counter with  $2k$  decoding gates to provide  $2k$  timing signals. The eight AND gates listed in the table, when connected to the circuit, will complete the construction of the Johnson counter. Since each gate is enabled during one particular state sequence, the outputs of the gates generate eight timing signals in succession.

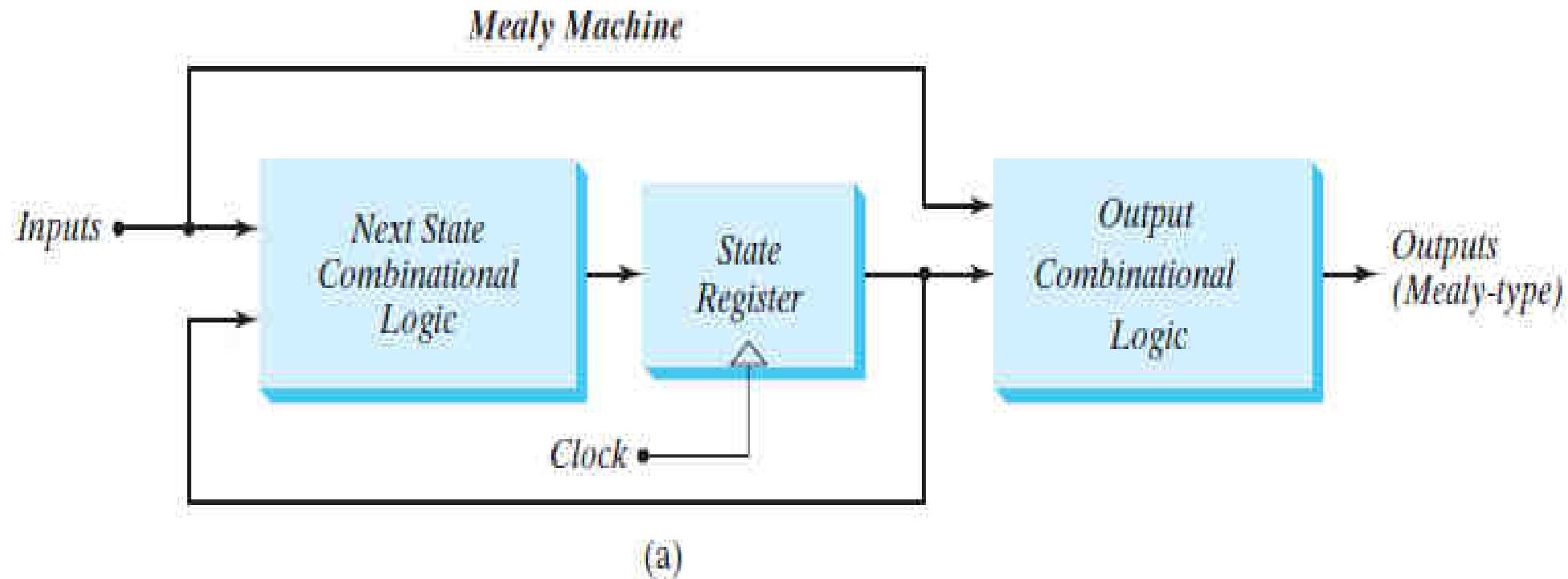
The decoding of a  $k$ -bit Johnson counter to obtain  $2k$  timing signals follows a regular pattern. The all-0's state is decoded by taking the complement of the two extreme flip-flop outputs. The all-1's state is decoded by taking the normal outputs of the two extreme flip-flops. All other states are decoded from an adjacent 1, 0 or 0, 1 pattern in the sequence.

For example, sequence 7 has an adjacent 0, 1 pattern in flip-flops  $B$  and  $C$ . The decoded output is then obtained by taking the complement of  $B$  and the normal output of  $C$ , or  $B'C$ .

# Mealy and Moore Models of Finite State Machines

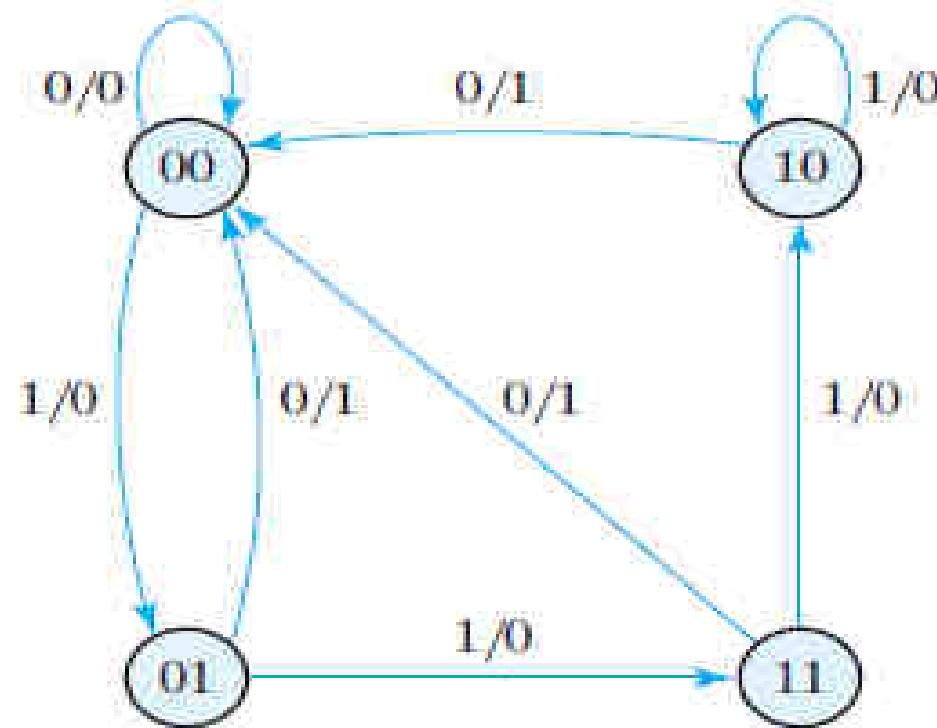
The most general model of a sequential circuit has inputs, outputs, and internal states. It is customary to distinguish between two models of sequential circuits: the Mealy model and the Moore model. They differ only in the way the output is generated.

# Mealy Model



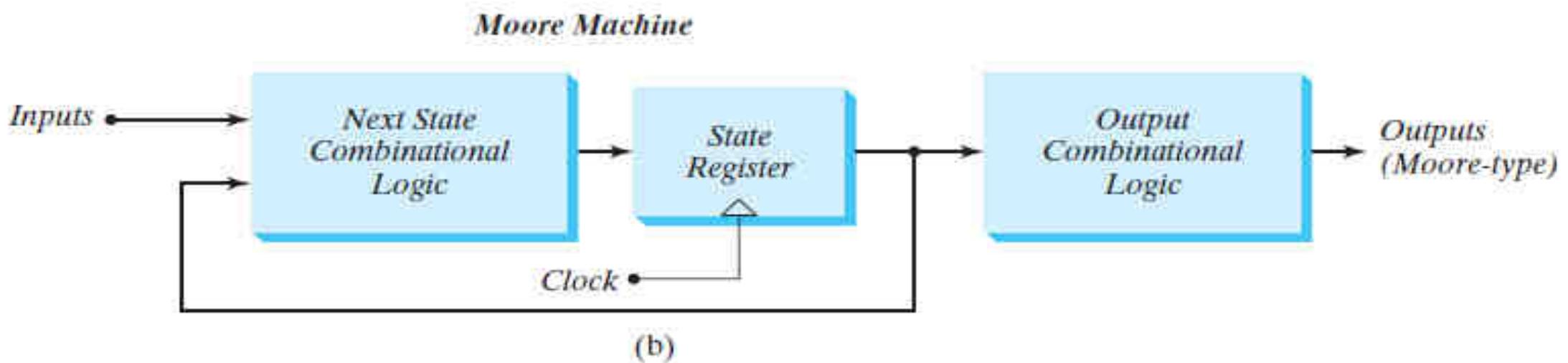
In the Mealy model, the output is a function of both the present state and the input.

Example:  
State Diagram:



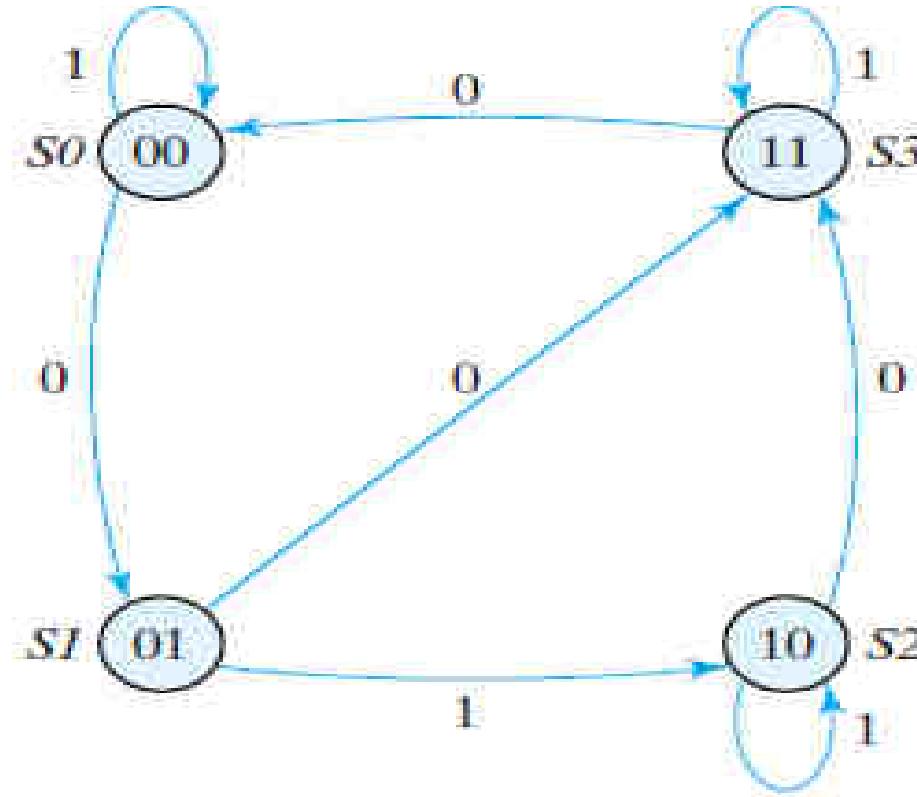
Both the input and output values are shown, separated by a slash along the directed lines between the states.

# Moore Model



- In the Moore model, the output is a function of only the present state.
- A circuit may have both types of outputs.
- The two models of a sequential circuit are commonly referred to as a finite state machine, abbreviated FSM.

Example:  
State Diagram:



It has only inputs marked along the directed lines. The outputs are the flip-flop states marked inside the circles. There may be other outputs also, which are functions of present state only.

Design a circuit that meets the following specification:

1. The circuit has one input,  $w$ , and one output,  $z$ .
2. All changes in the circuit occur on the positive edge of the clock signal.
3. The output  $z$  is equal to 1 if during two immediately preceding clock cycles the input  $w$  was equal to 1. Otherwise, the value of  $z$  is equal to 0.

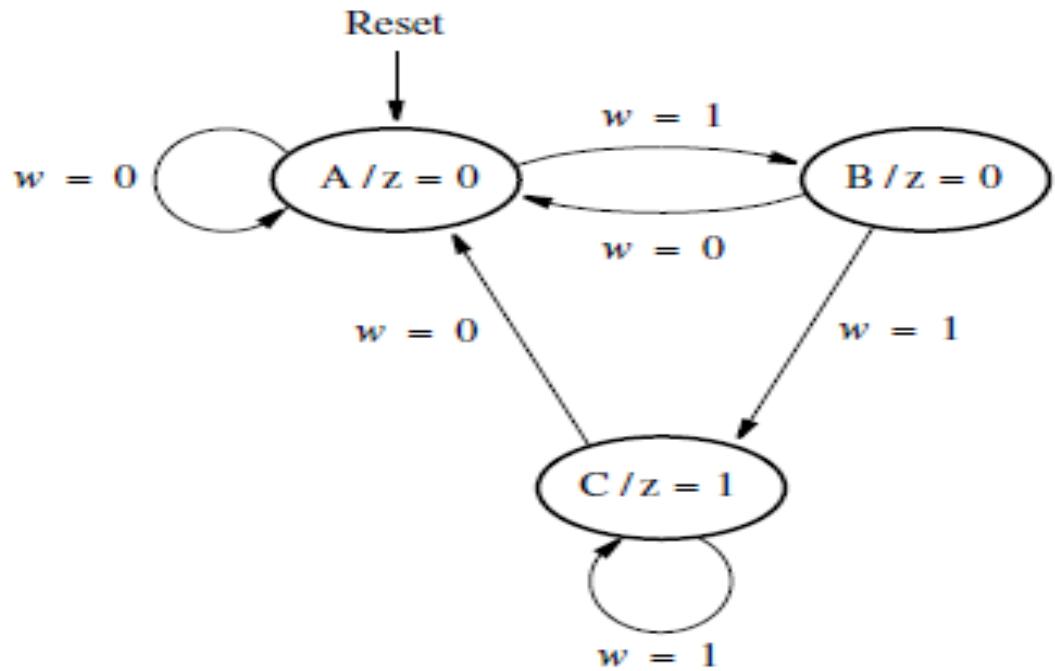
From this specification it is apparent that the output  $z$  depends on both present and past values of  $w$ .

Consider the sequence of values of the  $w$  and  $z$  signals for the clock cycles shown in Figure 6.2.

Clock cycle:	$t_0$	$t_1$	$t_2$	$t_3$	$t_4$	$t_5$	$t_6$	$t_7$	$t_8$	$t_9$	$t_{10}$
$w$ :	0	1	0	1	1	0	1	1	1	0	1
$z$ :	0	0	0	0	0	1	0	0	1	1	0

**Figure 6.2** Sequences of input and output signals.

As seen in the figure, for a given value of input  $w$  the output  $z$  may be either 0 or 1. For example,  $w = 0$  during clock cycles  $t_2$  and  $t_5$ , but  $z = 0$  during  $t_2$  and  $z = 1$  during  $t_5$ . Similarly,  $w = 1$  during  $t_1$  and  $t_8$ , but  $z = 0$  during  $t_1$  and  $z = 1$  during  $t_8$ . This means that  $z$  is not determined only by the present value of  $w$ , so there must exist different states in the circuit that determine the value of  $z$ .



**Figure 6.3** State diagram of a simple sequential circuit.

Present state	Next state		Output $z$
	$w = 0$	$w = 1$	
A	A	B	0
B	A	C	0
C	A	C	1

**Figure 6.4** State table corresponding to Figure 6.3.

*Reset* input is used to force the circuit into state  $A$ , regardless of what state the circuit happens to be in.

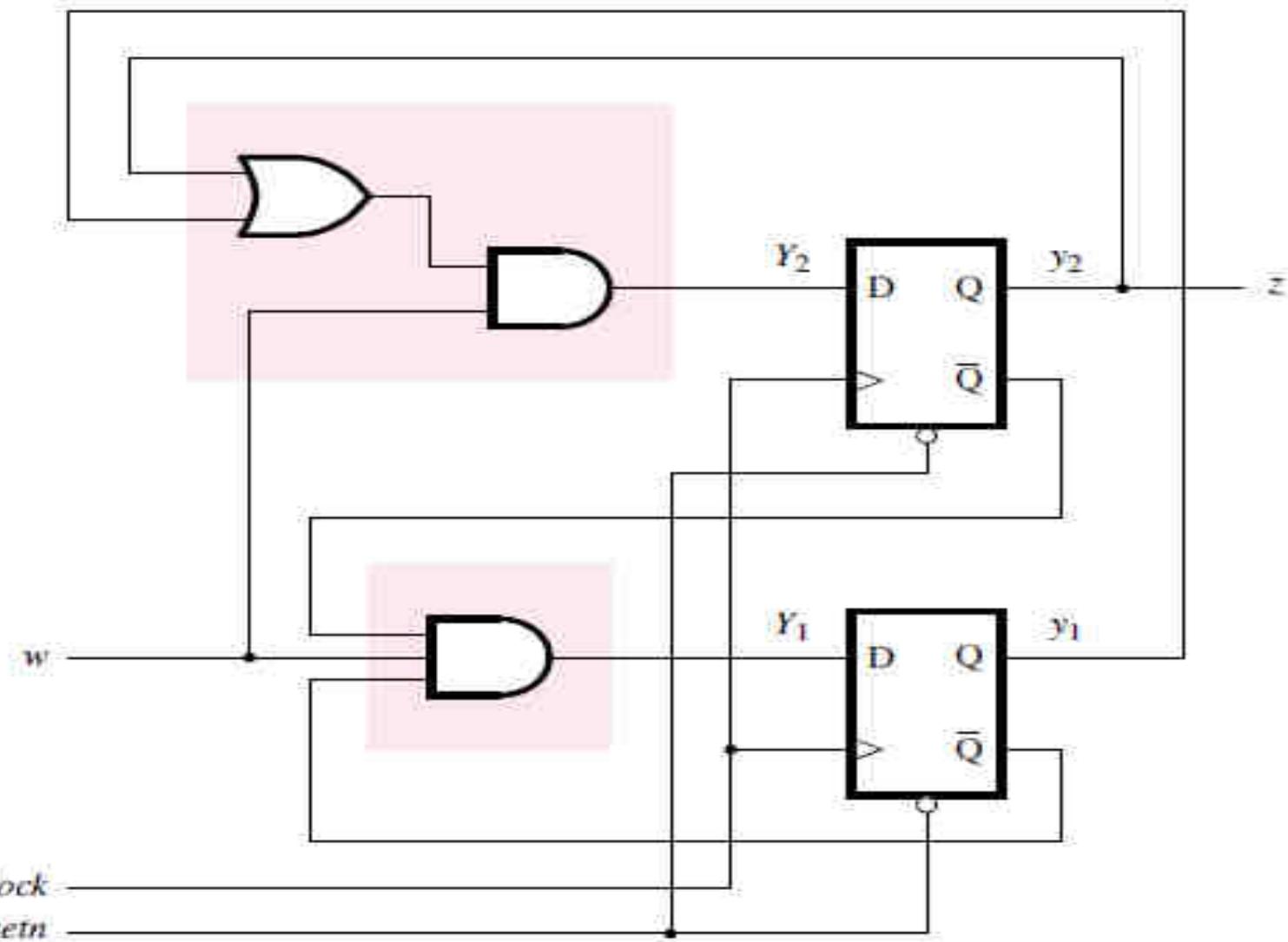
Present state	Next state		Output $z$	
	$w = 0$	$w = 1$		
	$y_2 y_1$	$Y_2 Y_1$		
A	00	00	01	0
B	01	00	10	0
C	10	00	10	1
	11	$dd$	$dd$	$d$

**Figure 6.6** State-assigned table corresponding to Figure 6.4.

$$Y_1 = w \bar{y}_1 \bar{y}_2$$

$$Y_2 = w(y_1 + y_2)$$

$$z = Y_2$$



**Figure 6.8** Final implementation of the sequential circuit.

# Using Verilog Constructs for Storage Elements

```
module flipflop (D, Clock, Q);
input D, Clock;
output reg Q;
always @(posedge Clock)
Q = D;
endmodule
```

## Blocking and Non-Blocking Assignments

```
f = x1 & x2;
```

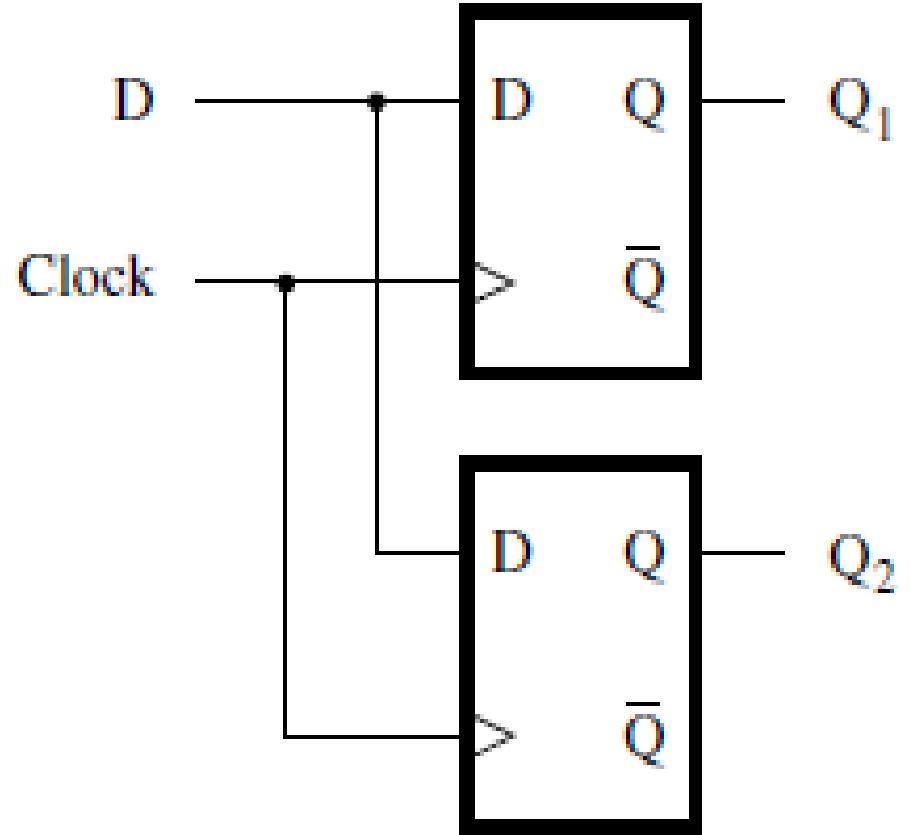
This notation is called a *blocking* assignment.

A Verilog compiler evaluates the statements in an **always** block in the order in which they are written.

If a variable is given a value by a blocking assignment statement, then this new value is used in evaluating all subsequent statements in the block.

```
module example5_3 (D, Clock, Q1, Q2);  
input D, Clock;  
output reg Q1, Q2;  
always @(posedge Clock)  
begin  
Q1 = D;  
Q2 = Q1;  
end  
endmodule
```

Incorrect code for two cascaded flip-flops.



Since the **always** block is sensitive to the positive clock edge, both Q1 and Q2 will be implemented as the outputs of D flip-flops.

However, because blocking assignments are involved, these two flip-flops will not be connected in cascade.

The first statement  $Q1 = D;$  sets Q1 to the value of D. This new value is used in evaluating the subsequent statement  
 $Q2 = Q1;$  which results in  $Q2 = Q1 = D.$

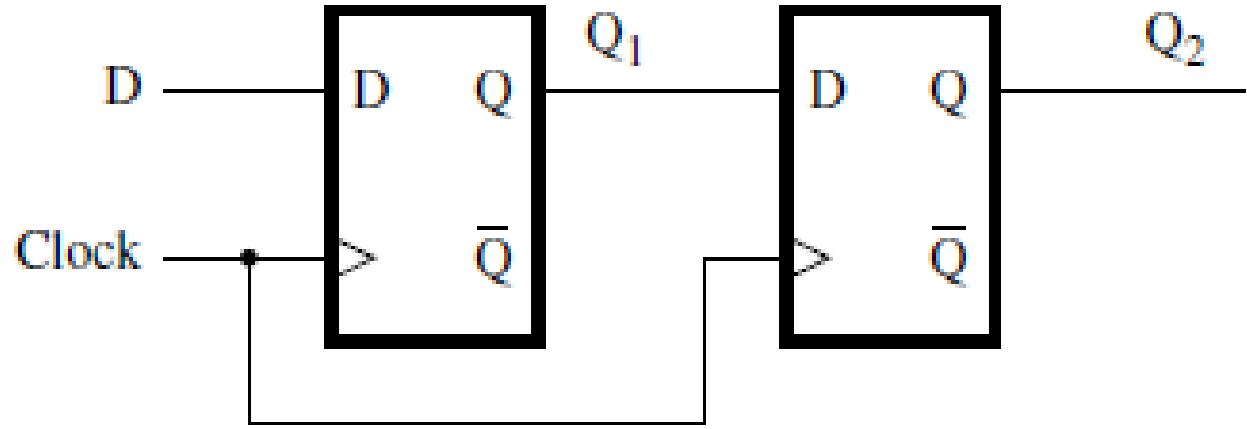
Verilog also provides a *non-blocking* assignment, denoted with `<=`.

All non-blocking assignment statements in an **always** block are evaluated using the values that the variables have when the **always** block is entered.

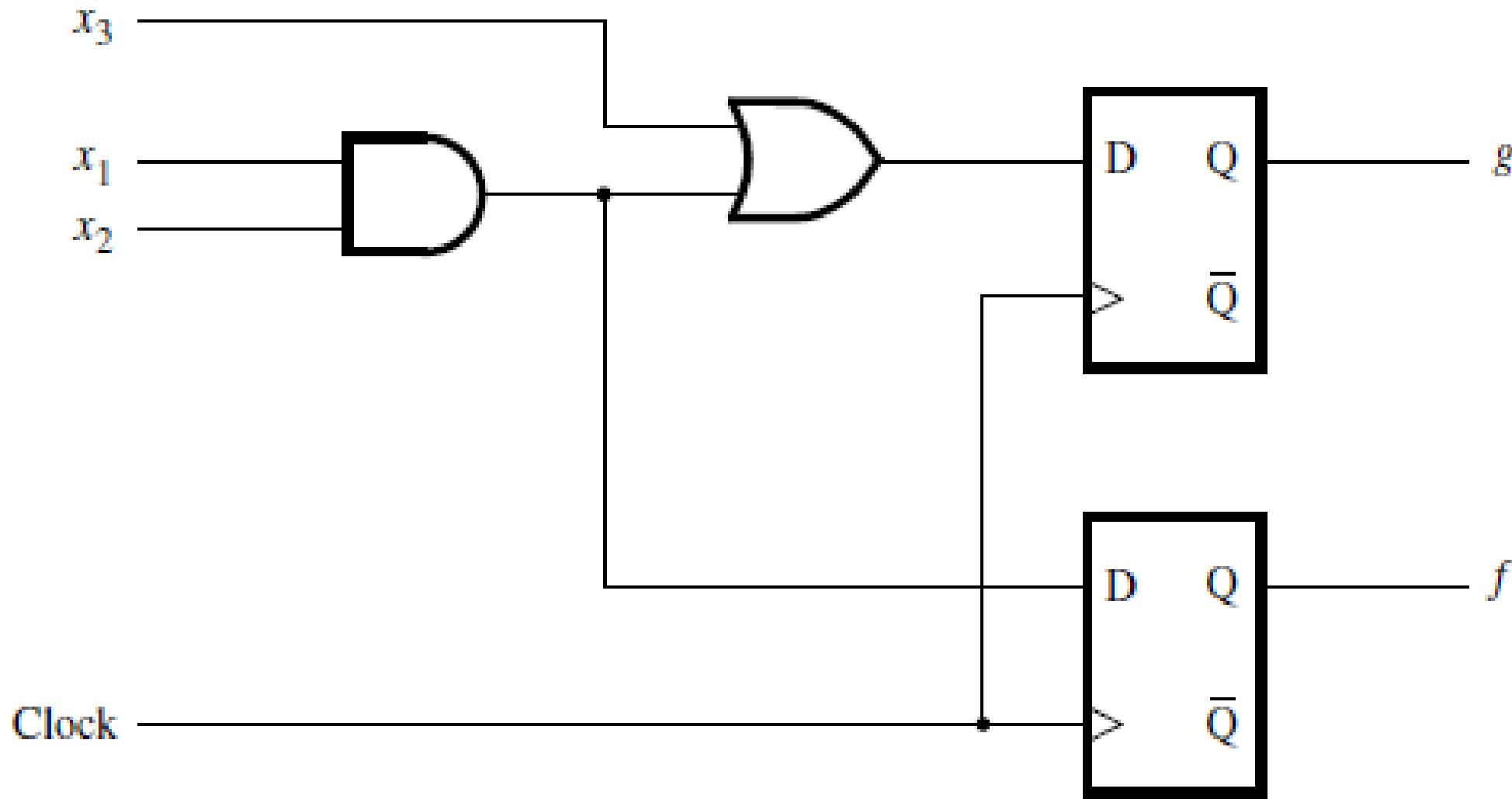
Thus, a given variable has the same value for all statements in the block.

The meaning of non-blocking is that the result of each assignment is not seen until the end of the **always** block.

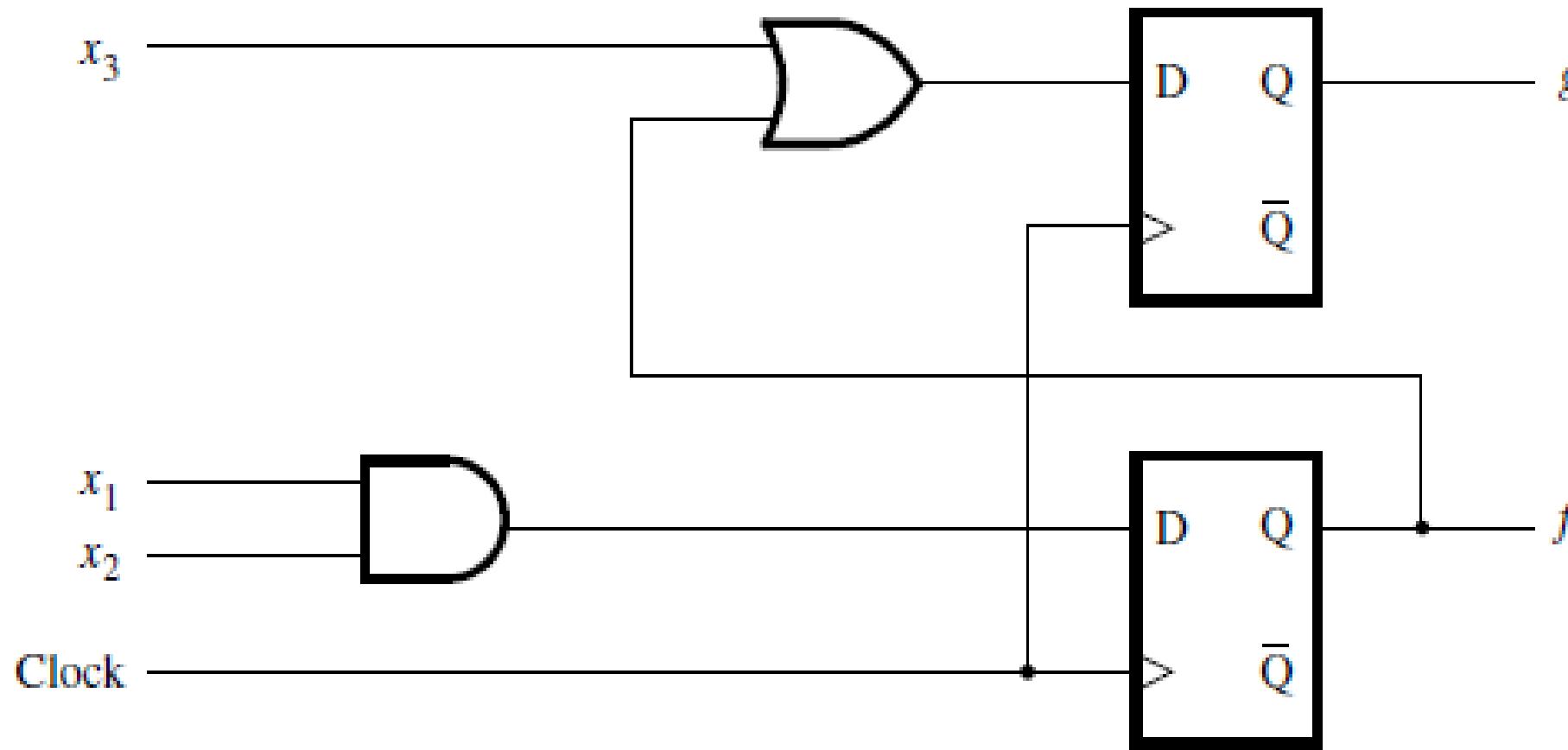
```
module example5_4 (D, Clock, Q1, Q2);
input D, Clock;
output reg Q1, Q2;
always @(posedge Clock)
begin
    Q1 <= D;
    Q2 <= Q1;
end
endmodule
```



```
module example5_5 (x1, x2, x3, Clock, f, g);
input x1, x2, x3, Clock;
output reg f, g;
always @(posedge Clock)
begin
    f = x1 & x2;
    g = f | x3;
end
endmodule
```



```
module example5_6 (x1, x2, x3, Clock, f, g);
input x1, x2, x3, Clock;
output reg f, g;
always @(posedge Clock)
begin
    f <= x1 & x2;
    g <= f | x3;
end
endmodule
```



## ASYNCHRONOUS CLEAR

Following module defines a D flip-flop with an asynchronous active-low reset (clear) input.

When *Resetn*, the reset input, is equal to 0, the flip-flop's Q output is set to 0.

The sensitivity list specifies the negative edge of *Resetn* as an event trigger along with the positive edge of the clock.

We cannot omit the keyword **negedge** because the sensitivity list cannot have both edge-triggered and level sensitive signals.

```
module flipflop (D, Clock, Resetn, Q);
input D, Clock, Resetn;
output reg Q;
always @(negedge Resetn or posedge Clock)
if (!Resetn)
    Q <= 0;
else
    Q <= D;
endmodule
```

# **SYNCHRONOUS CLEAR**

Module below shows how a D flip-flop with a synchronous reset input can be described.

In this case the reset signal is acted upon only when a positive clock edge arrives.

```
module flipflop (D, Clock, Resetn, Q);
input D, Clock, Resetn;
output reg Q;
always @(posedge Clock)
if (!Resetn)
    Q <= 0;
else
    Q <= D;
endmodule
```

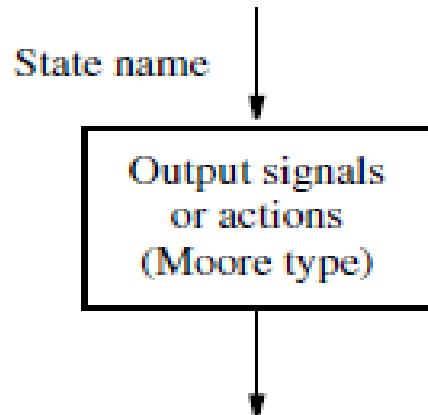
```
module regn (D, Clock, Resetn, Q);
parameter n = 16;
input [n -1:0] D;
input Clock, Resetn;
output reg [n -1:0] Q;
always @ (negedge Resetn, posedge Clock)
if (!Resetn)
    Q <= 0;
else
    Q <= D;
endmodule
```

```
module shift4 (w, Clock, Q);
input w, Clock;
output reg [3:0] Q;
always @(posedge Clock)
begin
    Q[0] <= Q[1];
    Q[1] <= Q[2];
    Q[2] <= Q[3];
    Q[3] <= w;
end
endmodule
```

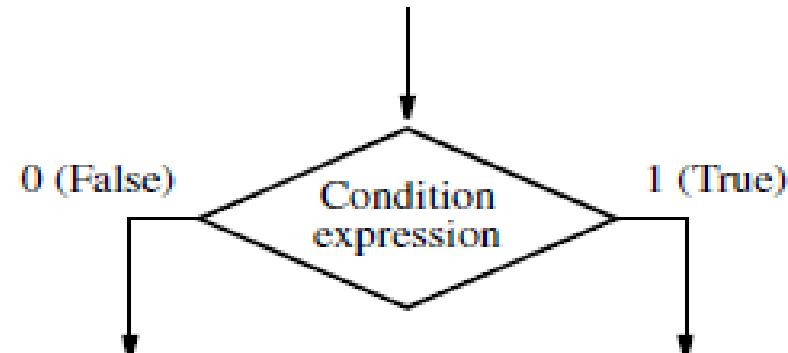
**for** (**k** = 0; **k** < **n** - 1; **k** = **k**+1)  
**Q[k] <= Q[k+1];**  
**Q[n-1] <= w;**

# Algorithmic State Machine (ASM) Charts

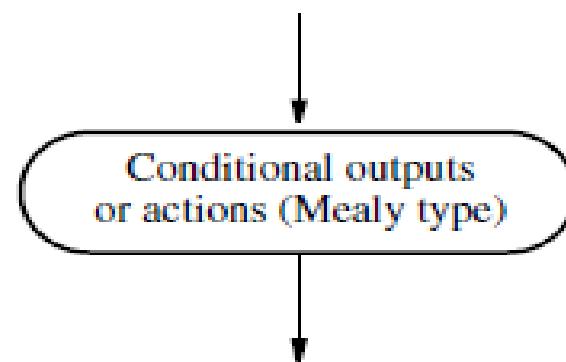
- For larger machines the designers often use a different form of representation, called the *algorithmic state machine (ASM) chart*.
- An ASM chart is a type of flowchart that can be used to represent the state transitions and generated outputs for an FSM.



(a) State box



(b) Decision box



(c) Conditional output box

**Figure 6.81** Elements used in ASM charts.

## State box:

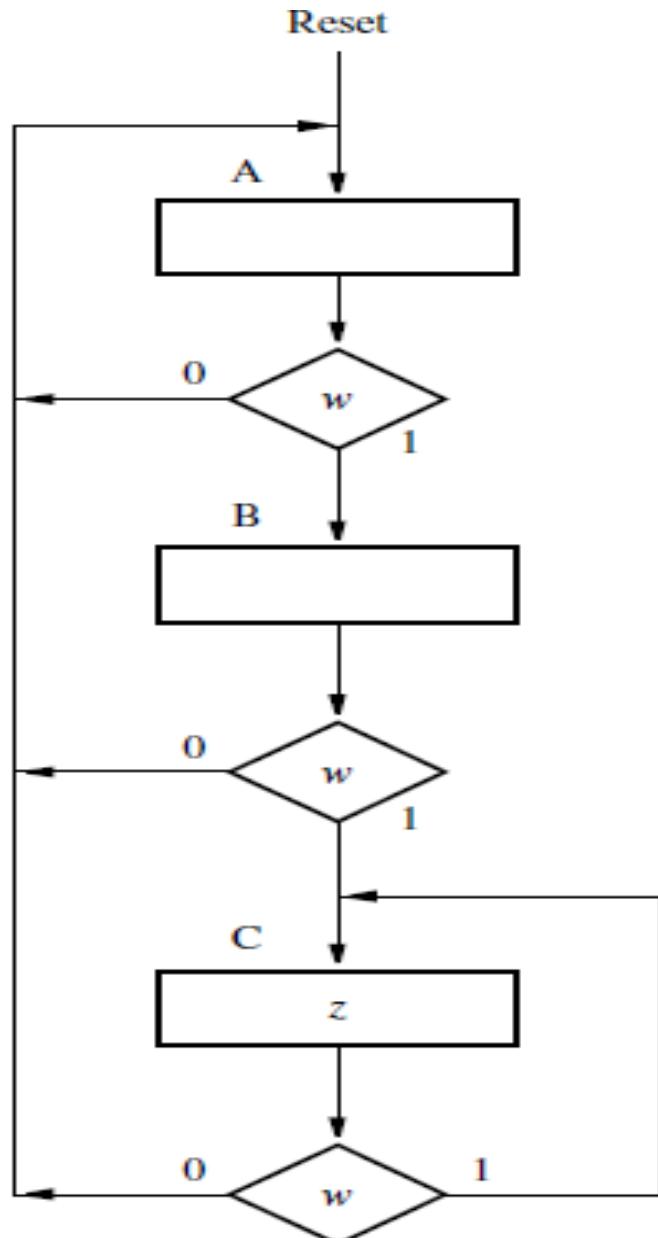
- A rectangle represents a state of the FSM. It is equivalent to a node in the state diagram or a row in the state table. The name of the state is indicated outside the box in the top-left corner.
- The Moore-type outputs are listed inside the box. It is customary to write only the name of the signal that has to be asserted. Thus it is sufficient to write  $z$ , rather than  $z = 1$ , to indicate that the output  $z$  must have the value 1.
- Also, it may be useful to indicate an action that must be taken; for example,  $Count \leftarrow Count + 1$  specifies that the contents of a counter have to be incremented by 1.

## **Decision box:**

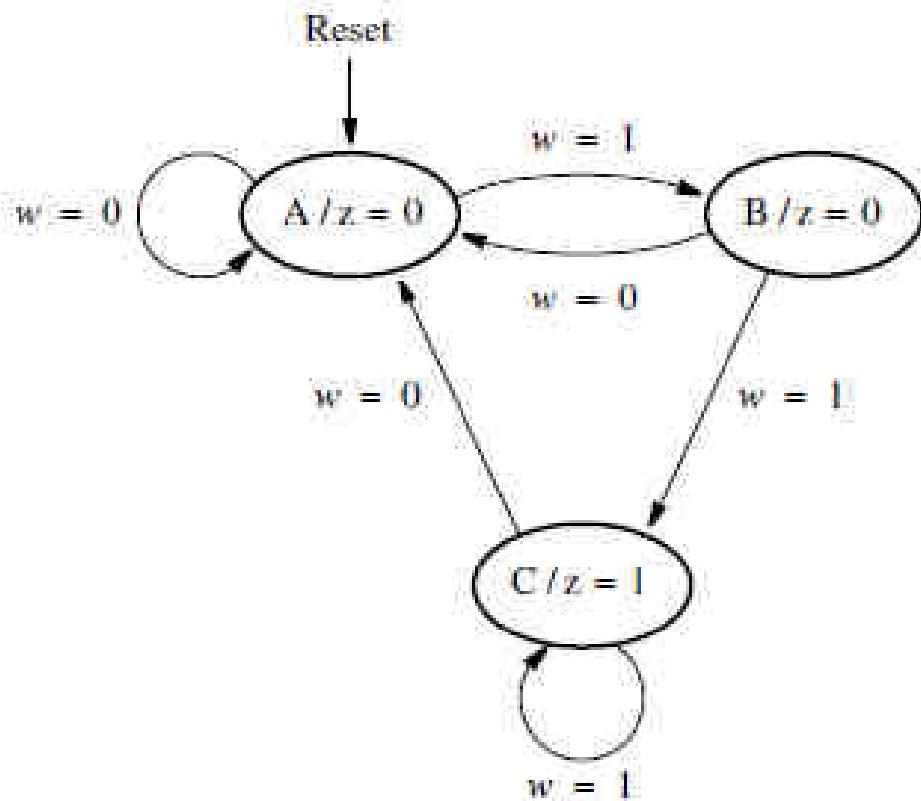
- A diamond indicates that the stated condition expression is to be tested and the exit path is to be chosen accordingly.
- The condition expression consists of one or more inputs to the FSM. For example,  $w$  indicates that the decision is based on the value of the input  $w$ , whereas  $w_1 \cdot w_2$  indicates that the true path is taken if  $w_1 = w_2 = 1$  and the false path is taken otherwise.

## **Conditional output box:**

- An oval denotes the output signals that are of Mealy type.
- The condition that determines whether such outputs are generated is specified in a decision box.



**Figure 6.82** ASM chart for the FSM in Figure 6.3.



**Figure 6.3** State diagram of a simple sequential circuit.

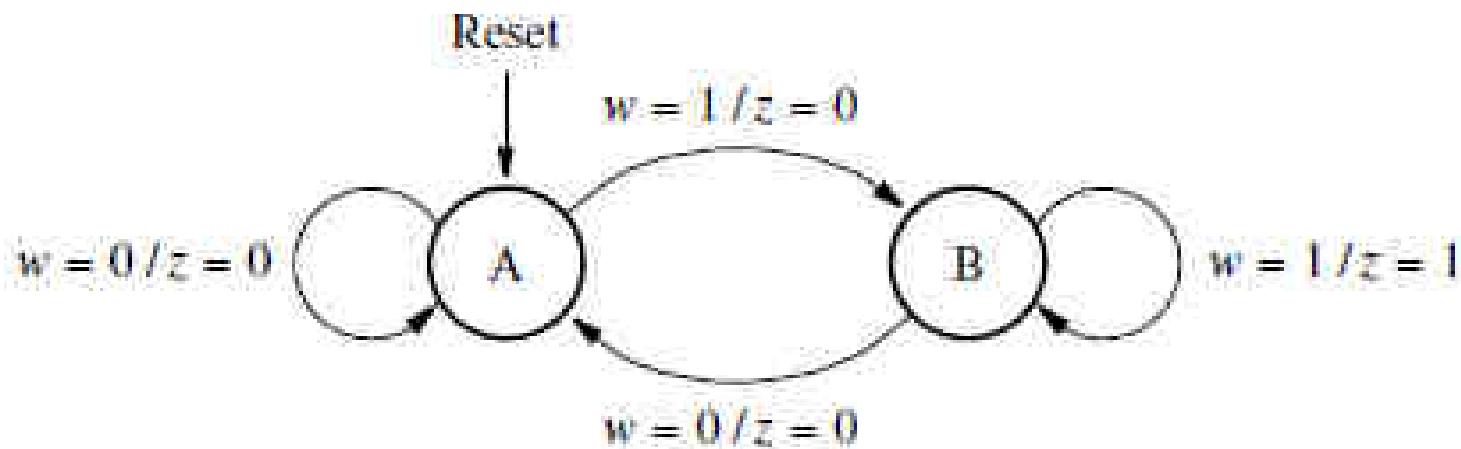
# Mealy State Model

- output values are generated based on both the state of the circuit and the present values of its inputs.

Examlpe: with respect to the previous example, the output  $z$  should be equal to 1 in the same clock cycle when the second occurrence of  $w = 1$  is detected.

Clock cycle:	$t_0$	$t_1$	$t_2$	$t_3$	$t_4$	$t_5$	$t_6$	$t_7$	$t_8$	$t_9$	$t_{10}$
$w:$	0	1	0	1	1	0	1	1	1	0	1
$z:$	0	0	0	0	1	0	0	1	1	0	0

**Figure 6.22** Sequences of input and output signals.



**Figure 6.23** State diagram of an FSM that realizes the task in Figure 6.22.

Present state	Next state		Output $z$	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	A	B	0	0
B	A	B	0	1

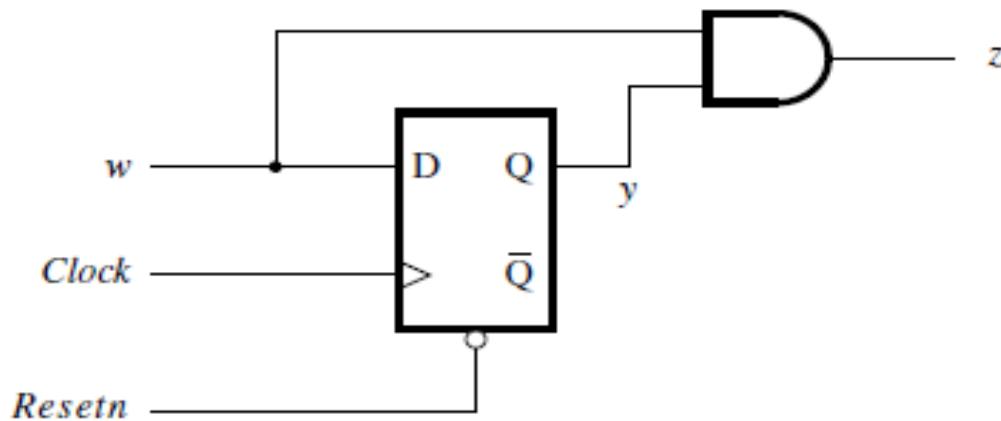
**Figure 6.24** State table for the FSM in Figure 6.23.

Present state	Next state		Output	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
y	$Y$	$Y$	$z$	$z$
A	0	1	0	0
B	0	1	0	1

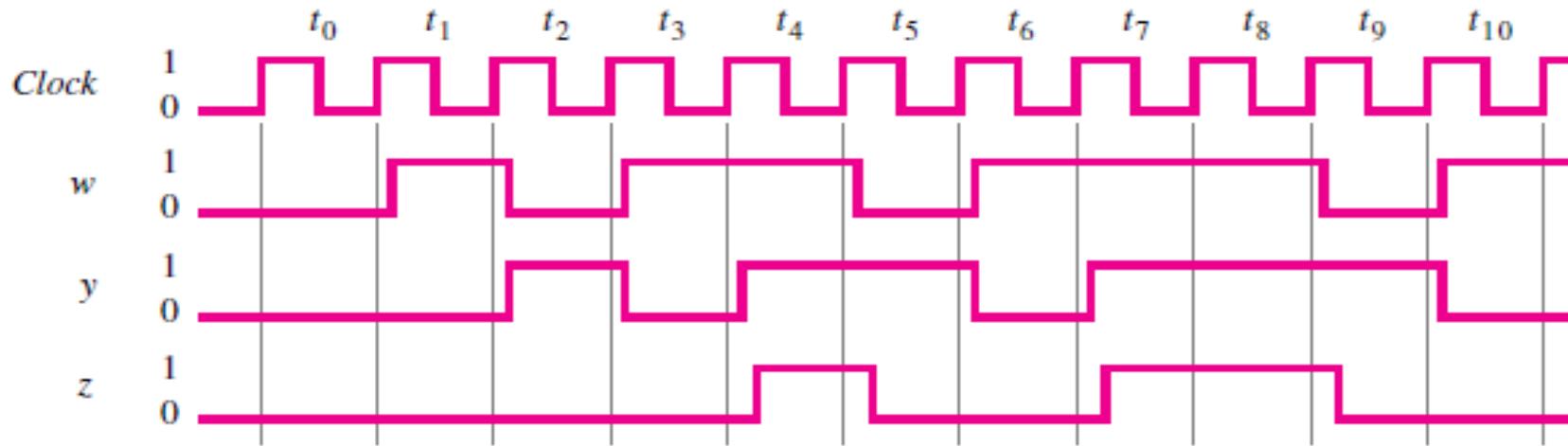
$$Y = D = w$$

$$z = wy$$

**Figure 6.25** State-assigned table for the FSM in Figure 6.24.

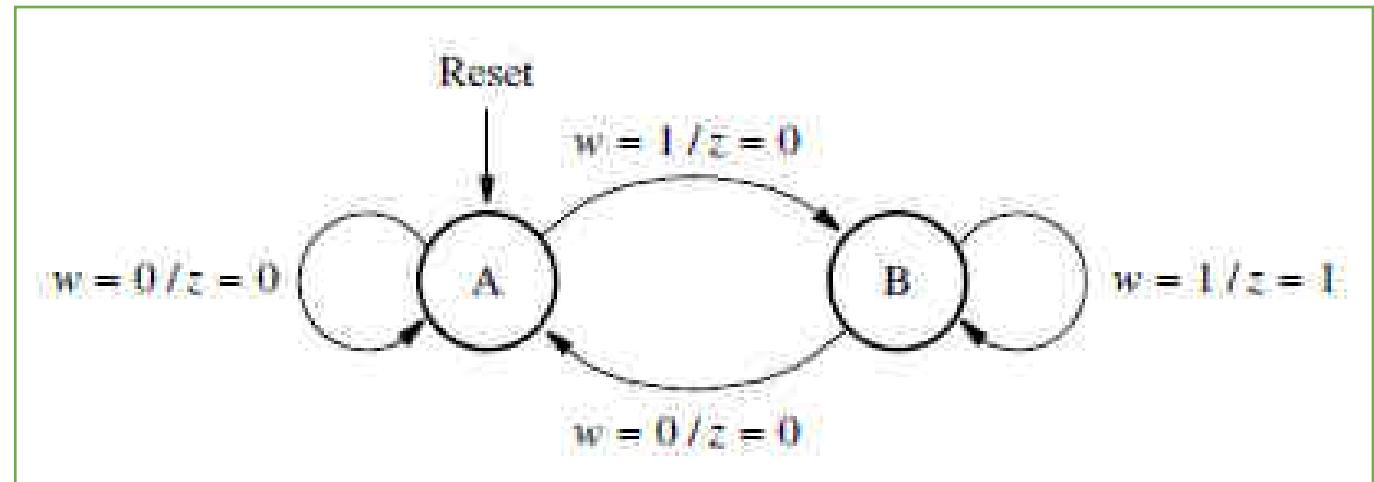
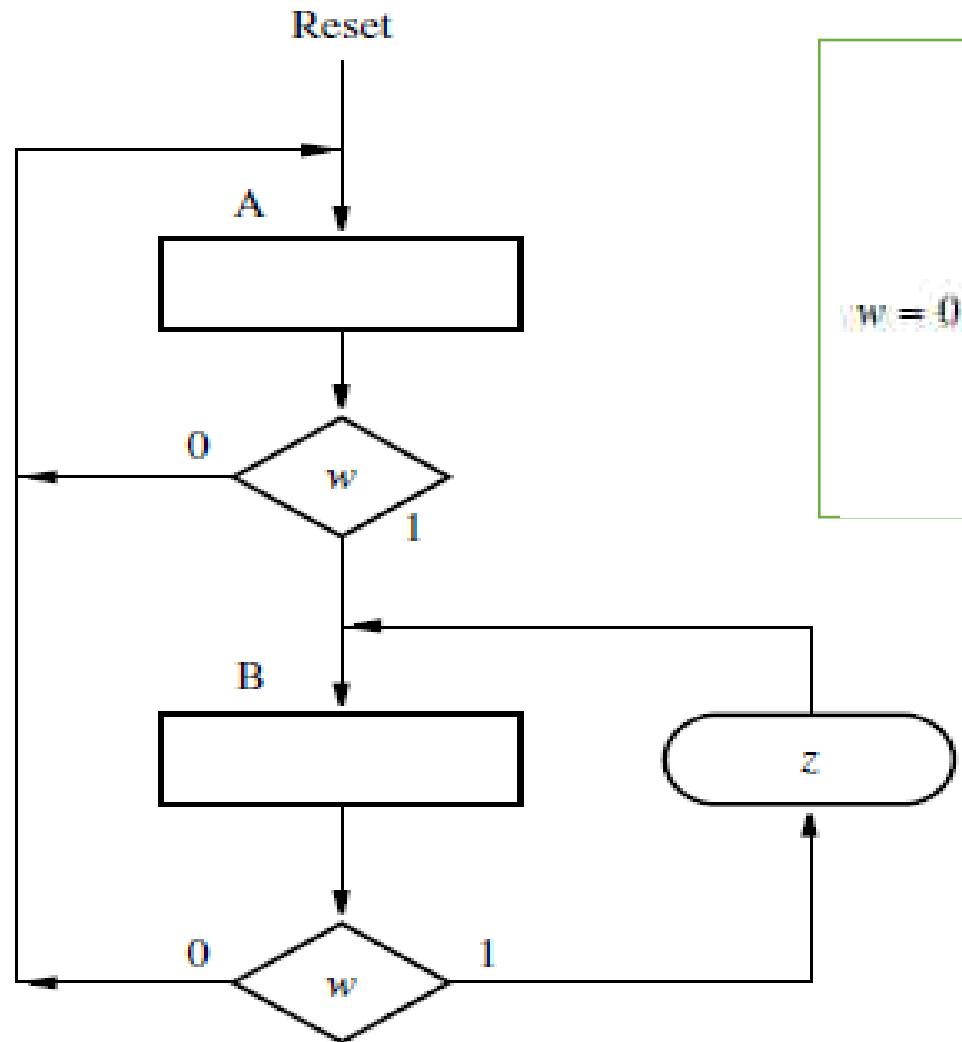


(a) Circuit



(b) Timing diagram

**Figure 6.26** Implementation of the FSM in Figure 6.25.



**Figure 6.83** ASM chart for the FSM in Figure 6.23.

Example: swap the contents of two registers

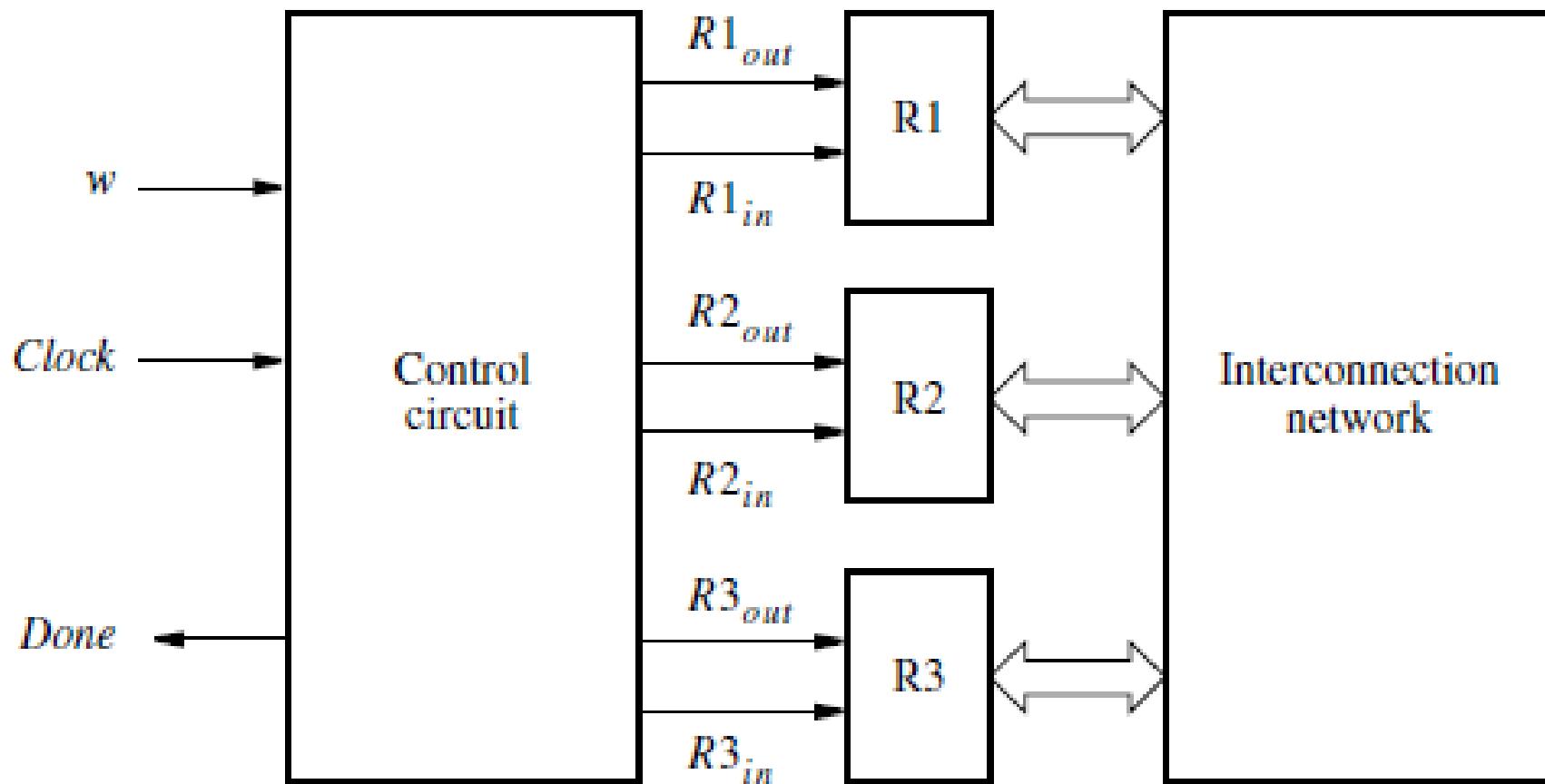
R1 $\leftarrow\rightarrow$ R2

R3 $\leftarrow$ R2

R2 $\leftarrow$ R1

R1 $\leftarrow$ R3

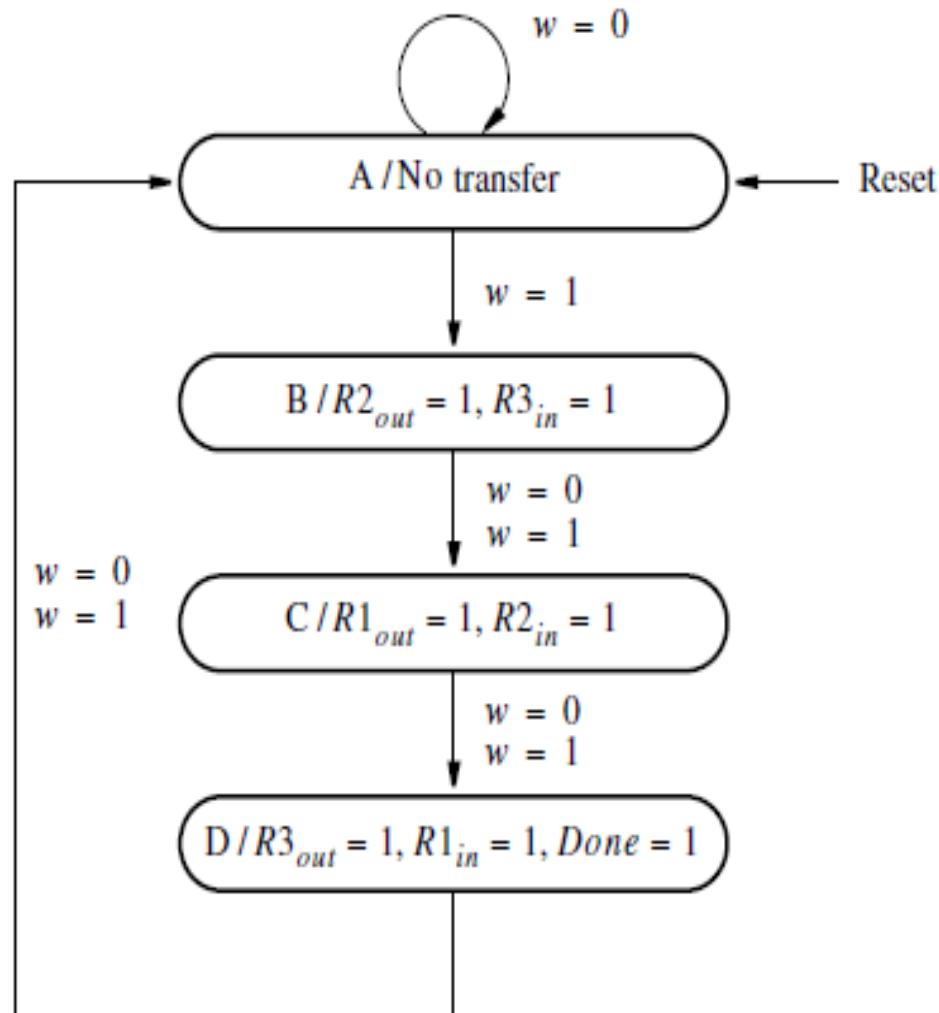
R3 is temp reg.



**Figure 6.10** System for Example 6.1.

- Each register has two control signals: the  $Rkout$  signal causes the contents of register  $Rk$  to be placed into the interconnection network; the  $Rkin$  signal causes the data from the network to be loaded into  $Rk$ .
- The  $Rkout$  and  $Rkin$  signals are generated by a control circuit, which is a finite state machine.
- We will design a control circuit that swaps the contents of  $R1$  and  $R2$ , in response to an input  $w = 1$ .
- Therefore, the inputs to the control circuit will be  $w$  and Clock. The outputs will be  $R1out$  ,  $R1in$ ,  $R2out$  ,  $R2in$ ,  $R3out$  ,  $R3in$ , and Done which indicates the completion of the required transfers.

- The contents of  $R2$  are first loaded into  $R3$ , using the control signals  $R2out = 1$  and  $R3in = 1$ .
- Then the contents of  $R1$  are transferred into  $R2$ , using  $R1out = 1$  and  $R2in = 1$ .
- Finally, the contents of  $R3$  (which are the previous contents of  $R2$ ) are transferred into  $R1$ , using  $R3out = 1$  and  $R1in = 1$ . Since this step completes the required swap, we will set the signal  $Done = 1$ .



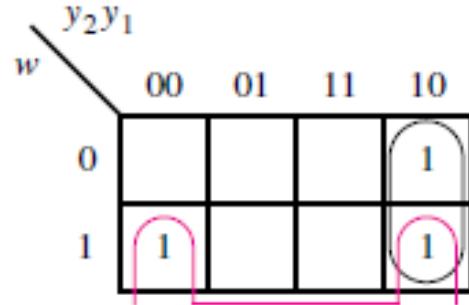
**Figure 6.11** State diagram for Example 6.1.

Present state	Next state		Outputs						
	$w = 0$	$w = 1$	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	A	B	0	0	0	0	0	0	0
B	C	C	0	0	1	0	0	1	0
C	D	D	1	0	0	1	0	0	0
D	A	A	0	1	0	0	1	0	1

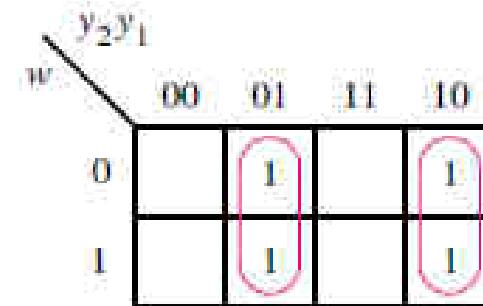
**Figure 6.12** State table for Example 6.1.

Present state	Next state		Outputs						
	$w = 0$	$w = 1$	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	A	B	0	0	0	0	0	0	0
B	C	C	0	0	1	0	0	1	0
C	D	D	1	0	0	1	0	0	0
D	A	A	0	1	0	0	1	0	1

Present state	Next state		Outputs							
	$w = 0$	$w = 1$								
	$y_2 y_1$	$Y_2 Y_1$	$Y_2 Y_1$	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	00	00	01	0	0	0	0	0	0	0
B	01	10	10	0	0	1	0	0	1	0
C	10	11	11	1	0	0	1	0	0	0
D	11	00	00	0	1	0	0	1	0	1



$$Y_1 = w\bar{y}_1 + \bar{y}_1y_2$$



$$Y_2 = y_1\bar{y}_2 + \bar{y}_1y_2$$

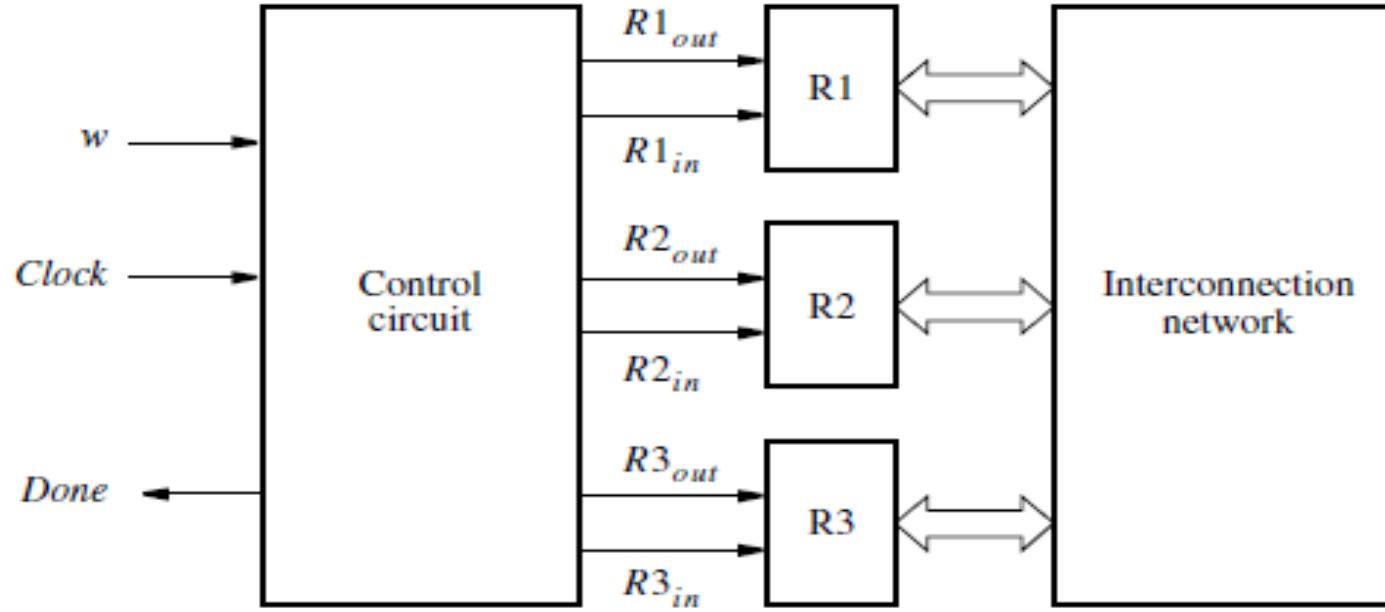
The output control signals are derived as

$$R1_{out} = R2_{in} = \bar{y}_1y_2$$

$$R1_{in} = R3_{out} = Done = y_1y_2$$

$$R2_{out} = R3_{in} = y_1\bar{y}_2$$

# Bus Structure

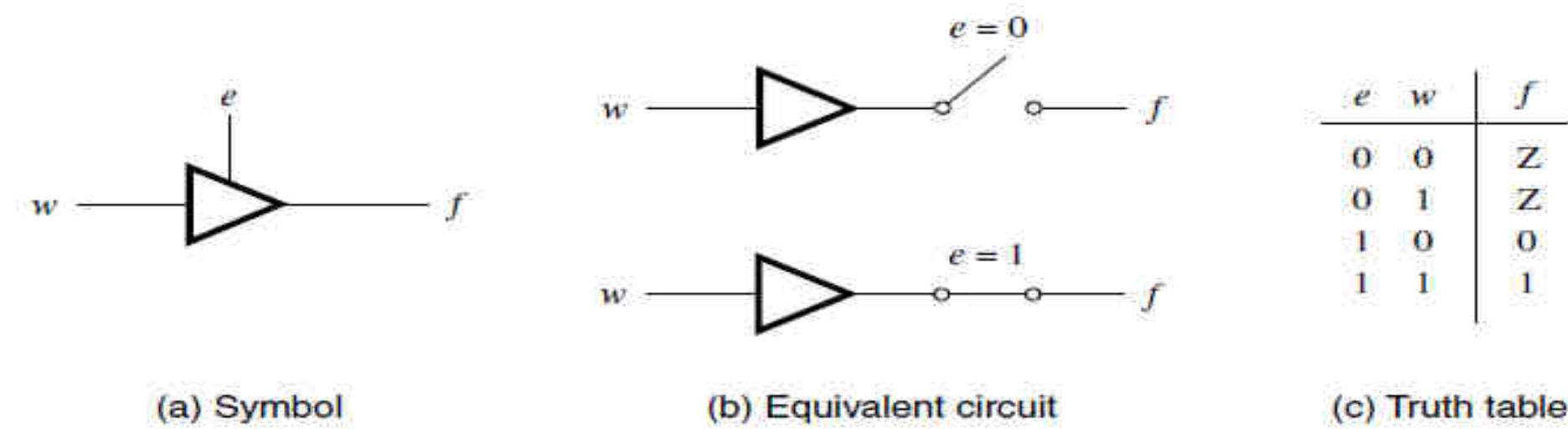


**Figure 6.10** System for Example 6.1.

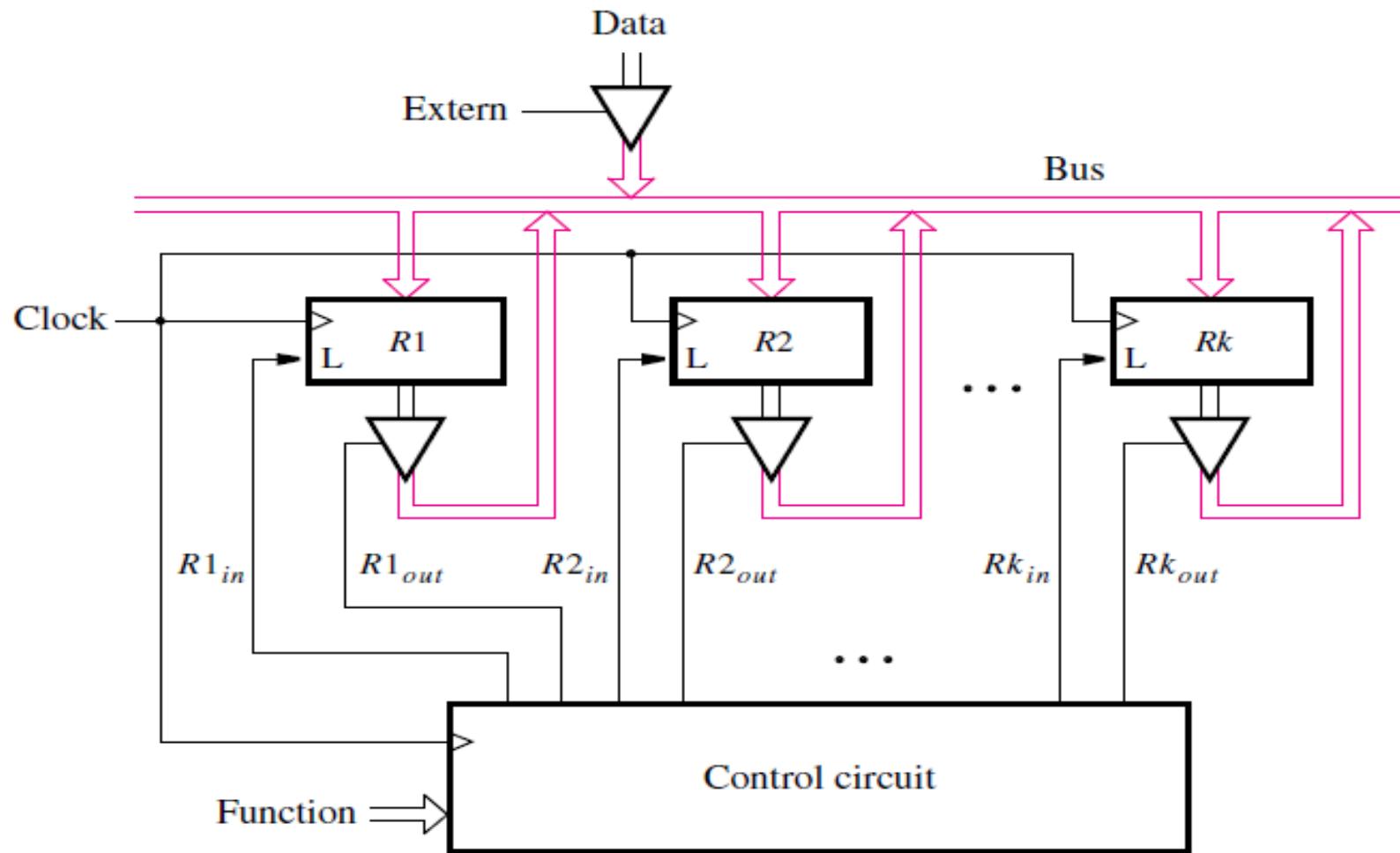
- A simple way of providing the desired interconnectivity may be to connect each register to a common set of  $n$  wires, which are used to transfer data into and out of the registers. This common set of wires is usually called a *bus*.
- If common paths are used to transfer data from multiple sources to multiple destinations, it is necessary to ensure that only one register acts as a source at any given time and other registers do not interfere.

# Using Tri-State Drivers to Implement a Bus

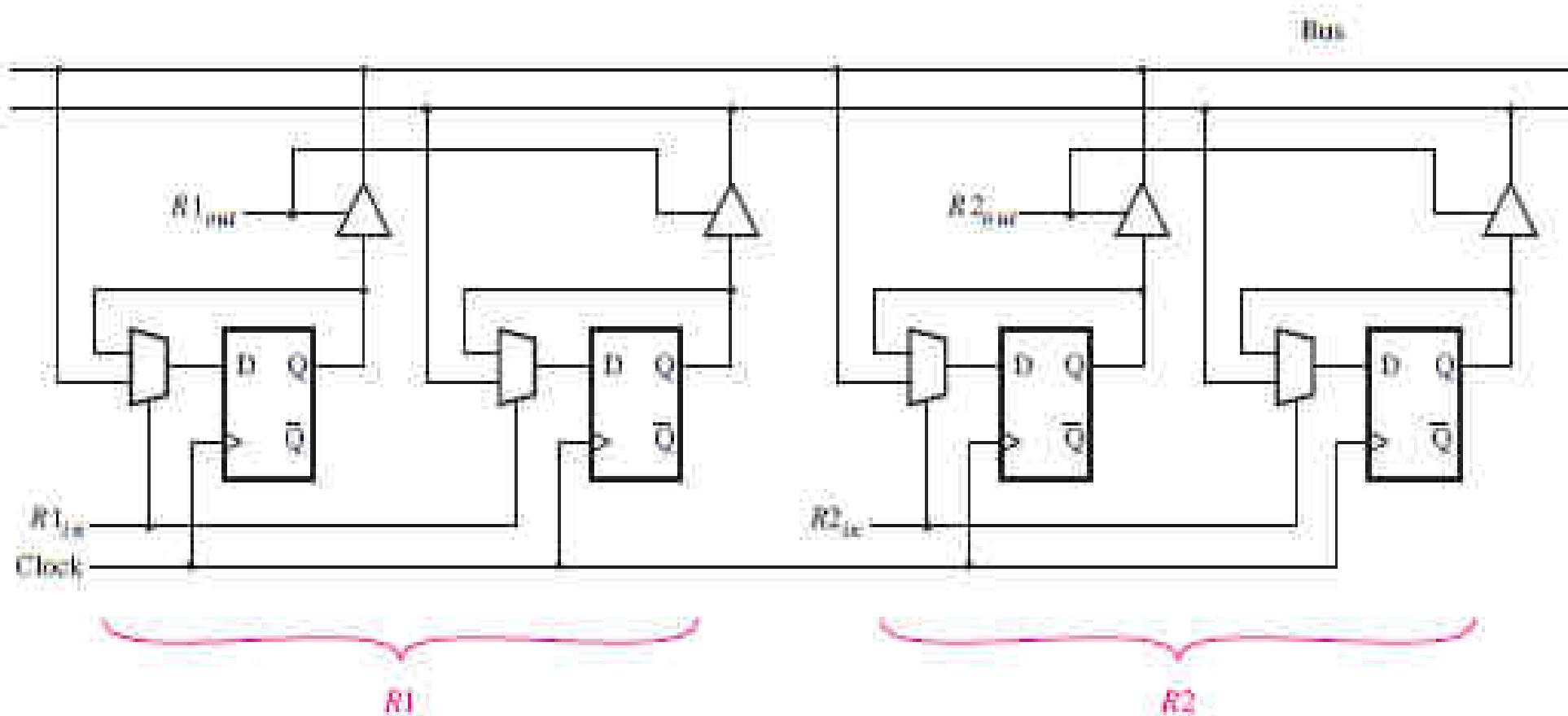
Outputs of two ordinary logic gates cannot be connected together, because a short circuit would result if one gate forces the output value 1 while the other gate forces the value 0. Therefore, some special gates are needed if the outputs of two registers are to be connected to a common set of wires. A commonly used circuit element for this purpose is a *tri-state* driver (or buffer). It performs no logic operation and its output simply replicates the input signal. Its purpose is to provide additional electrical driving capability.



**Figure 7.1** Tri-state driver.

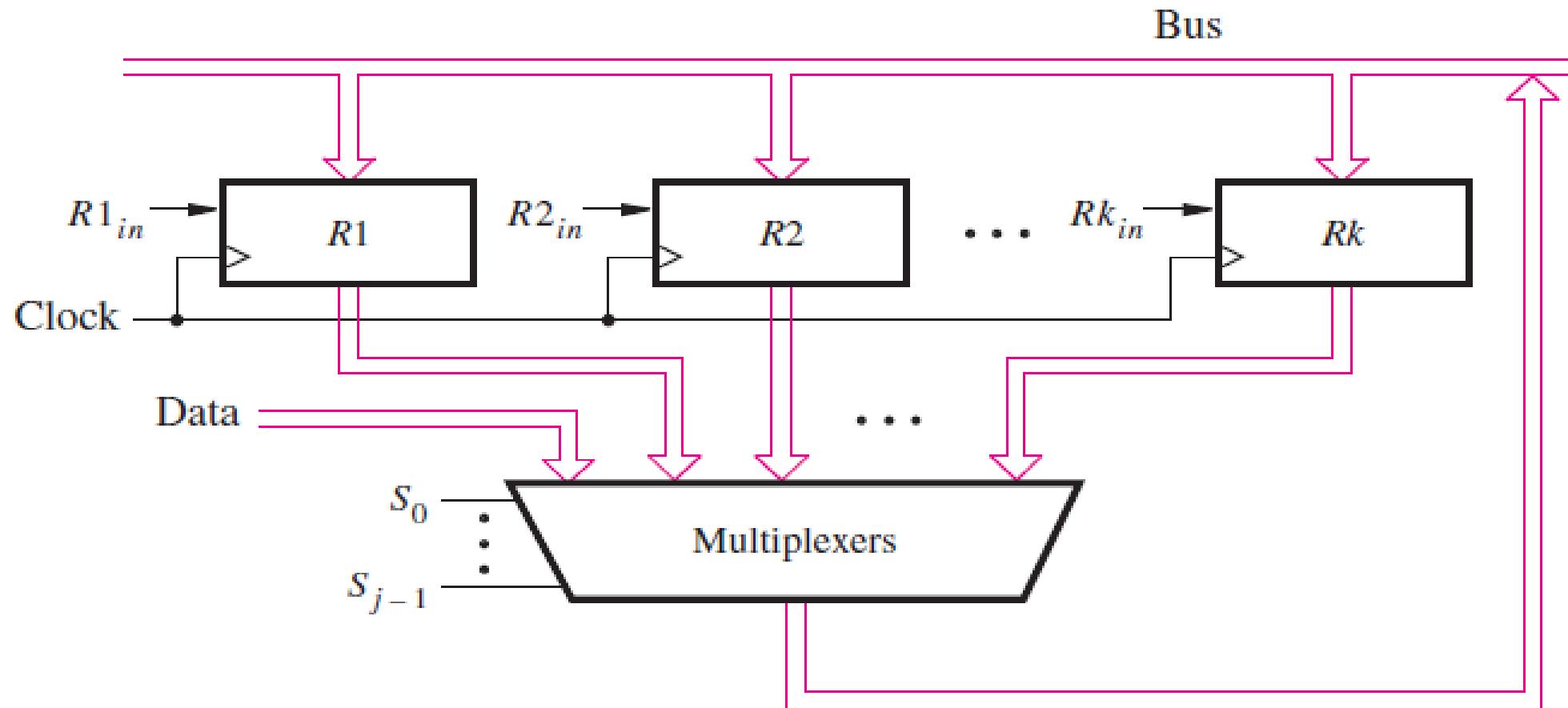


**Figure 7.2** A digital system with  $k$  registers.



**Figure 7.3** Details for connecting registers to bus

## Using Multiplexers to Implement a Bus



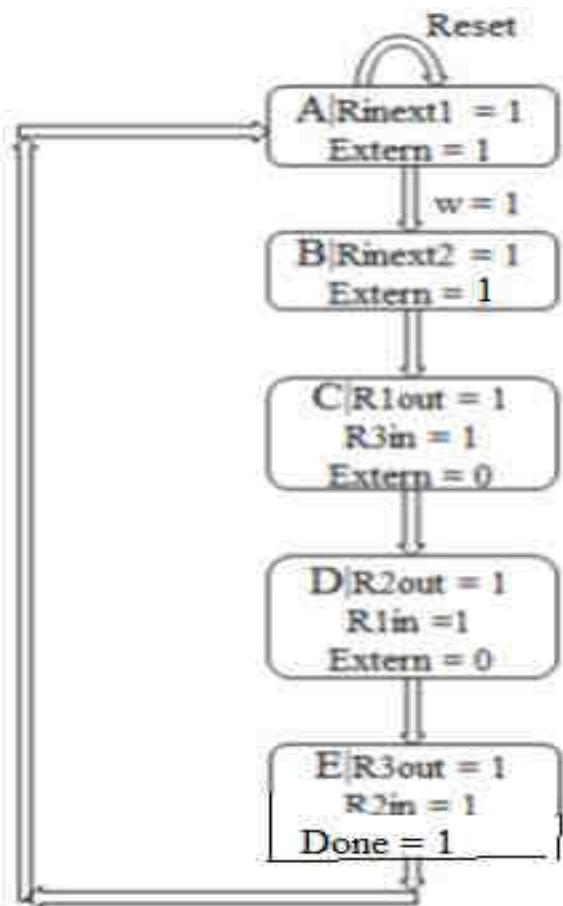
**Figure 7.4** Using multiplexers to implement a bus.

## **Verilog Code for Specification of Bus Structures (Register swap through bus using tristate drivers)**

As a simple example, consider a system that has three registers, R1, R2, and R3. We will specify a control circuit that performs a single function—it swaps the contents of registers R1 and R2, using R3 for temporary storage.

The required swapping is done in three steps, each needing one clock cycle. In the first step the contents of R2 are transferred into R3. Then the contents of R1 are transferred into R2. Finally, the contents of R3, which are the original contents of R2, are transferred into R1.

To transfer the contents of one register into another buses are used. The control circuit for this task can be explained in the form of a finite state machine as shown in Figure 12.2. Its state table is shown in Table 12.1



- In state A and state B,  $\text{Extern} = 1$  to load two different data to the bus and then from the bus to the registers R1 and R2.
- The states B, C and D change their states if  $\text{Extern} = 0$ .
- From B, it goes to C state where R1 is copied to R3.
- From C it goes to state D where R2 is copied to R1.
- From D, the next state is E to copy R3 to R2 and an output  $\text{Done} = 1$  to indicate that swap is completed.
- From E the next state is the initial state A. All transfers are taking place through the bus.

- We assume that three control signals named  $RinExt1$ ,  $RinExt2$ , and  $RinExt3$  exist, which allow the externally supplied data to be loaded from the bus into register  $R1$ ,  $R2$ , or  $R3$ .
- When  $RinExt1 = 1$ , the data on the bus is loaded into register  $R1$ ; when  $RinExt2 = 1$ , the data is loaded into  $R2$ ; and when  $RinExt3 = 1$ , the data is loaded into  $R3$ .
- We have assumed that an input signal named  $w$  exists, which is set to 1 for one clock cycle to start the swapping task.
- We have also assumed that at the end of the swapping task, which is indicated by the  $Done$  signal being asserted, the control circuit returns to the starting state.

PS	NS		Outputs								
	Extern=1	Extern=0	Rinext1	Rinext2	R1in	R1out	R2in	R2out	R3in	R3out	done
A	B	A	1	0	0	0	0	0	0	0	0
B	B	C	0	1	0	0	0	0	0	0	0
C	C	D	0	0	0	1	0	0	1	0	0
D	D	E	0	0	1	0	0	1	0	0	0
E	A	A	0	0	0	0	1	0	0	1	1

```
module regn (R, L, Clock, Q);
parameter n = 8;
input [n-1:0] R;
input L, Clock;
output [n-1:0] Q;
reg [n-1:0] Q;
always @ (posedge Clock)
if (L)
Q <= R;
endmodule // Code for 8-bit
register
```

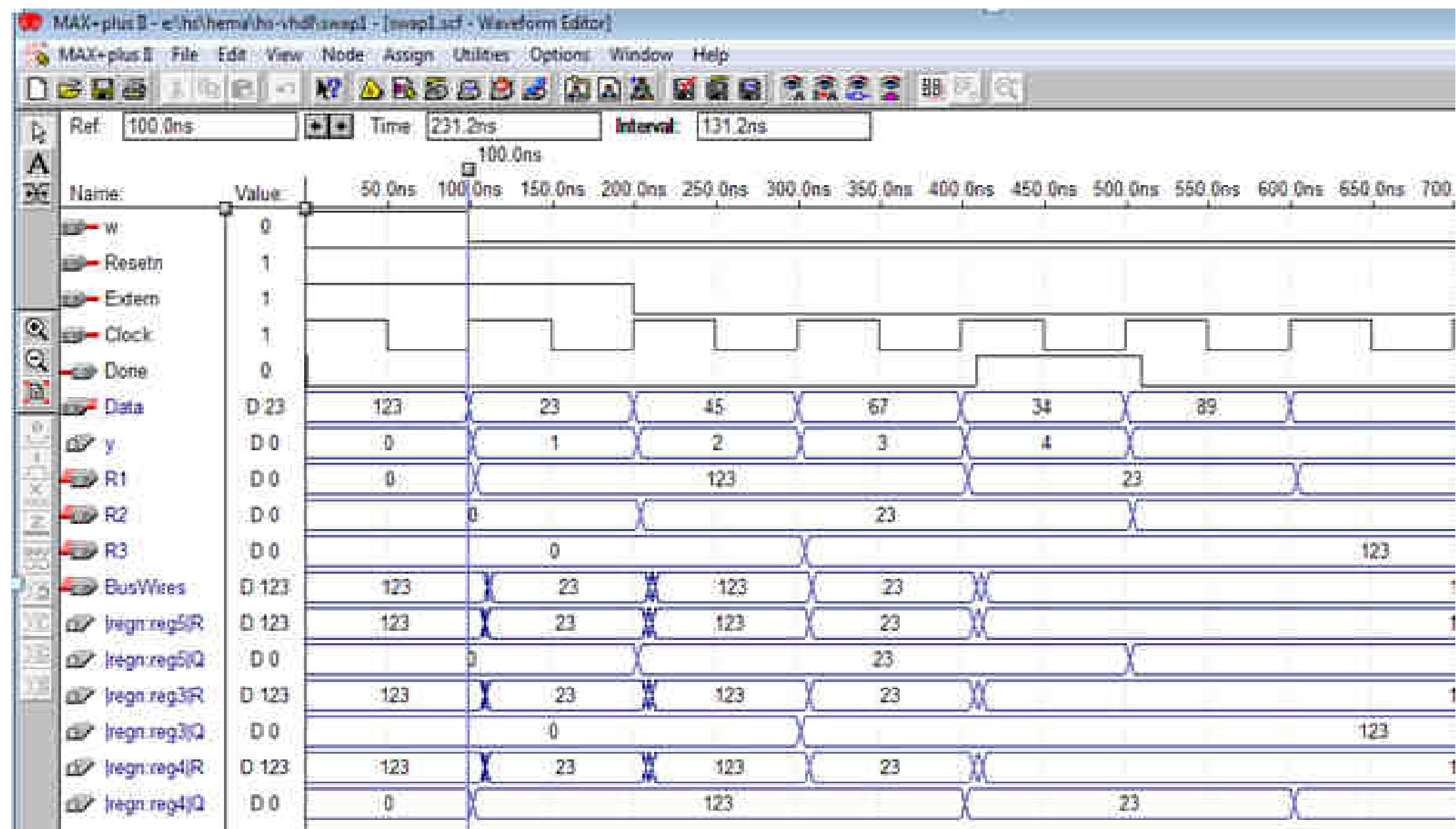
```
module swap1 (Resetn, Clock, w, Data, Extern, R1, R2, R3, BusWires, Done);
parameter n = 8;
input Resetn, Clock, w, Extern;
input [n-1:0] Data;
output [n-1:0] BusWires ,R1, R2, R3 ;
reg [n-1:0] BusWires, R1, R2, R3;
output Done;
wire R1in, R1out, R2in, R2out, R3in, R3out, RinExt1, RinExt2;
reg [2:0] y, Y;
parameter [2:0] A = 3'b000, B = 3'b001, C = 3'b010, D = 3'b011, E = 3'b100;
// Define the next state combinational circuit for FSM
always @(w or y)
begin
case (y)
A: if (w) Y = B;
else Y = A;
B: Y = C;
C: Y = D;
D: Y = E;
E: Y = A;
endcase
end
```

```

// Define the sequential block for FSM
always @(negedge Resetn or posedge Clock)
begin
if (Resetn == 0) y <= A;
else y <= Y;
end
// Define outputs of FSM
assign RinExt1 = (y == A);
assign RinExt2 = (y == B);
assign R3in = (y == C);
assign R1out = (y == C);
assign R2out = (y == D);
assign R1in = (y == D);
assign R3out = (y == E);
assign R2in = (y == E);
assign Done = (y == E);
always @((Extern or R1out or R2out or R3out))
if (Extern) BusWires = Data;
else if (R1out) BusWires = R1;
else if (R2out) BusWires = R2;
else if (R3out) BusWires = R3;
regn reg3 (BusWires, R3in, Clock, R3);
regn reg4 (BusWires, RinExt1 | R1in, Clock, R1);
regn reg5 (BusWires, RinExt2 | R2in, Clock, R2);
endmodule

```

PS	NS		Outputs									
	Exter n=1	Exter n=0	Rine xt1	Rine xt2	R1 in	R1 out	R2 in	R2o ut	R3 in	R3 out	do ne	
A	B	A	1	0	0	0	0	0	0	0	0	
B	B	C	0	1	0	0	0	0	0	0	0	
C	C	D	0	0	0	1	0	0	1	0	0	
D	D	E	0	0	1	0	0	1	0	0	0	
E	A	A	0	0	0	0	1	0	0	1	1	



## Verilog Code Using Multiplexers

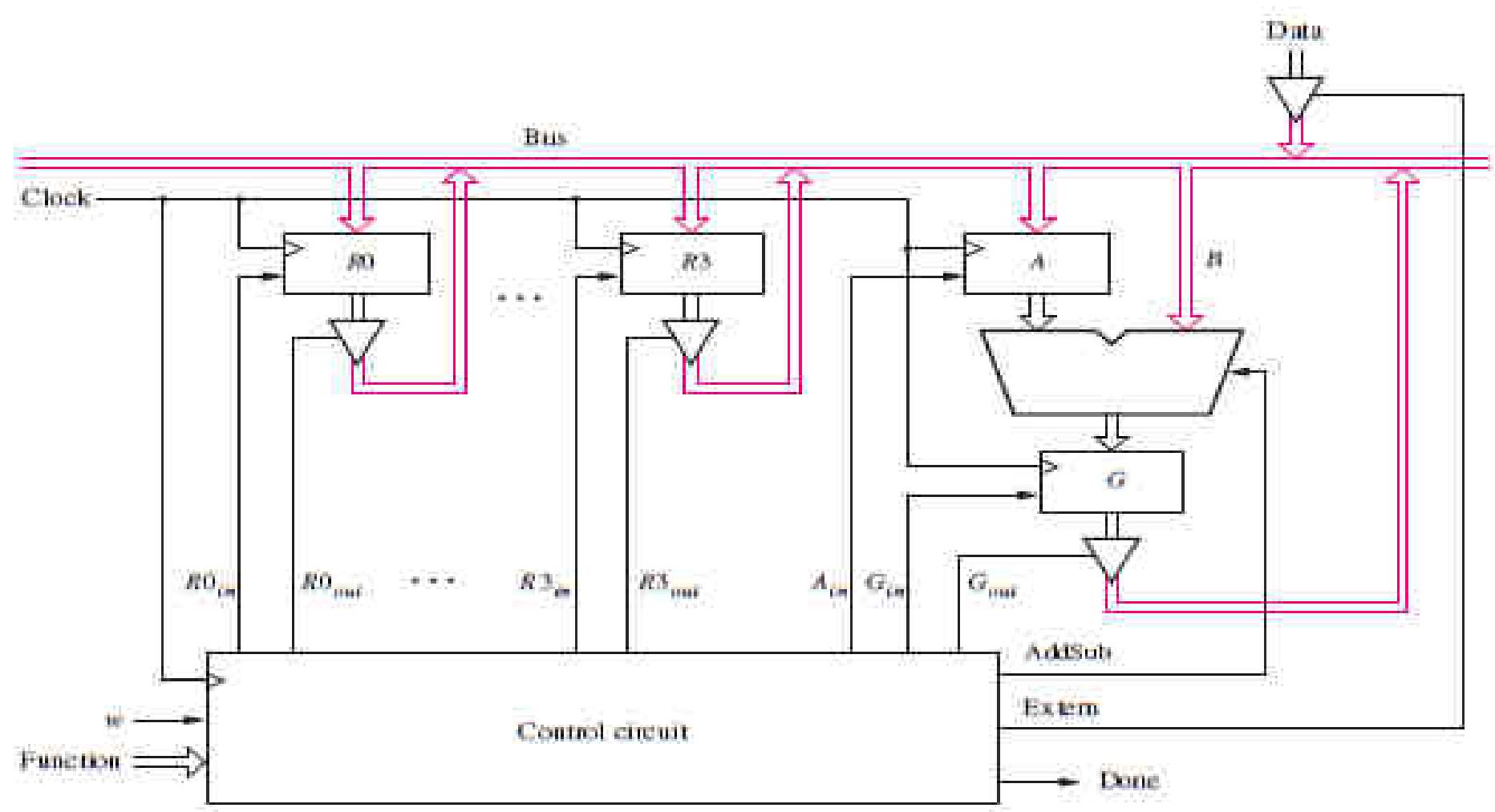
- Using the circuit structure shown in Figure 7.4, the bus is implemented with eight 4-to-1 multiplexers. Three of the data inputs on each 4-to-1 multiplexer are connected to one bit from registers  $R1$ ,  $R2$ , and  $R3$ . The fourth data input is connected to one bit of the  $Data$  input signal to allow externally supplied data to be written into the registers.
- The same FSM control circuit is used. However, the control signals  $R1out$ ,  $R2out$ , and  $R3out$  are not needed because tri-state drivers are not used. Instead, the required multiplexers are defined in an **if-else** statement by specifying the source of data based on the state of the FSM. Hence, when the FSM is in state  $A$ , the selected input to the multiplexers is  $Data$ . When the state is  $B$ , the register  $R2$  provides the input data to the multiplexers, and so on

```
module swapmux (Resetn, Clock, w, Data, RinExt1, RinExt2, RinExt3, BusWires,  
Done);  
parameter n = 8;  
input Resetn, Clock, w, RinExt1, RinExt2, RinExt3;  
input [n-1:0] Data;  
output [n-1:0] BusWires;  
reg [n-1:0] BusWires;  
output Done;  
wire [n-1:0] R1, R2, R3;  
wire R1in, R2in, R3in;  
reg [2:1] y, Y;  
parameter [2:1] A = 2'b00, B = 2'b01, C = 2'b10, D = 2'b11;  
// Define the next state combinational circuit for FSM  
always @(w or y)  
case (y)  
A: if (w) Y = B;  
else Y = A;  
B: Y = C;  
C: Y = D;  
D: Y = A;  
endcase
```

```
always @(negedge Resetn or posedge Clock)
if (Resetn == 0) y<=A;
else y<=Y;
// Define control signals
assign R3in = (y == B);
assign R2in = (y == C);
assign R1in = (y == D);
assign Done = (y == D);
// Instantiate registers
regn reg1 (BusWires, RinExt1 | R1in, Clock, R1);
regn reg2 (BusWires, RinExt2 | R2in, Clock, R2);
regn reg3 (BusWires, RinExt3 | R3in, Clock, R3);
// Define the multiplexers
always @(y or Data or R1 or R2 or R3)
if (y == A) BusWires = Data;
else if (y == B) BusWires = R2;
else if (y == C) BusWires = R1;
else BusWires = R3;
endmodule
```

```
module regn(R, L, Clock, Q);
parameter n = 8;
input [n-1:0] R;
input L, Clock;
output[n-1:0] Q;
reg [n-1:0] Q;
always @(posedge Clock)
if (L)
Q <= R;
endmodule
```

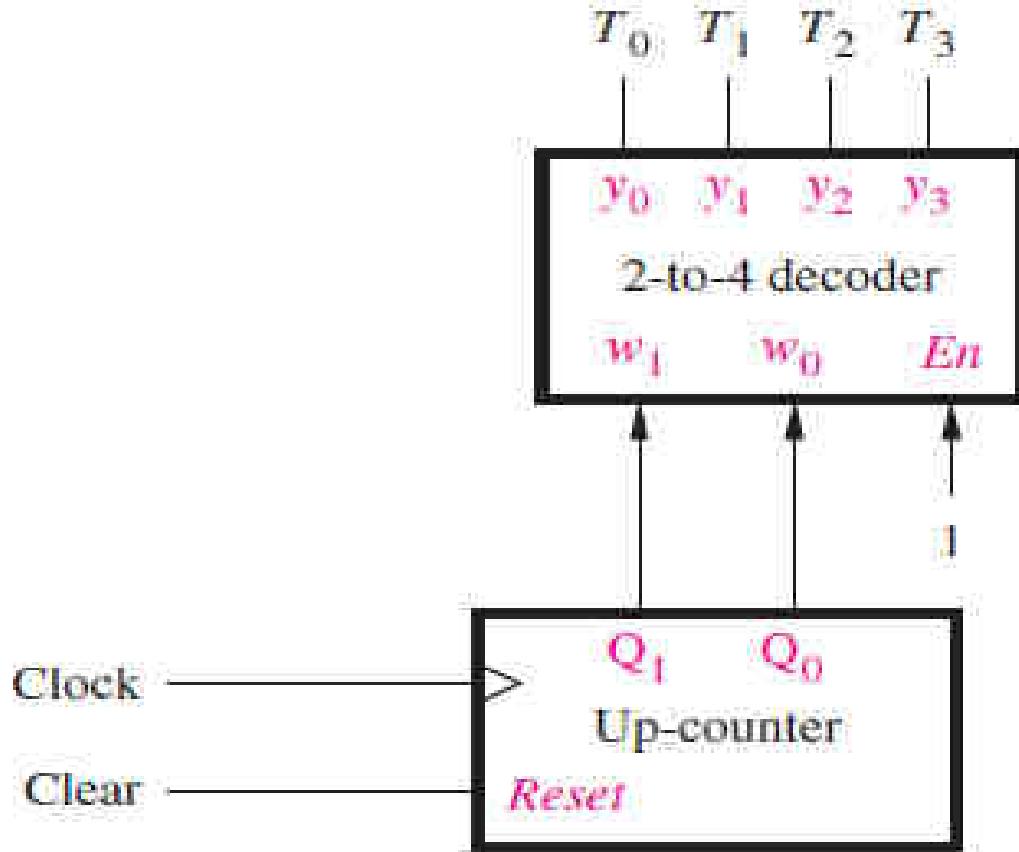
# Simple Processor



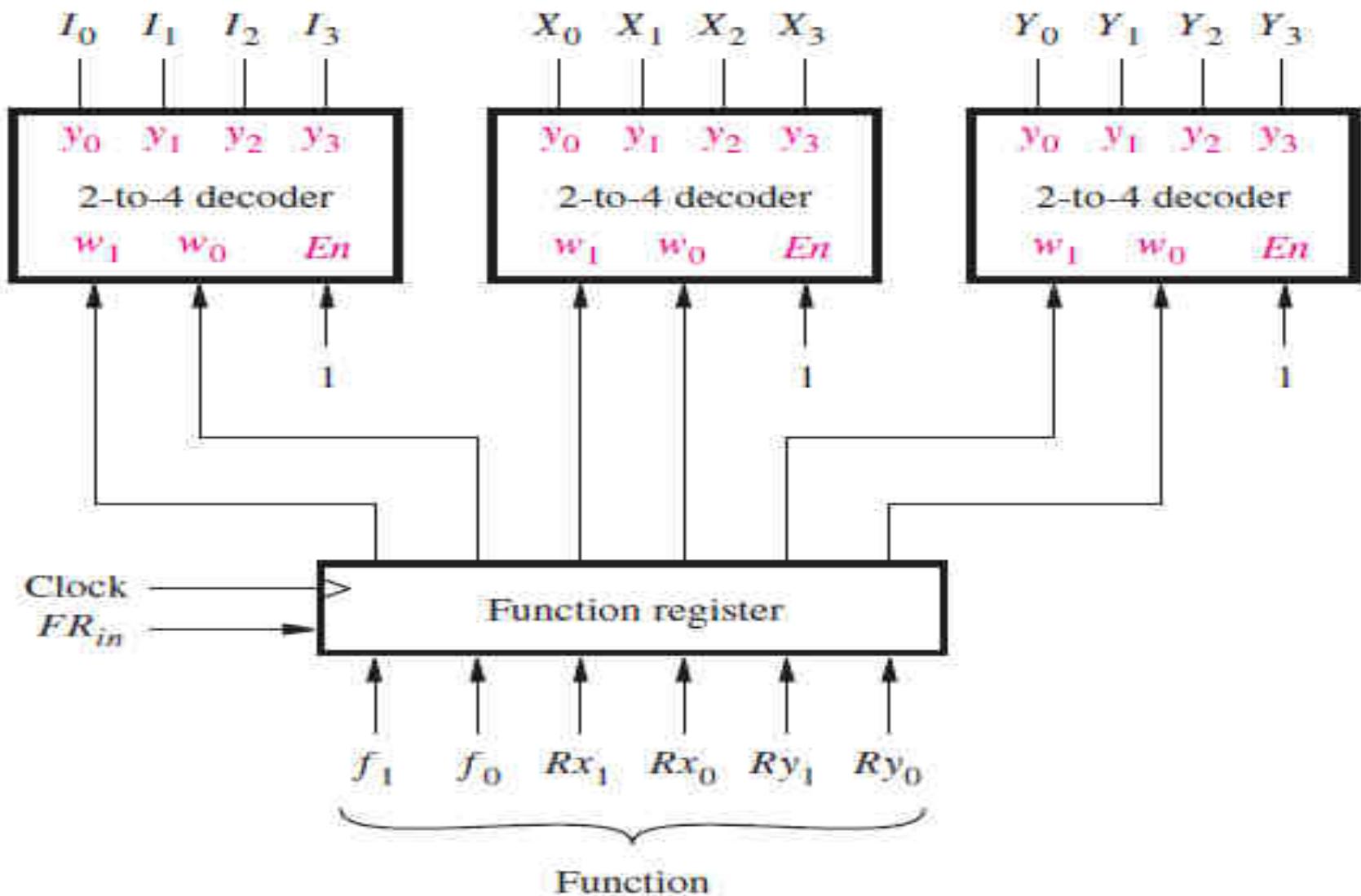
**Figure 7.9** A digital system that implements a simple processor.

**Table 7.11** Operations performed by the processor.

Operations	Function performed
Load Rx, Data	$Rx \leftarrow Data$
Move Rx, Ry	$Rx \leftarrow [Ry]$
Add Rx, Ry	$Rx \leftarrow [Rx] + [Ry]$
Sub Rx, Ry	$Rx \leftarrow [Rx] - [Ry]$



**Figure 7.10** A part of the control circuit for the processor.



**Figure 7.11** The function register and decoders.

**Table 7.2**

Control signals asserted in each operation/time step.

	$T_1$	$T_2$	$T_3$
(Load): $I_0$	$Extern, R_{in} = X,$ <i>Done</i>		
(Move): $I_1$	$R_{in} = X, R_{out} = Y,$ <i>Done</i>		
(Add): $I_2$	$R_{out} = X, A_{in}$	$R_{out} = Y, G_{in},$ $AddSub = 0$	$G_{out}, R_{in} = X,$ <i>Done</i>
(Sub): $I_3$	$R_{out} = X, A_{in}$	$R_{out} = Y, G_{in},$ $AddSub = 1$	$G_{out}, R_{in} = X,$ <i>Done</i>

$$Clear = w' T_0 + Done$$
$$FRin = w T_0$$
$$Extern = I_0 T_1$$
$$Done = (I_0 + I_1) T_1 + (I_2 + I_3) T_3$$
$$Ain = (I_2 + I_3) T_1$$
$$Gin = (I_2 + I_3) T_2$$
$$Gout = (I_2 + I_3) T_3$$
$$AddSub = I_3$$

The values of  $R0in, \dots, R3in$  are determined using either the  $X0, \dots, X3$  signals or the  $Y0, \dots, Y3$  signals. In Table 7.2 these actions are indicated by writing either  $Rin = X$  or  $Rin = Y$ . The meaning of  $Rin = X$  is that  $R0in = X0, R1in = X1$ , and so on. Similarly, the values of  $R0out, \dots, R3out$  are specified using either  $Rout = X$  or  $Rout = Y$ .

The table shows that  $R0in$  is set to the value of  $X0$  in the first step of both the *Load* and *Move* operations and in the third step of both the *Add* and *Sub* operations, which leads to the expression

$$R0in = (I0 + I1)T1X0 + (I2 + I3)T3X0.$$

Similarly,  $R0out$  is set to the value of  $Y0$  in the first step of *Move*. It is set to  $X0$  in the first step of *Add* and *Sub* and to  $Y0$  in the second step of these operations, which gives

$$R0out = I1T1Y0 + (I2 + I3)(T1X0 + T2Y0).$$

The expressions for  $R1in$  and  $R1out$  are the same as those for  $R0in$  and  $R0out$  except that  $X1$  and  $Y1$  are used in place of  $X0$  and  $Y0$ . The expressions for  $R2in$ ,  $R2out$ ,  $R3in$ , and  $R3out$  are derived in the same way.

```
module proc_mux(Data, Reset, w, Clock, F, Rx, Ry,R0, R1, R2, R3,
Count, l, BusWires);
input [3:0] Data;
input Reset, w, Clock;
input [1:0] F, Rx, Ry;
output [1:0] Count, l;
output [3:0] BusWires, R0, R1, R2, R3;
reg [3:0] BusWires;
reg [3:0] Sum;
reg [0:3] Rin, Rout;
reg Extern, Ain, Gin, Gout, AddSub;
//wire [1:0] Count, l;
wire [0:3] Xreg, Y;
wire [3:0] A, G;
wire [1:6] Func, FuncReg, Sel;
```

```
upcount1 counter (Reset, Clock, Count);
assign Func = {F, Rx, Ry};
wire FRin = w & ~Count[1] & ~Count[0];
regn3 functionreg (Func, FRin, Clock, FuncReg);
//defparam functionreg.n = 6;
assign I = FuncReg[1:2];
dec2to4 decX (FuncReg[3:4], 1'b1, Xreg);
dec2to4 decY (FuncReg[5:6], 1'b1, Y);
```

```
always @(Count or I or Xreg or Y)
begin
Extern = 1'b0; Ain = 1'b0; Gin = 1'b0;
Gout = 1'b0; AddSub = 1'b0; Rin = 4'b0; Rout = 4'b0;
case (Count)
2'b00: ; //no signals asserted in time step T0
2'b01: //define signals in time step T1
case (I)
2'b00: begin //Load
Extern = 1'b1; Rin = Xreg;
end
2'b01: begin //Move
Rout = Y; Rin = Xreg;
end
default: begin //Add, Sub
Rout = Xreg; Ain = 1'b1;
end
endcase
```

```
2'b10: //define signals in time step T2
case(l)
 2'b10: begin //Add
   Rout = Y; Gin = 1'b1;
 end
 2'b11: begin //Sub
   Rout = Y; AddSub = 1'b1; Gin = 1'b1;
 end
 default: ; //Add, Sub
endcase
2'b11:
case (l)
 2'b10, 2'b11:
begin
 Gout = 1'b1; Rin = Xreg;
end
default: ; //Add, Sub
endcase
endcase
end
```

```
regn2 reg_0 (BusWires, Rin[0], Clock, R0);
regn2 reg_1 (BusWires, Rin[1], Clock, R1);
regn2 reg_2 (BusWires, Rin[2], Clock, R2);
regn2 reg_3 (BusWires, Rin[3], Clock, R3);
regn2 reg_A (BusWires, Ain, Clock, A);
// alu
always @(AddSub or A or BusWires)
begin
if (!AddSub)
Sum = A + BusWires;
else
Sum = A - BusWires;
end
regn2 reg_G (Sum, Gin, Clock, G);
assign Sel = {Rout, Gout, Extern};
```

```
always @(Sel or R0 or R1 or R2 or R3 or  
G or Data)
```

```
begin
```

```
if (Sel == 6'b100000)
```

```
BusWires = R0;
```

```
else if (Sel == 6'b010000)
```

```
BusWires = R1;
```

```
else if (Sel == 6'b001000)
```

```
BusWires = R2;
```

```
else if (Sel == 6'b000100)
```

```
BusWires = R3;
```

```
else if (Sel == 6'b000010)
```

```
BusWires = G;
```

```
else BusWires = Data;
```

```
end
```

```
endmodule
```

```
module upcount1(Clear, Clock, Q);
input Clear, Clock;
output [1:0] Q;
reg [1:0] Q;
always @(posedge Clock)
if (Clear)
Q <= 0;
else
Q <= Q + 1;
endmodule

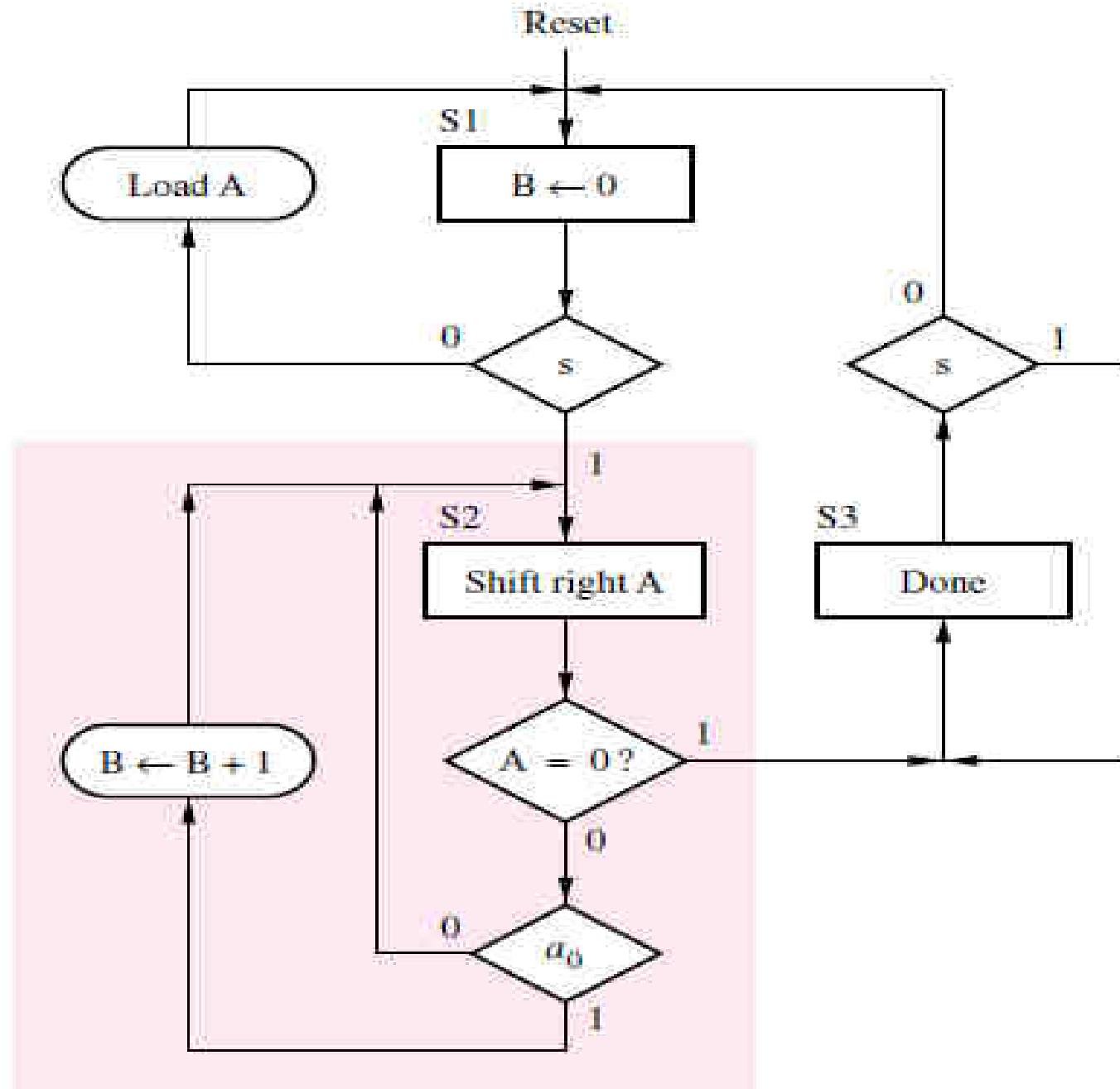
module regn2(R, L, Clock, Q);
parameter n = 4;
input [n-1:0] R;
input L, Clock;
output[n-1:0] Q;
reg [n-1:0] Q;
always @(posedge Clock)
if (L)
Q <= R;
endmodule
```

```
module dec2to4 (W, En, Y);
input [1:0]W;
input En;
output [0:3] Y;
reg [0:3] Y;
always @(W or En)
case ({En,W})
3'b100: Y = 4'b1000;
3'b101: Y = 4'b0100;
3'b110: Y = 4'b0010;
3'b111: Y = 4'b0001;
default: Y = 4'b0000;
endcase
endmodule
module regn3(R, L, Clock, Q);
parameter n = 6;
input [n-1:0] R;
input L, Clock;
output[n-1:0] Q;
reg [n-1:0] Q;
always @(posedge Clock)
if (L)
Q <= R;
endmodule
```

# A Bit-Counting Circuit

To count the number of bits in a register,  $A$ , that have the value 1.

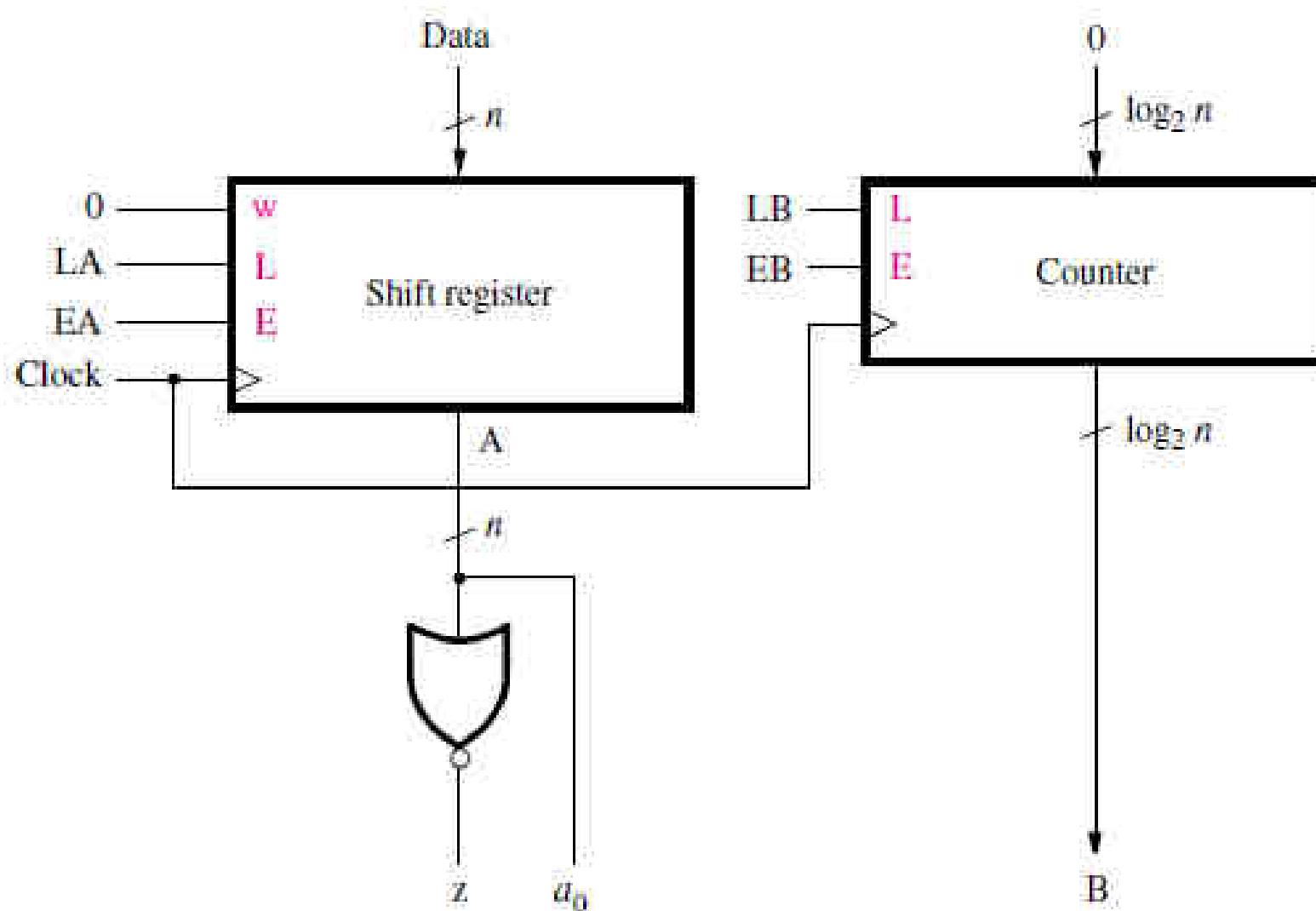
```
B = 0;  
while A ≠ 0 do  
    if  $a_0 = 1$  then  
        B = B + 1;  
    end if;  
    Right-shift A;  
end while;
```



- In a traditional flowchart, when state  $S2$  is entered, the value of  $A$  would first be shifted to the right. Then we would examine the value of  $A$  and if  $A$ 's LSB is 1, we would immediately add 1 to  $B$ .
- But, since the ASM chart represents a sequential circuit, changes in  $A$  and  $B$ , which represent the outputs of flip-flops, take place after the active clock edge. The same clock signal that controls changes in the state of the machine also controls changes in  $A$  and  $B$ .

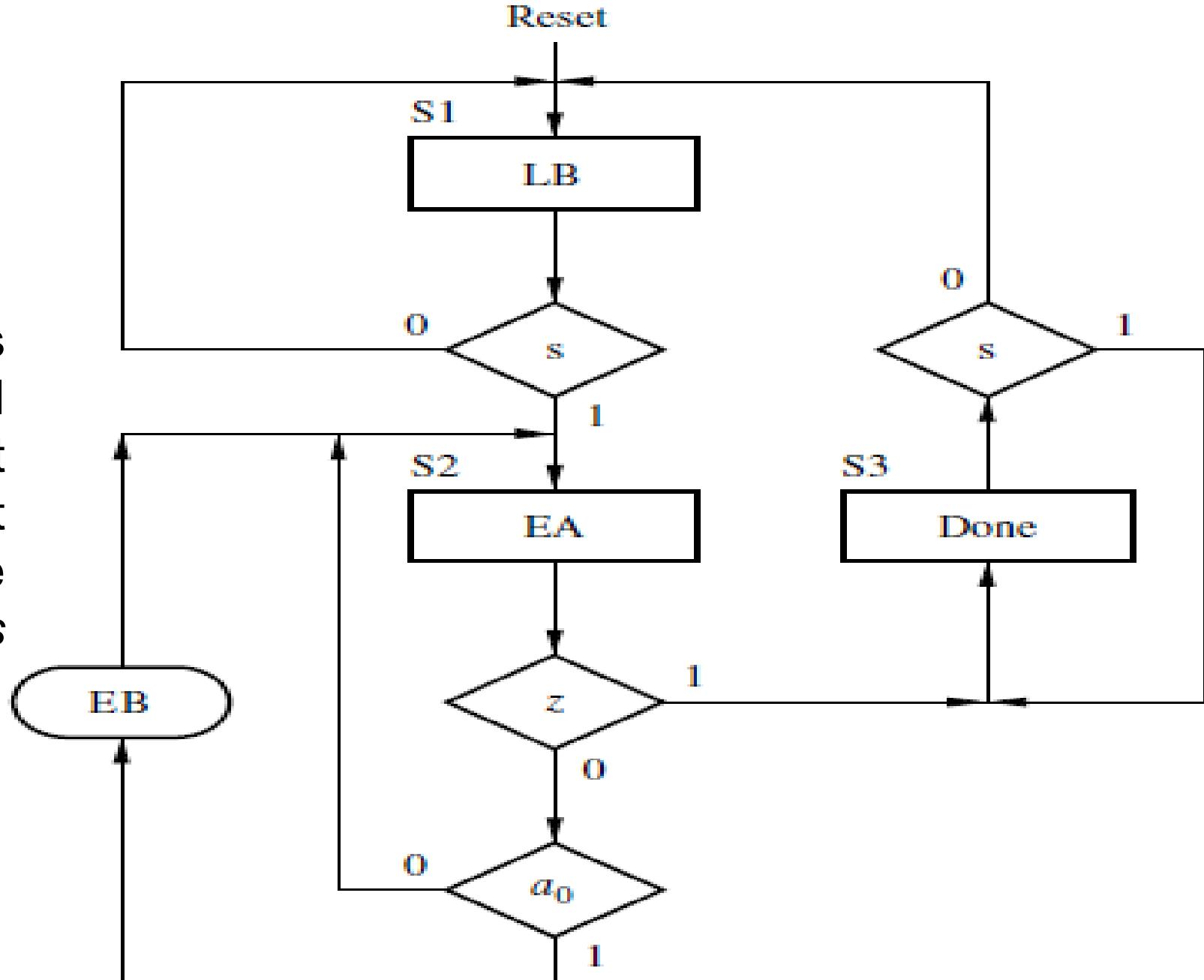
- When the machine is in state  $S1$ , the next active clock edge will only perform the action specified inside the state box for  $S1$ , which is  $B \leftarrow 0$ . Hence in state  $S2$ , the decision box that tests whether  $A = 0$ , as well as the box that checks the value of  $a0$ , check the bits in  $A$  before they are shifted.
- If  $A = 0$ , then the FSM will change to state  $S3$  on the next clock edge (this clock edge also shifts  $A$ , which has no effect because  $A$  is already 0 in this case.)
- On the other hand, if  $A \neq 0$ , then the FSM does not change to  $S3$ , but remains in  $S2$ . At the same time,  $A$  is still shifted, and  $B$  is incremented if  $a0$  has the value 1.

# Datapath Circuit



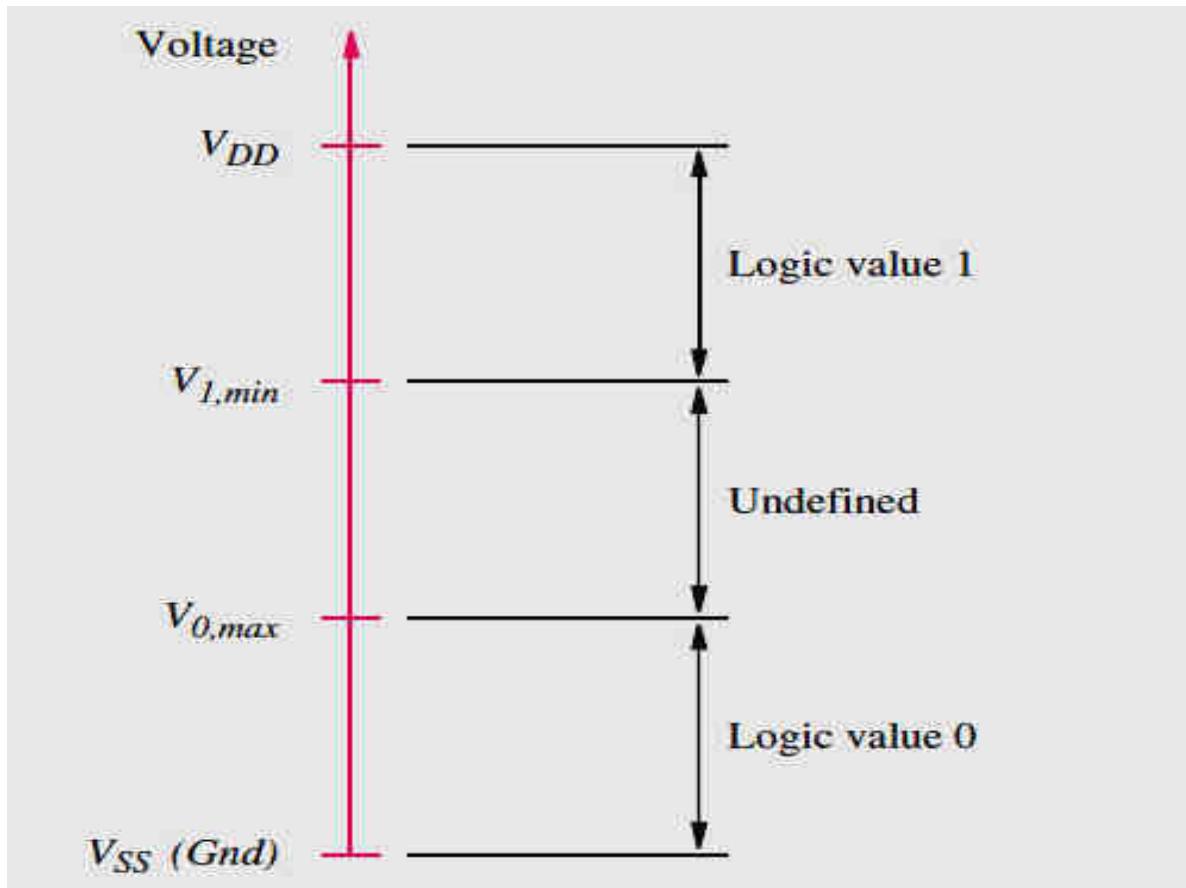
## Control Circuit

We assume that external circuitry drives  $LA$  to 1 when valid data is present at the parallel inputs of the shift register, so that the shift register contents are initialized before changes to 1.



# Switching Circuits

# Representation of logic values by voltage levels



Minimum voltage is  $V_{SS}$  (Gnd), maximum voltage is  $V_{DD}$  (supply). The voltages in the range Gnd to  $V_{0,max}$  represent logic value 0.  $V_{0,max}$  is the maximum voltage level that a logic circuit must recognize as low. Similarly, the range from  $V_{L,min}$  to  $V_{DD}$  corresponds to logic value 1, and  $V_{L,min}$  is the minimum voltage level that a logic circuit must interpret as high.

# Transistor Switches

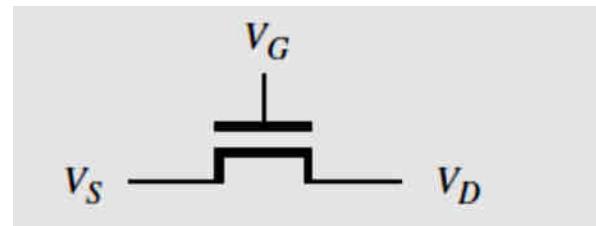
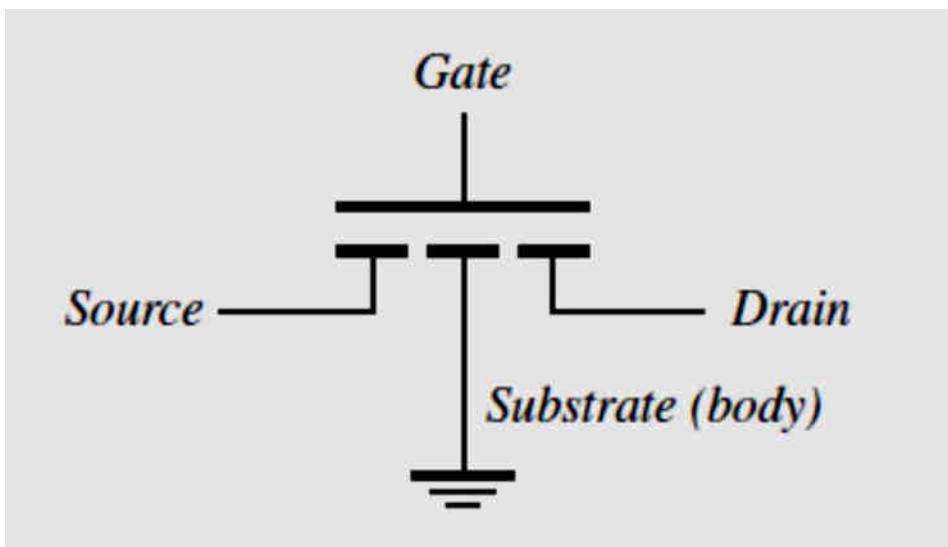
- Logic circuits are built with transistors.
- The most popular type of transistor for implementing the **switch** is the metal oxide semiconductor field-effect transistor (**MOSFET**).
- Two different types of MOSFET are:
  - n-channel (NMOS)
  - p-channel (PMOS)

# NMOS Transistor

$x = \text{"low"}$



$x = \text{"high"}$



Simplified symbol for an NMOS Trnsistor

NMOS transistor has four electrical terminals, called the source, drain, gate, and substrate. In logic circuits the substrate (also called body) terminal is connected to Gnd. So the simplified graphical symbol which omits the substrate node is used. The source and drain terminals are distinguished in practice by the voltage levels applied to the transistor; by convention, the terminal with the lower voltage level is deemed to be the source.

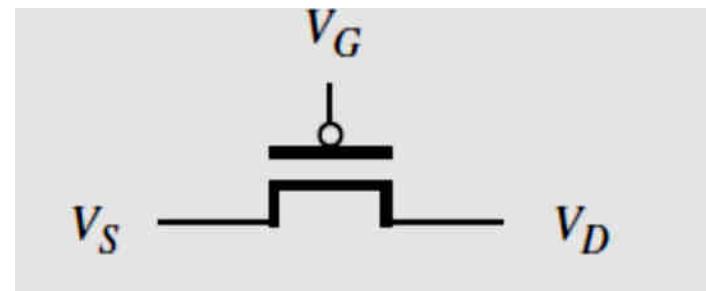
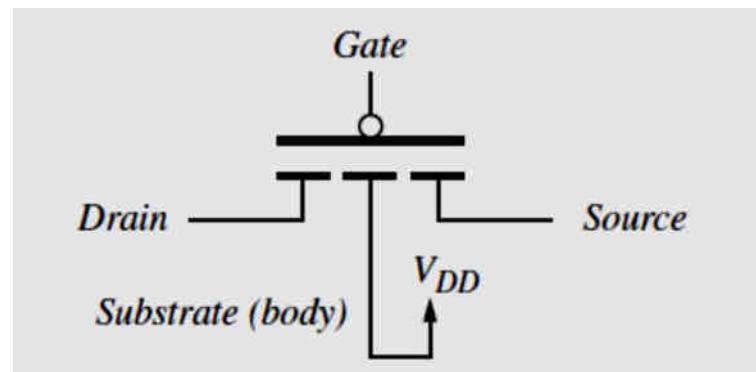
- If **V<sub>g</sub>** is **low**, then there is no connection between the source and drain, transistor is **turned off**.
- If **V<sub>g</sub>** is **high** then the transistor is **turned on** and act as a closed switch that connects source and drain terminals.

# PMOS Transistor

$x = \text{"high"}$

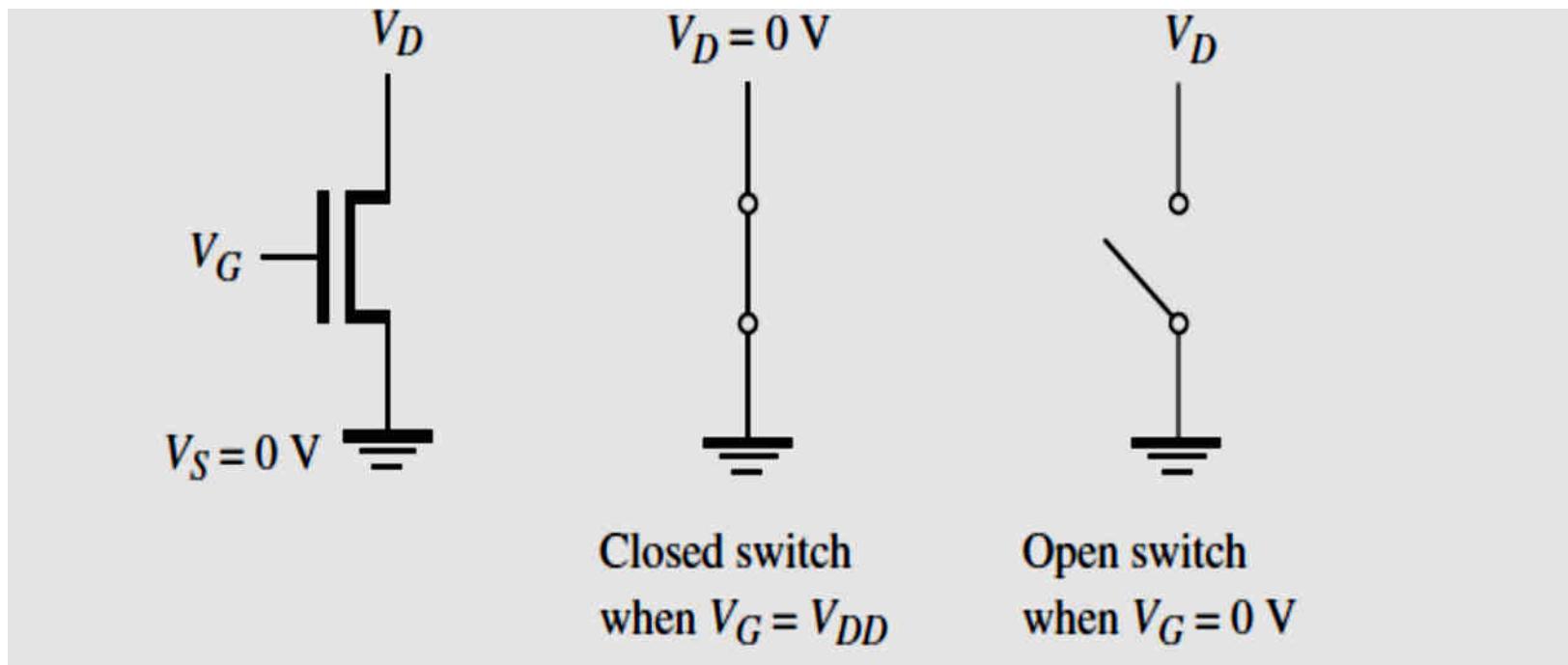


$x = \text{"low"}$

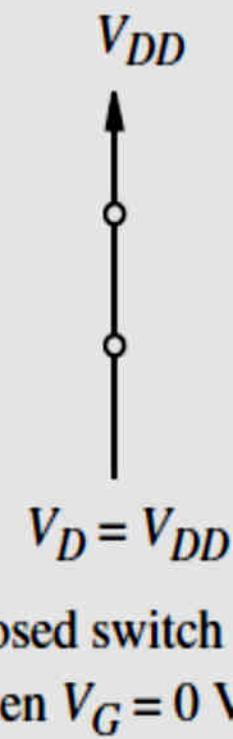
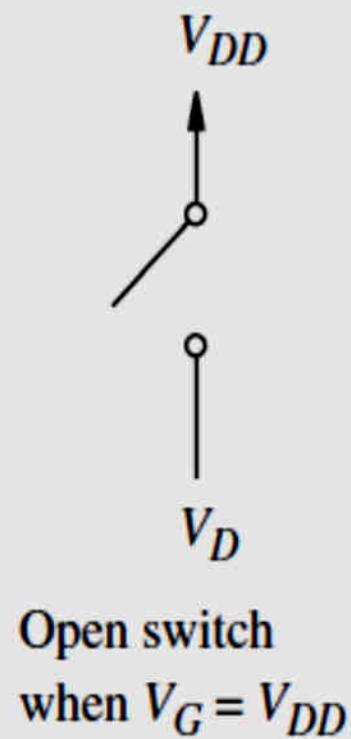
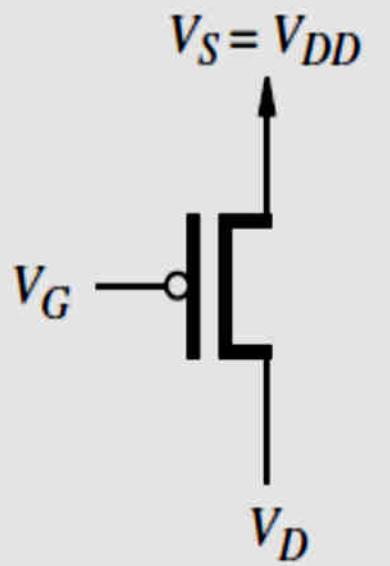


- In logic circuits, the substrate of the PMOS transistor is always connected to VDD.
- If **V<sub>g</sub> is high**, then the transistor is **turned off** and acts like a open switch.
- If **V<sub>g</sub> is low**, then the transistor is **turned on** and act as a closed switch that connects source and drain terminals.
- In the PMOS transistor the source is the node with the higher voltage.

- When the NMOS Transistor is turned on, its drain is pulled down to Gnd.



When the PMOS Transistor is turned on, its drain is pulled up to  $V_{DD}$ .



Open switch  
when  $V_G = V_{DD}$

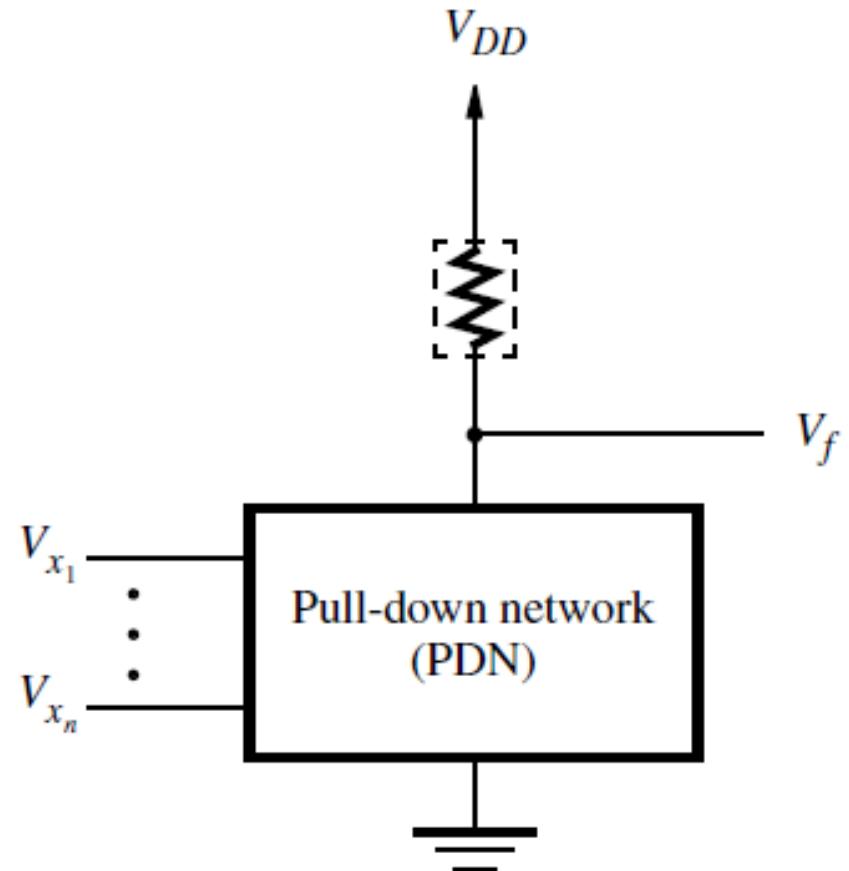
Closed switch  
when  $V_G = 0 \text{ V}$

# CMOS Logic Gates

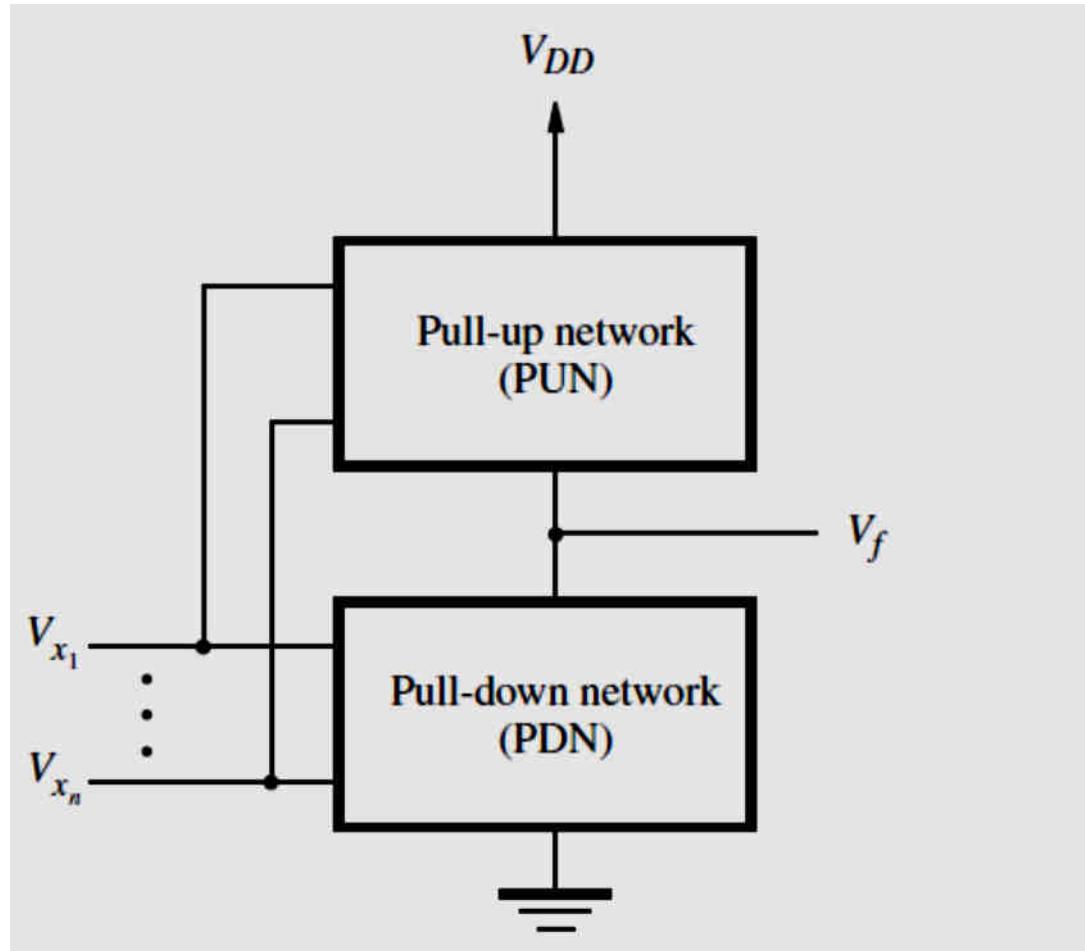
- The most popular approach that uses **both NMOS and PMOS transistors** together is the **CMOS** Technology.
- In NMOS circuits the logic functions are realized by arrangements of NMOS transistors, combined with a pull-up device that acts as a resistor.

- The part of the circuit that involves **NMOS transistors** is called the pull-down network(**PDN**).
- In CMOS, pull-up device is replaced with a pull-up network (**PUN**) that is built using PMOS transistors, such that functions realized by the PDN and PUN networks are complements of each other.

# Structure of a NMOS circuit

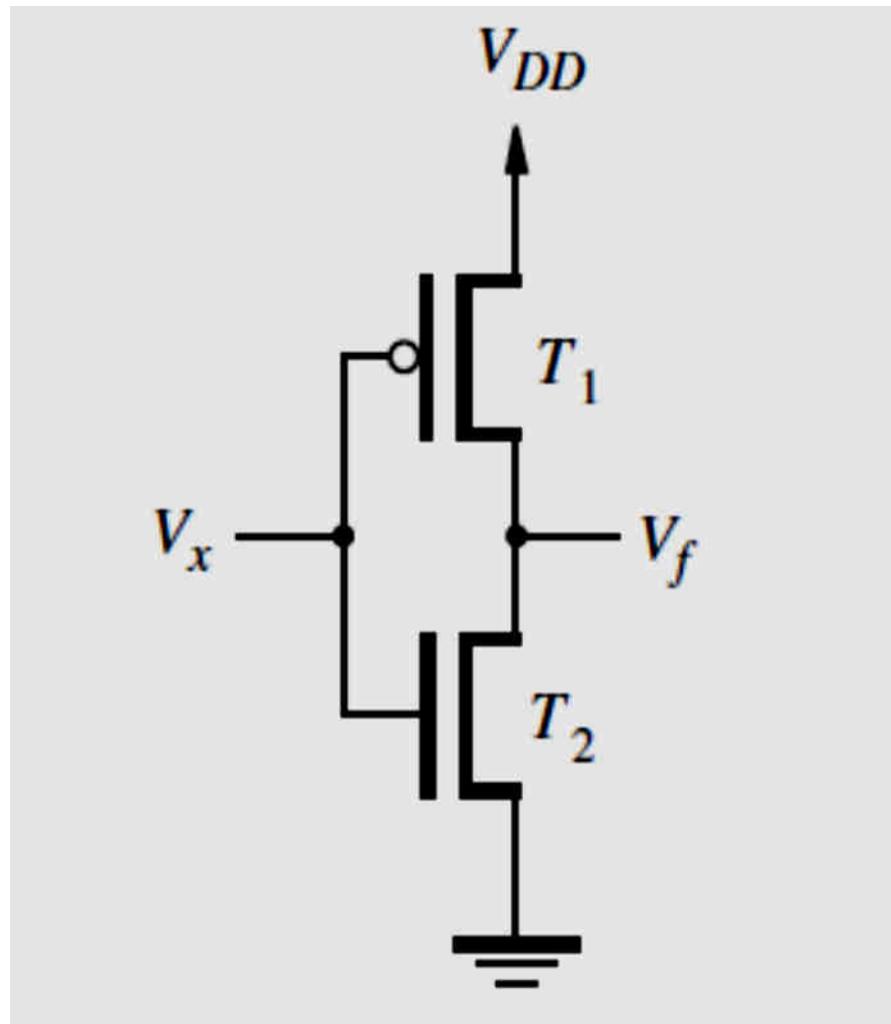


# Structure of a CMOS circuit



- For any given valuation of the input signals, either the **PDN pulls  $V_f$  down to Gnd** or the **PUN pulls  $V_f$  up to  $V_{DD}$** .

# CMOS realization of a NOT Gate

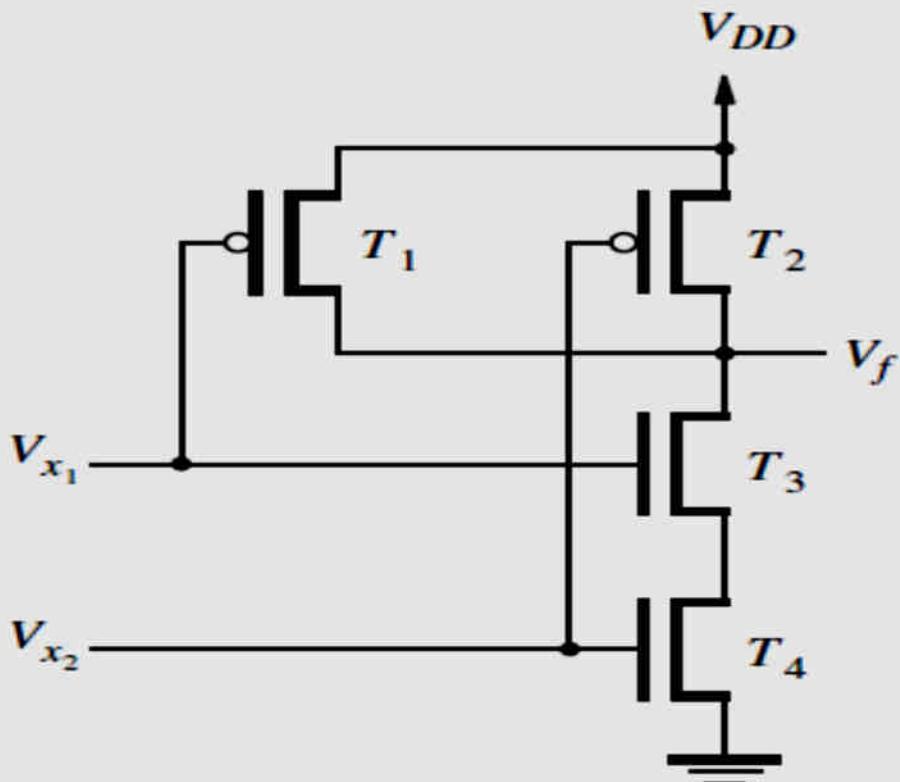


$x$	$T_1$	$T_2$	$f$
0	on	off	1
1	off	on	0

- When  $V_x=0V$ , then transistor T2 is off and transistor T1 is on. This makes  $V_f=5V$  and since T2 is off no current flows through transistors.
- When  $V_x=5V$ , T2 is on and T1 is off. Thus  $V_f =0V$ , and no current flows because T1 is off.

- No current flows in all CMOS circuits when the input is either low or high.
- **No power dissipated** under steady state conditions.

# CMOS realization of a NAND Gate



(a) Circuit

$x_1$	$x_2$	$T_1$	$T_2$	$T_3$	$T_4$	$f$
0	0	on	on	off	off	1
0	1	on	off	off	on	1
1	0	off	on	on	off	1
1	1	off	off	on	on	0

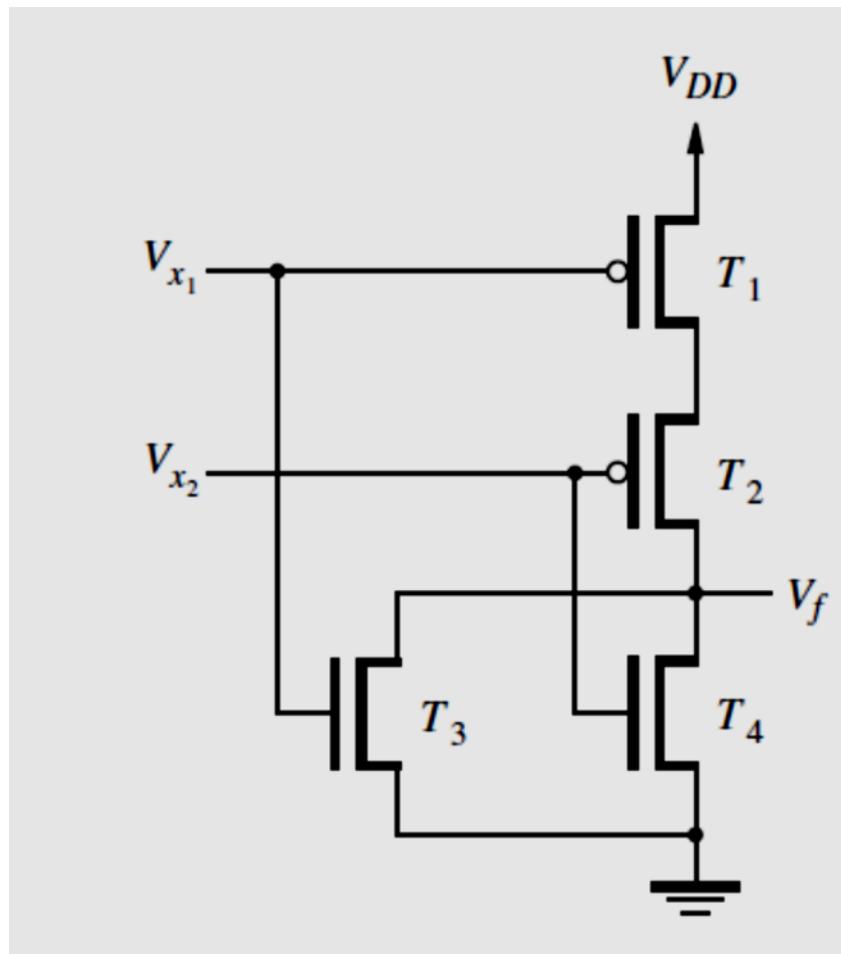
- NAND operation,  $f = (x_1 x_2)' = x_1' + x_2'$
- The above expression specifies the condition for which  **$f=1$ ; hence it defines PUN.**
- In PUN an input variable  $x_i$  turns on a transistor if  **$x_i=0$** .
- Thus  $f=1$  when either input  $x_1$  or  $x_2$  has the value 0, which means PUN must have 2 PMOS transistors **connected in parallel**.

- The PDN must implement complement of  $f$ ,

$$f' = x_1 \cdot x_2$$

Since  $f'$  is 1 when both  $x_1$  and  $x_2$  are 1, it follows that PDN must have **2 NMOS transistors connected in series**.

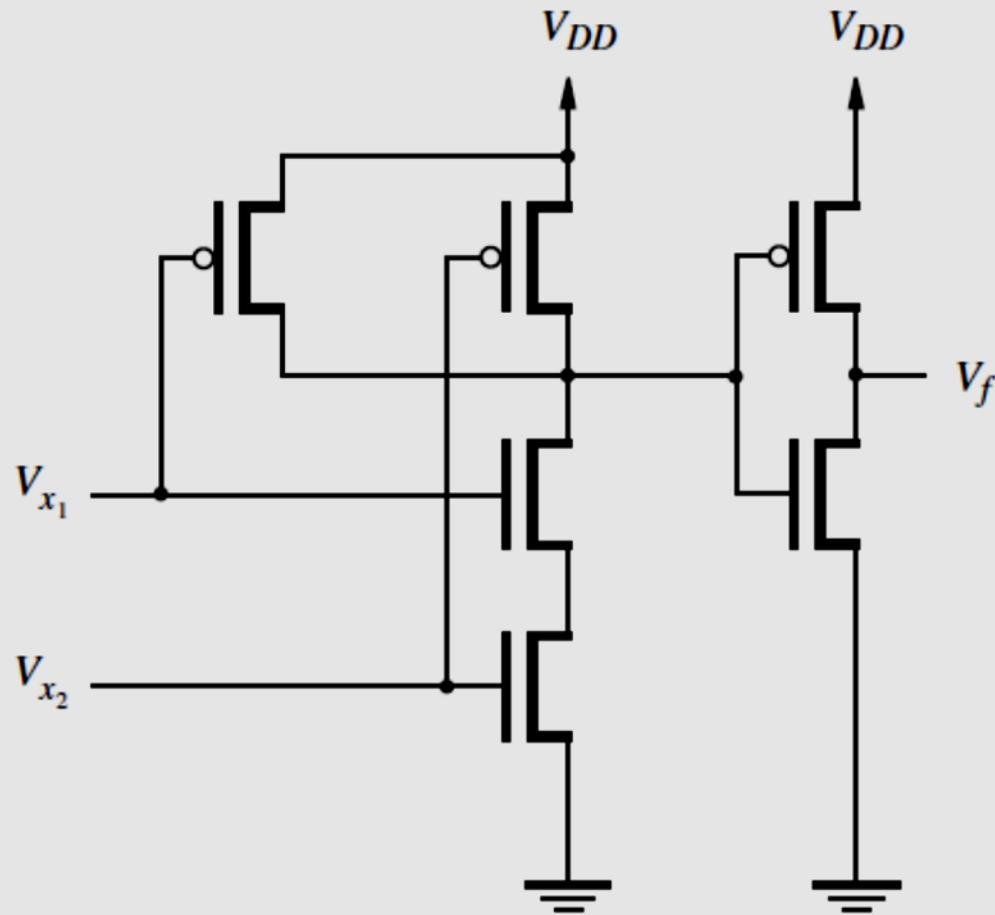
# CMOS realization of a NOR Gate



$x_1$	$x_2$	$T_1$	$T_2$	$T_3$	$T_4$	$f$
0	0	on	on	off	off	1
0	1	on	off	off	on	0
1	0	off	on	on	off	0
1	1	off	off	on	on	0

# CMOS AND gate

- CMOS AND gate is built by connecting a NAND gate to an inverter.

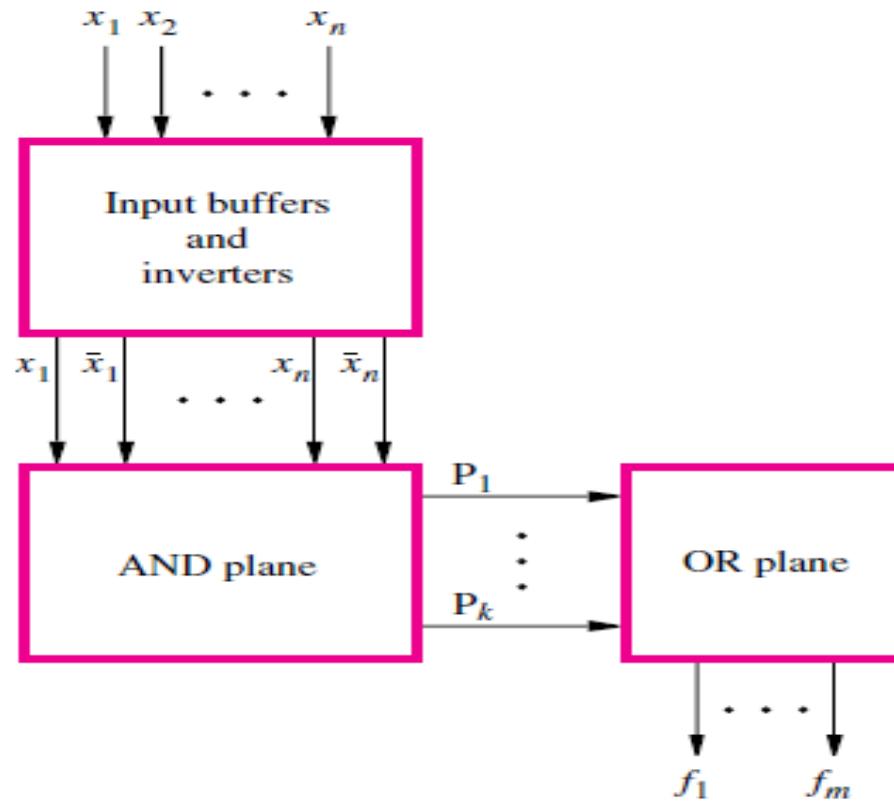


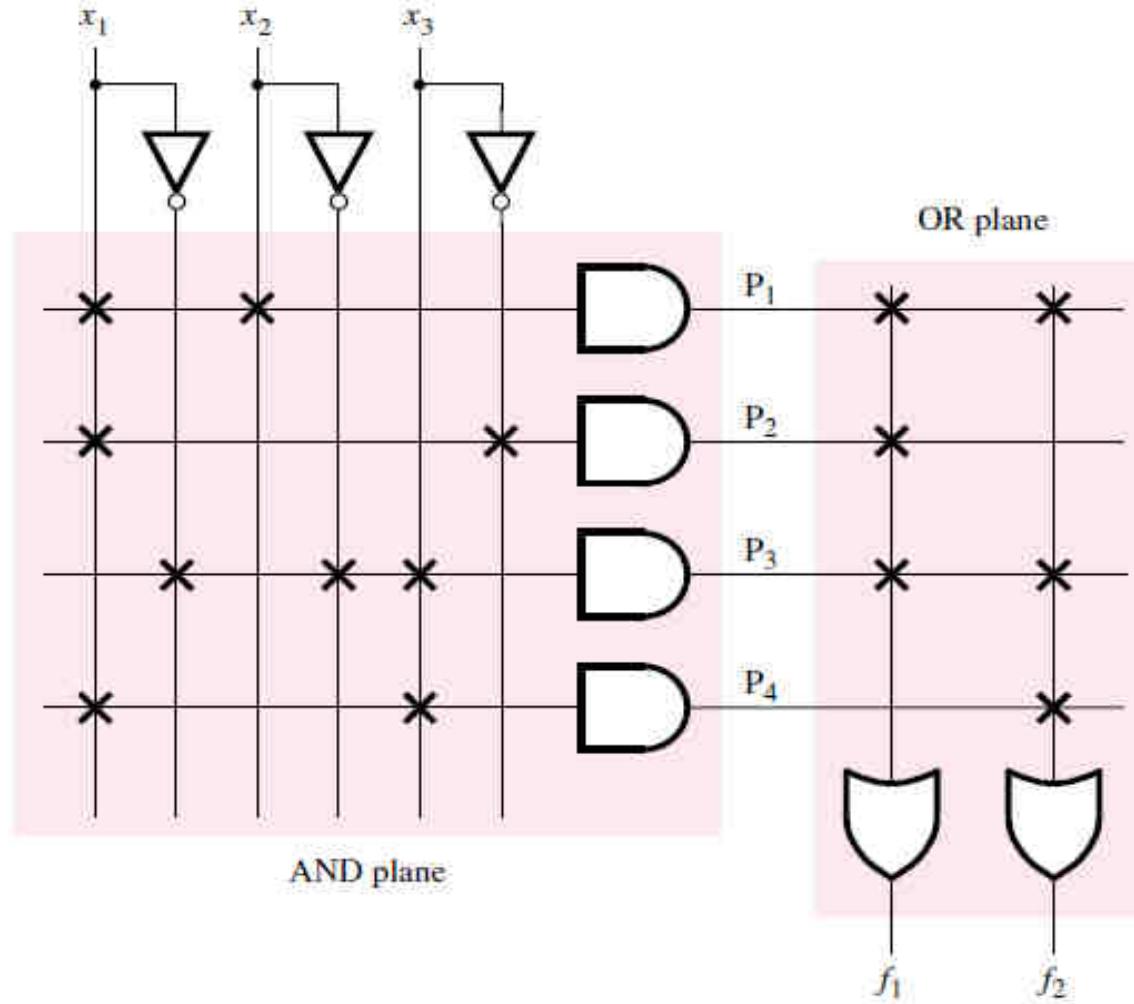
# Programmable Logic Devices

It is possible to manufacture chips that contain relatively large amounts of logic circuitry with a structure that is not fixed. Such chips are called programmable logic devices (PLDs). Two main types of PLDs are Programmable Logic Array (PLA) and Programmable Array Logic (PAL).

# Programmable Logic Array (PLA)

A PLA comprises a collection of AND gates that feeds a set of OR gates.





$$f_1 = x_1x_2 + x_1x_3' + x_1'x_2'x_3.$$

$$f_2 = x_1x_2 + x_1'x_2'x_3 + x_1x_3.$$

Example: Implement using PLA

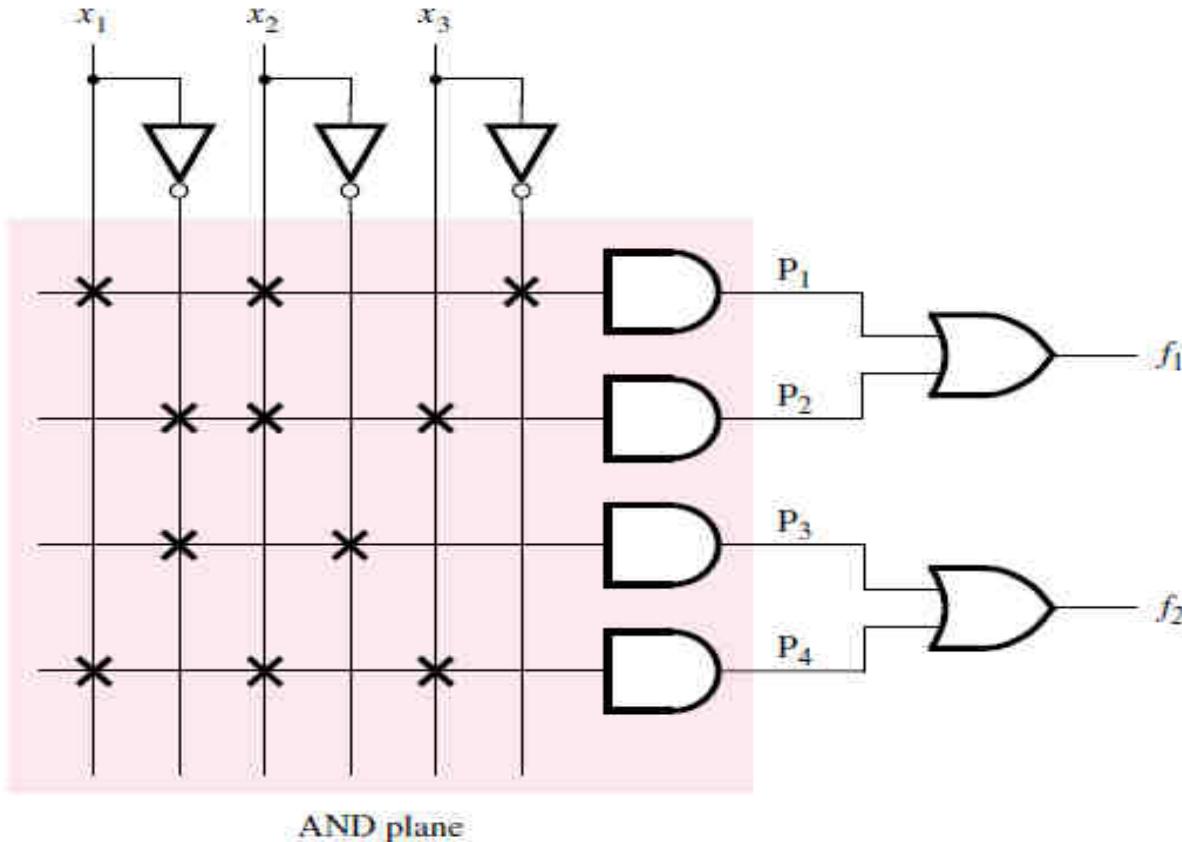
$$F_1(A,B,C) = \sum m(0,1,6,7)$$

$$F_2(A,B,C) = \sum m(2,3,4,5)$$

## **Programmable Array Logic (PAL)**

The PLAs are hard to fabricate and the speed-performance of circuits implemented in the PLAs are reduced because both AND and OR planes are programmable. These drawbacks led to the development of a similar device in which the AND plane is programmable, but the OR plane is fixed. Such a chip is known as a programmable array logic (PAL). They are simpler to manufacture, and thus less expensive than PLAs, and offer better performance.

An example of a PAL with three inputs, four product terms, and two outputs :



$$f_1 = x_1 x_2 x_3' + x_1' x_2 x_3$$

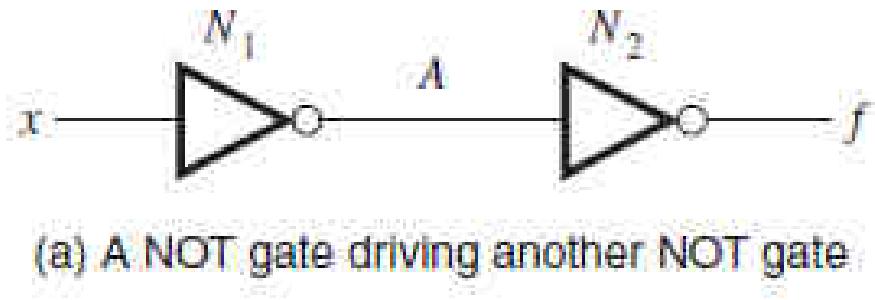
$$f_2 = x_1' x_2' + x_1 x_2 x_3.$$

Example: Implement using PAL

$$F1(A,B,C) = \sum m(0,1,6,7)$$

$$F2(A,B,C) = \sum m(2,3,4,5)$$

## Noise Margin:



- Electronic circuits are constantly subjected to random perturbations, called noise, which can alter the output voltage levels produced by the gate N1.
- It is essential that this noise not cause the gate N2 to misinterpret a low logic value as a high one, or vice versa.
- Consider the case where N1 produces its low voltage level VOL. The presence of noise may alter the voltage level, but as long as it remains less than VIL, it will be interpreted correctly by N2.

- The ability to tolerate noise without affecting the correct operation of the circuit is known as noise margin. For the low output voltage, we define the low noise margin as

$$NML = VIL - VOL$$

- A similar situation exists when N1 produces its high output voltage VOH. Any existing noise in the circuit may alter the voltage level, but it will be interpreted correctly by N2 as long as the voltage is greater than VIH. The high noise margin is defined as

$$NMH = VOH - VIH$$

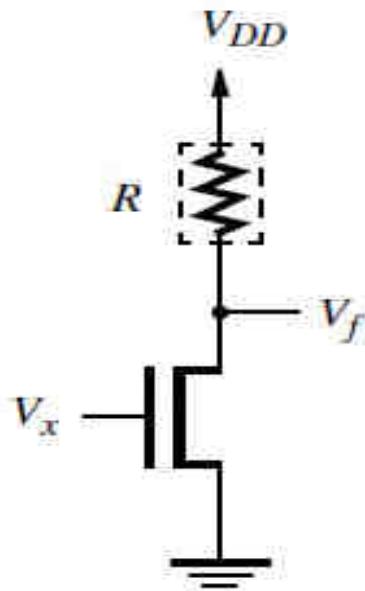
## **Fan in and Fan out:**

The fan-in of a logic gate is defined as the number of inputs to the gate. Depending on how a logic gate is constructed, it may be impractical to increase the number of inputs beyond a small value.

In real circuits each logic gate may be required to drive several others. The number of other gates that a specific gate drives is called its fan-out.

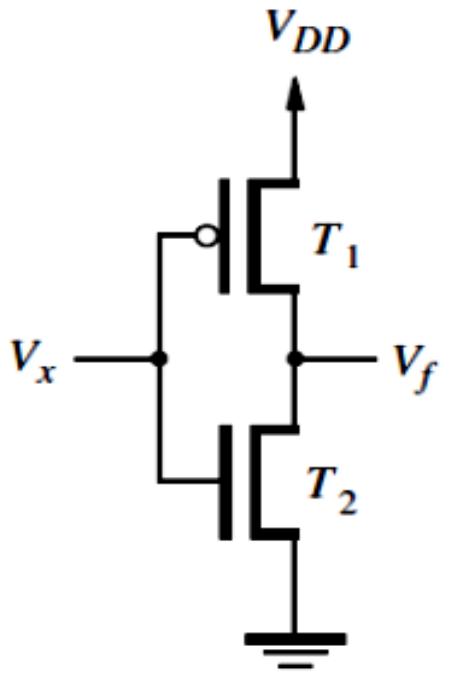
# Power Dissipation in Logic Gates

Consider the NMOS inverter shown below:



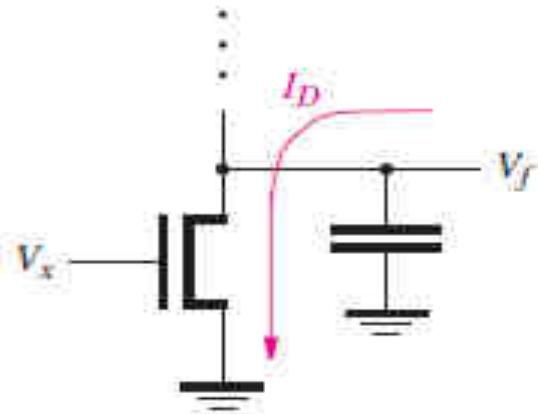
When  $V_x = 0$ , no current flows and hence no power is used. But when  $V_x = 5$  V, power is consumed because of the current flowing through the transistor.

Consider the CMOS inverter shown below:

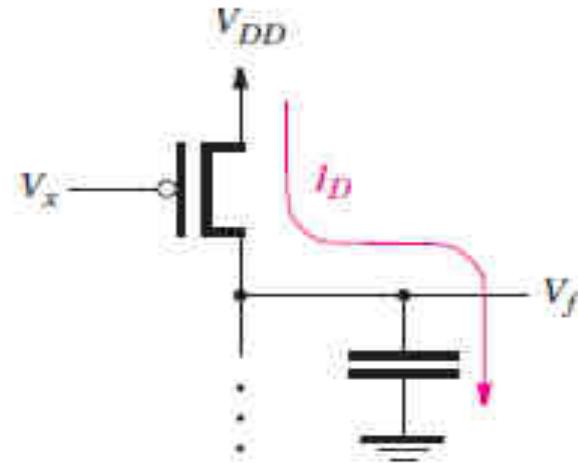


When the input  $V_x$  is low, no current flows because the NMOS transistor is off. When  $V_x$  is high, the PMOS transistor is off and again no current flows. Hence no current flows in a CMOS circuit under steady-state conditions. Current does flow in CMOS circuits, however, for a short time when signals change from one voltage level to another.

- **Static Power** : power dissipated by the current that flows in the steady state
- **Dynamic Power**: power dissipated when the current flows because of changes in signal levels.
- NMOS circuits consume static power as well as dynamic power, while CMOS circuits consume mostly dynamic power.



(a) Current flow when input  $V_x$  changes from 0 V to 5 V



(b) Current flow when input  $V_x$  changes from 5 V to 0 V

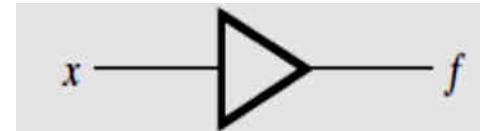
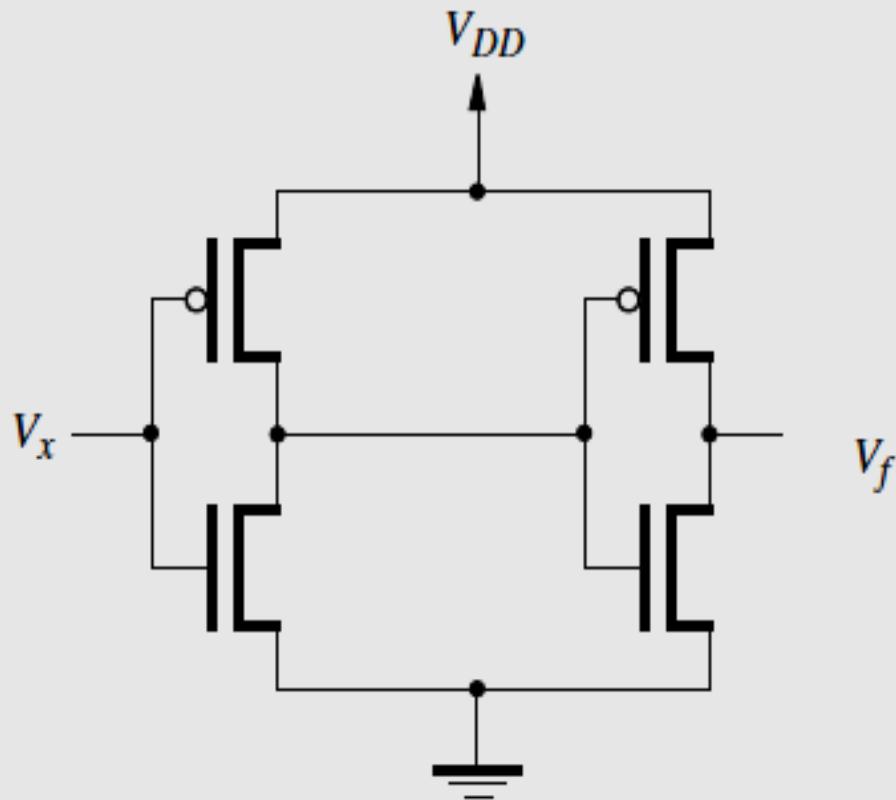
- Consider the situation in Figure (a) when  $V_f = V_{DD}$ . The amount of energy stored in the capacitor is equal to  $CV_{DD}^2/2$ . When the capacitor is discharged to 0 V, this stored energy is dissipated in the NMOS transistor.
- Similarly, for the situation in Figure (b), the energy  $CV_{DD}^2/2$  is dissipated in the PMOS transistor when C is charged up to  $V_{DD}$ . Thus for each cycle in which the inverter charges and discharges C, the amount of energy dissipated is equal to  $CV_{DD}^2$ .
- Since power is defined as energy used per unit time, the power dissipated in the inverter is the product of the energy used in one discharge/charge cycle and the number of such cycles per second, f. Hence the dynamic power consumed is

- $P_D = f CV_{DD}^2$

# Buffers

- A Buffer is a logic gate with one input  $x$  and one output  $f$  which produces  $f=x$ .
- The simplest implementation uses two inverters.
- Buffers are used for driving higher than normal capacitive loads.

# Non-inverting Buffers



(b) Graphical symbol

# Inverting Buffer

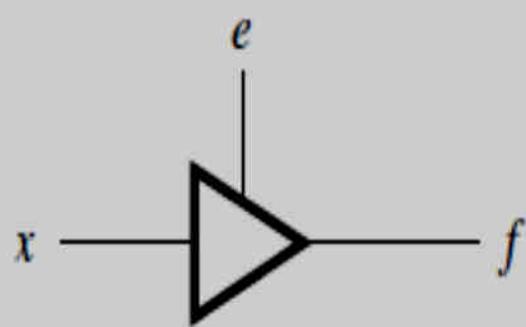
- It produces same output as an inverter,  $f=x'$  but built with large transistors.
- Buffers can handle relatively large amounts of current flow because they are built with large transistors.
- The graphical symbol for the inverting buffer is the same as for the NOT gate; an inverting buffer is just a NOT gate that is capable of driving large capacitive loads.

# Tri-state Buffers

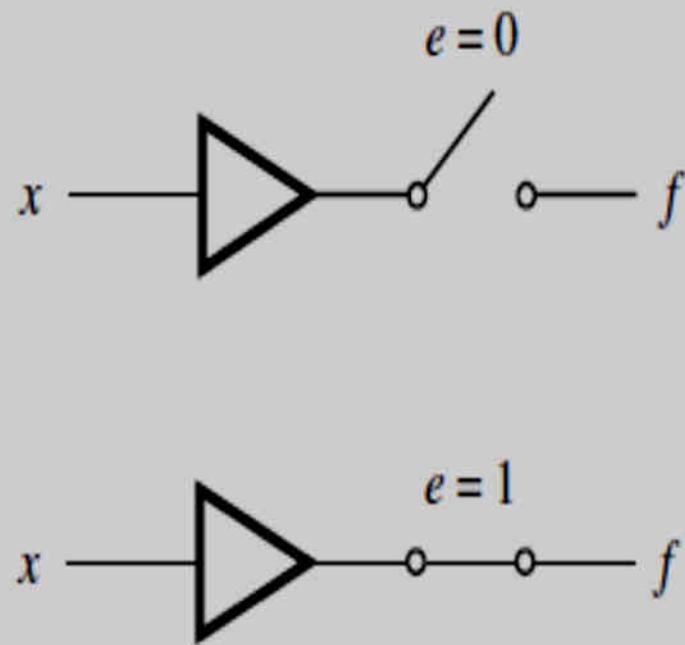
- A tri-state buffer has one input  $x$ , one output  $f$  and a control input called enable  $e$ .
- The enable input is used to determine whether or not the tri-state buffer produces an output signal.
- When  $e=0$ , the buffer is completely disconnected from the output  $f$ . When  $e=1$  the buffer drives the value of  $x$  onto  $f$  causing  $f=x$ .

- When  $e = 0$ , the output is denoted by the logic value Z, which is called the high-impedance state.
- The name tri-state derives from the fact that there are two normal states for a logic signal 0 and 1 and Z represents a third state that produces no output signal.

$e$	$x$	$f$
0	0	Z
0	1	Z
1	0	0
1	1	1

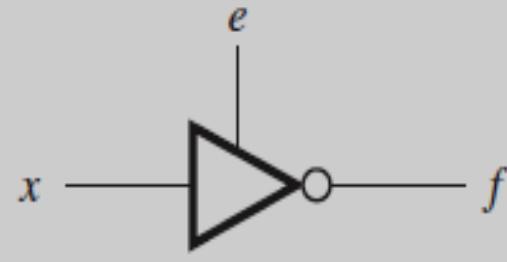
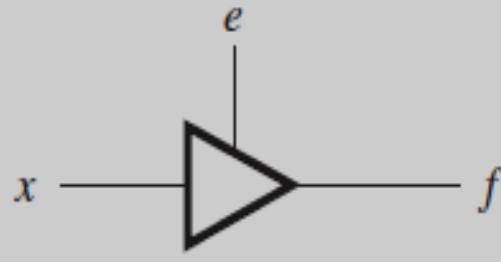


(a) A tri-state buffer



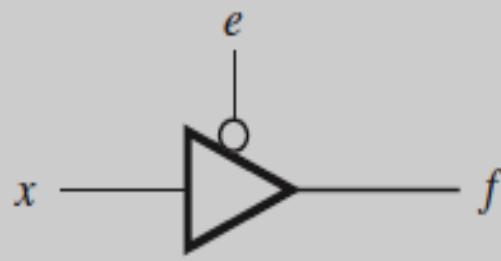
(b) Equivalent circuit

# Four Types of tri-state buffers

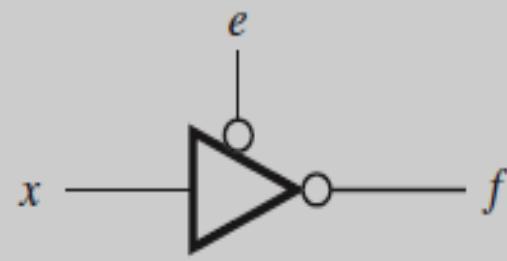


(a)

(b)



(c)

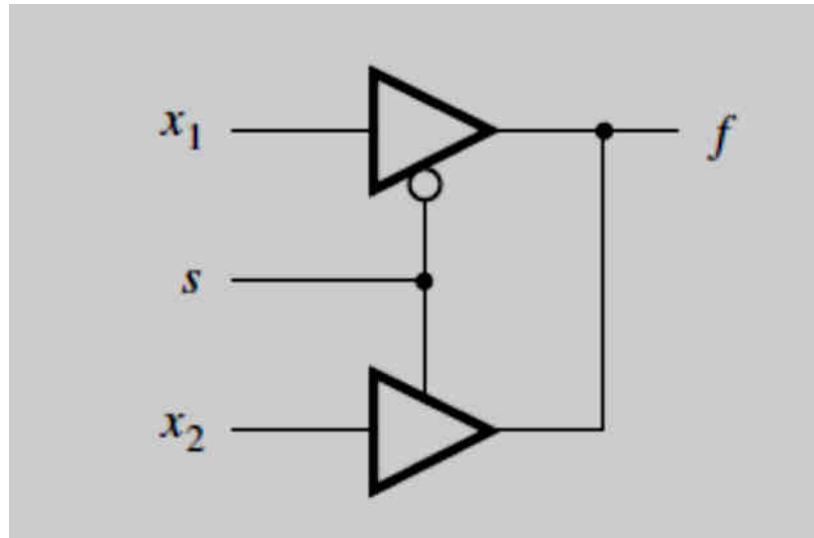


(d)

# Four Types of tri-state buffers

- The buffer in part(b) has the same behavior as the buffer in part (a) except when  $e=1$ , it produces  $f=x'$ .
- Part(c) is a tri-state buffer for which the enable signal has the opposite behavior, when  $e=0$ ,  $f=x$  and when  $e=1$ ,  $f=Z$ . (enable is active low)
- Part(d) has an active-low enable and it produces  $f=x'$  when  $e=0$ .

# Application of tri-state buffers

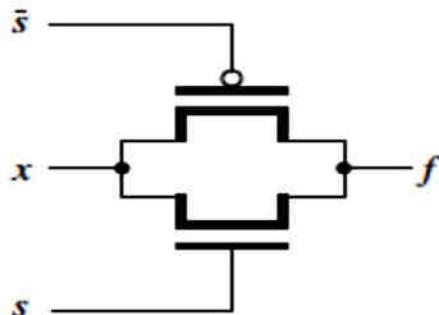


- In this circuit the output  $f$  is equal to either  $x_1$  or  $x_2$  depending on the value of  $s$ . When  $s=0$ ,  $f=x_1$  and when  $s=1$   $f=x_2$ . This is a multiplexer circuit.
- In this circuit, the outputs of the tri-state buffers are wired together. This connection is possible because the control input  $s$  is connected so that one of the two buffers is guaranteed to be in the high-impedance state.

- The  $x_1$  buffer is active only when  $s = 0$ , and the  $x_2$  buffer is active only when  $s = 1$ .
- It would be disastrous to allow both buffers to be active at the same time. Doing so would create a short circuit between VDD and Gnd as soon as the two buffers produce different values.
- For example, assume that  $x_1 = 1$  and  $x_2 = 0$ . The  $x_1$  buffer produces the output VDD, and the  $x_2$  buffer produces Gnd. A short circuit is formed between VDD and Gnd, through the transistors in the tri-state buffers.
- The kind of wired connection used for the tri-state buffers is not possible with ordinary logic gates, because their outputs are always active; hence a short circuit would occur.

# Transmission Gates

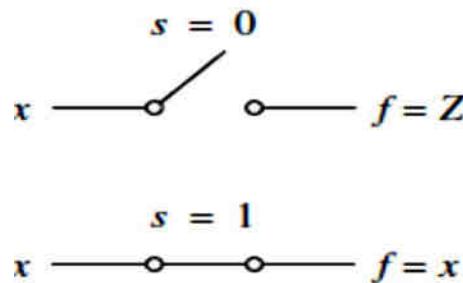
It is possible to combine an NMOS and a PMOS transistor into a single switch that is capable of driving its output terminal either to a low or high voltage



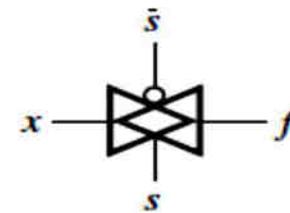
(a) Circuit

$s$	$f$
0	Z
1	$x$

(b) Truth table



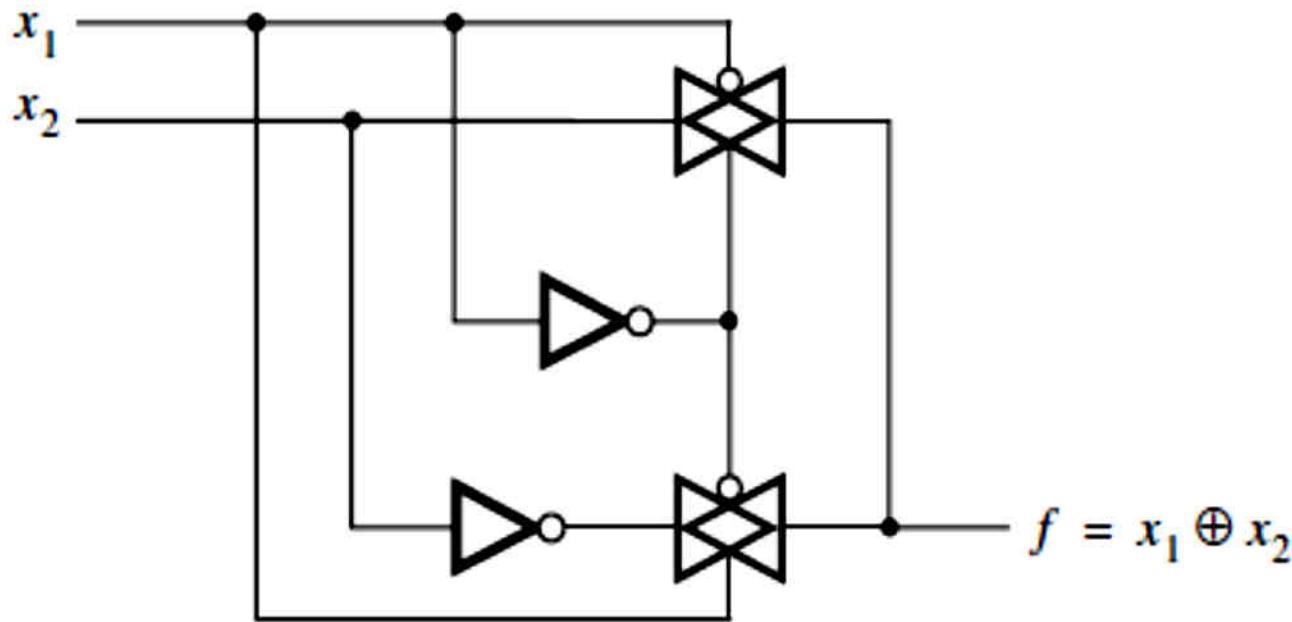
(c) Equivalent circuit



(d) Graphical symbol

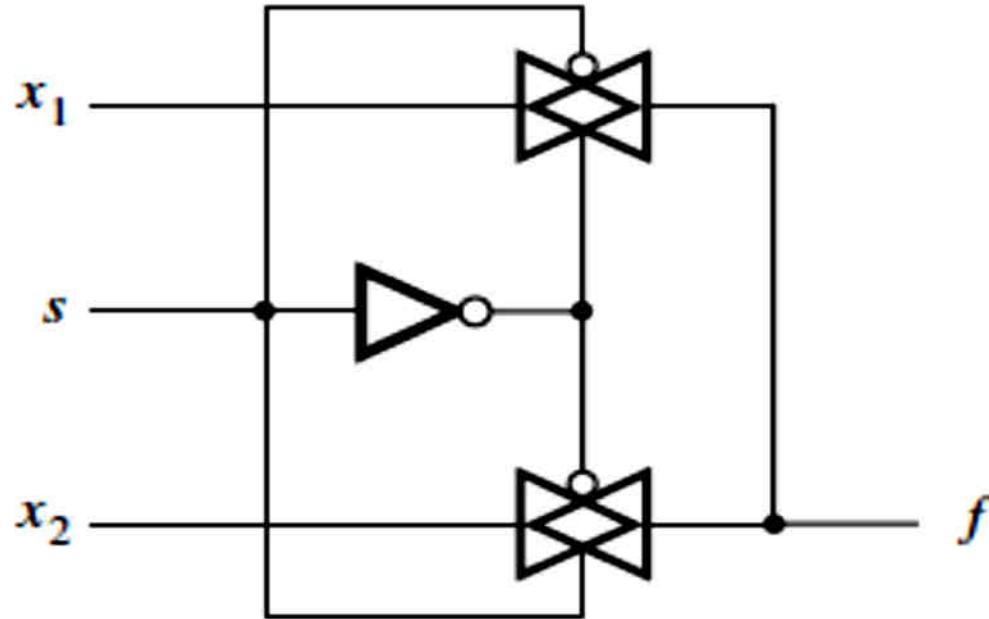
# Applications of Transmission gates

## Exclusive-OR Gates



The output  $f$  is set to the value of  $x_2$  when  $x_1 = 0$  by the top transmission gate. The bottom transmission gate sets  $f$  to  $x_2'$  when  $x_1 = 1$ .

# Multiplexer Circuit



The select input  $s$  is used to choose whether the output  $f$  should have the value of input  $x_1$  or  $x_2$ . If  $s = 0$ , then  $f = x_1$ ; if  $s = 1$ , then  $f = x_2$ .