



# DIGITAL DESIGN AND COMPUTER ORGANIZATION

## Boolean Algebra and Boolean Function

---

**Team DDCO**  
**Department of Computer Science and Engineering**

# Introduction

---

**Mathematical function:** defines a relationship between an independent variable and a dependent variable. E.g.  $A = \pi r^2$

- Example: Parabola
- Domain and range are set of real numbers
- Specified on Cartesian Plane

**Boolean function:** A mathematical function whose arguments, as well as the function itself, assume values from a two-element set (usually  $\{0,1\}$ ).

# Introduction

---

**Boolean algebra**, is a mathematical system, defined with a set of elements, a set of operators, and a number of unproved axioms or postulates.

A **Boolean function** is a function whose domain and range are the set  $\{0, 1\}$ .

A Boolean function has:

- At least one Boolean variable
- At least one Boolean operator
- At least one input from the set  $\{0, 1\}$

It produces an output that is also a member of the set  $\{0, 1\}$

# Introduction

---

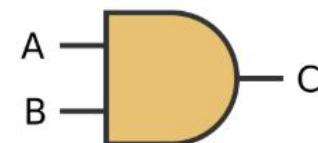
## Boolean Constants and Variables

- Inputs and outputs of Boolean function are from the set {0, 1}
  - 0 and 1 are called Boolean constants**
- In general, inputs and outputs of mathematical functions are represented by variables (like  $y = x^2$ )
  - Inputs and outputs of Boolean functions are called as Boolean variables (like a, b and y)**

Specified as a truth table:

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Specified as a logic gate:



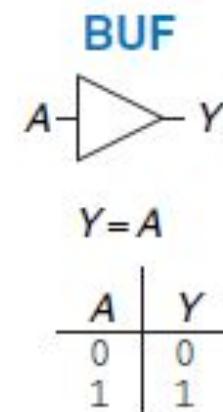
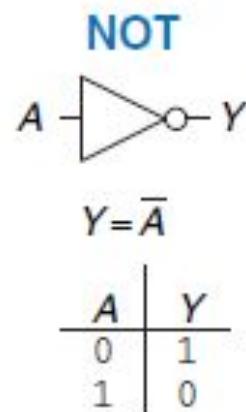
# Basic Logic Gates

Logic gates are **physical implementations of Boolean functions**.

A logic gate is a hardware component that performs a specific Boolean function.

## Single Input Logic gates

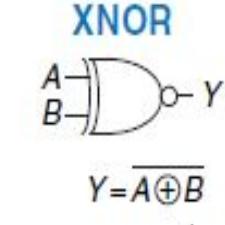
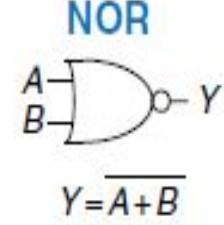
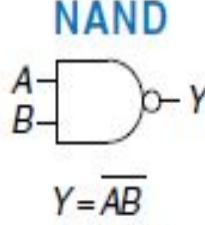
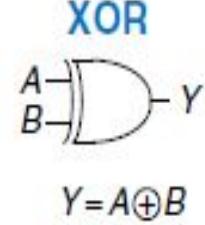
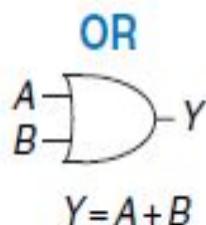
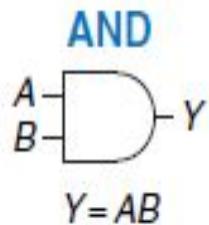
The NOT gate's output is the inverse of its input



Buffer gate simply copies the input to the output.

# Basic Logic Gates

## Two-Input Logic Gates



A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

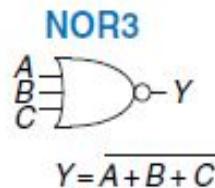
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

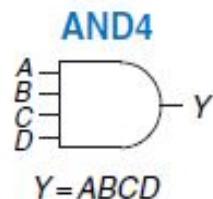
# Basic Logic Gates

## Multiple Input Logic Gates



A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

**Figure 1.20** Three-input NOR truth table

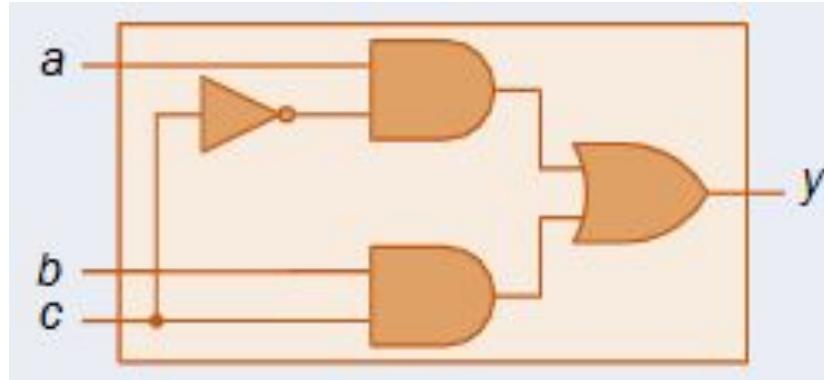


A	C	B	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

**Figure 1.22** Four-input AND truth table

# What is a logic circuit?

Multiple logic gates combined together, with the output of one gate being connected to the input of another, form a *logic circuit*.



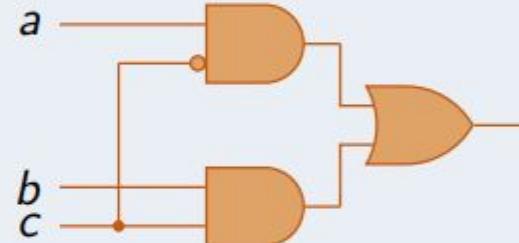
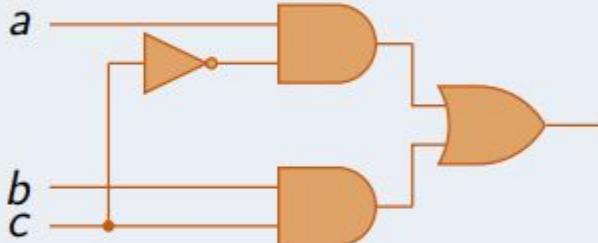
A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

# Logic Gates with Inverted Inputs

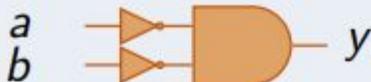
## Bubble

If a logic gate has a NOT gate on an input, the NOT gate can be replaced by a **bubble** on that input

## Bubble Example



## Bubbled AND Example



# Boolean Algebra(T1-Section 2.4)

## Algebra

In mathematics, an Algebra is composed of four things: a set of elements, operations on those elements, identity elements and laws/identities

## Standard Algebra

- **Set** Real numbers
- **Operations** Add, subtract, multiply, divide
- **Identity elements** 0 (for add), 1 (for multiply)
- **Laws/Identities** Commutative, associative, distributive, . . .

## Boolean Algebra

- **Set** {0, 1}
- **Operations** AND, OR, NOT
- **Identity elements:** 1 (for AND), 0 (for OR)
- **Laws/Identities** Commutative, associative, distributive, . . .

# Boolean Laws

**Table 2.1**  
*Postulates and Theorems of Boolean Algebra*

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

**Useful Identity**  $a + \bar{a} \cdot b = a + b$   $a \cdot (\bar{a} + b) = a \cdot b$

# Boolean Laws

---

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

<b>Statement</b>	<b>Justification</b>
$x + x = (x + x) \cdot 1$	postulate 2(b)
$= (x + x)(x + x')$	5(a)
$= x + xx'$	4(b)
$= x + 0$	5(b)
$= x$	2(a)

**THEOREM 1(b):**  $x \cdot x = x.$

<b>Statement</b>	<b>Justification</b>
$x \cdot x = xx + 0$	postulate 2(a)
$= xx + xx'$	5(b)
$= x(x + x')$	4(a)
$= x \cdot 1$	5(a)
$= x$	2(b)

# Boolean Laws

**THEOREM Z(a):**  $x + 1 = 1$ .

Statement	Justification
$x + 1 = 1 \cdot (x + 1)$	postulate 2(b)
$= (x + x')(x + 1)$	5(a)
$= x + x' \cdot 1$	4(b)
$= x + x'$	2(b)
$= 1$	5(a)

# Boolean Laws

Boolean Algebra is a branch of mathematics which deals with binary variables and their operations.

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

Statement	Justification
$x + xy = x \cdot 1 + xy$	postulate 2(b)
$= x(1 + y)$	4(a)
$= x(y + 1)$	3(a)
$= x \cdot 1$	2(a)
$= x$	2(b)

# Operator Precedence

---

The operator precedence for evaluating Boolean expressions is

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

In other words, expressions inside parentheses must be evaluated before all other operations. The next operation that holds precedence is the complement, and then follows the AND and, finally, the OR.

# Boolean Functions (T1- section 2.5)

---



Boolean function described by an **algebraic expression** consists of binary variables, the **constants 0 and 1**, and the logic operation symbols

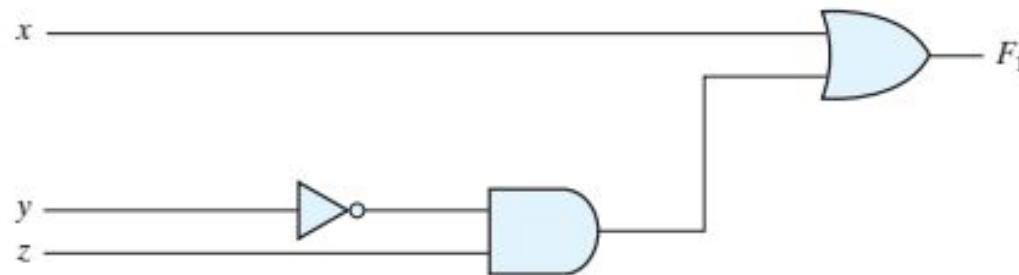
A Boolean function expresses the **logical relationship** between binary variables and is evaluated by determining the binary value of the expression for all possible values of the variables.

**EX:**  $F_1 = x + y'z$

Value of  $F_1$  if  $x = 1$  or  $y = 0$  and  $z = 1$  ?

# Boolean Functions

A Boolean function can be transformed from an algebraic expression into a circuit diagram composed of logic gates connected in a particular structure.



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

# Boolean Functions

---

For  $N$  = Number of variables in the function:

- There exists  $2^N$  possible input combinations
- This results in  $2^N$  rows in the truth table
- Binary numbers counting from 0 through  $2^N - 1$
- So, there are  $2^{2^N}$  different truth tables for Boolean functions of  $N$  variables.
- $2^{2^N}$  different Boolean functions

x	y	z	$F_1$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

# Boolean Functions

---

**There is only one way that a Boolean function can be represented in a truth table.** However, when the function is in algebraic form, it can be expressed in a variety of ways, all of which have equivalent logic.

In Boolean algebra, it is sometimes possible to obtain a simpler expression for the same function and thus reduce the number of gates in the circuit and the number of inputs to the gate.

**Designers are motivated to reduce the complexity and number of gates because their effort can significantly reduce the cost of a circuit.**

# Boolean Functions

---

*What is simplest form of below equation?*

$$F_2 = x'y'z + x'yz + xy'$$

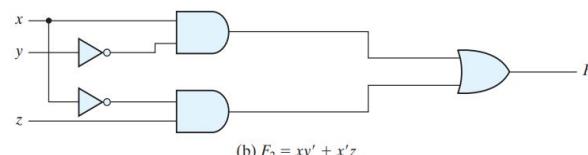
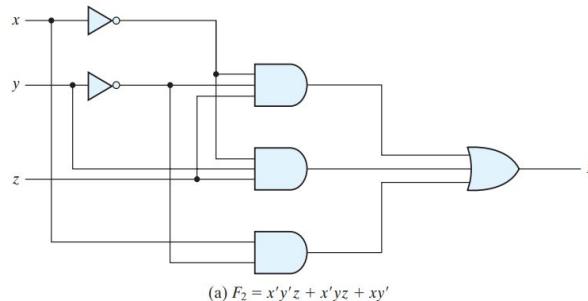
# Boolean Functions

*What is simplest form of below equation?*

$$F_2 = x'y'z + x'yz + xy'$$

$$F_2 = x'y'z + x'yz + xy'$$

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



**FIGURE 2.2**  
 Implementation of Boolean function  $F_2$  with gates

“Digital Design”, M Morris Mano, Section 2.5

# Questions

---

*Simplify the following Boolean expressions to a minimum number of literals:*

1.  $x(x' + y)$
2.  $x + x'y$
3.  $(x + y)(x + y')$
4.  $xy + x'z + yz$
5.  $(x + y)(x' + z)(y + z)$

# Questions

---

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

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

(4) and (5) are together known as consensus theorem.

# Complement of a Function

---

The complement of a function  $F$  is  $F'$  and is obtained from an interchange of 0's for 1's and 1's for 0's in the value of  $F$ . The complement of a function may be derived algebraically through DeMorgan's theorems.

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

# Questions

---

*Find the complement of the functions  $F_1 = x'y'z' + x'y'z$  and  $F_2 = x(y'z' + yz)$ .*

# Questions

***Find the complement of the functions  $F_1 = x'yz' + x'y'z$  and  $F_2 = x(y'z' + yz)$ .***

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

$$F'_1 = (x'yz' + x'y'z)' = (x'yz')'(x'y'z)' = (x + y' + z)(x + y + z')$$

$$\begin{aligned} F'_2 &= [x(y'z' + yz)]' = x' + (y'z' + yz)' = x' + (y'z')'(yz)' \\ &= x' + (y + z)(y' + z') \\ &= x' + yz' + y'z \end{aligned}$$



A simpler procedure for deriving the complement of a function is to take the dual of the function and complement each literal. This method follows from the generalized forms of DeMorgan's theorems. Remember that the dual of a function is obtained from the interchange of AND and OR operators and 1's and 0's.

# Questions

---

*Find the complement of the functions  $F1 = x'y'z + x'y'z'$  and  $F2 = x(y'z' + yz)$  taking their duals and complementing each literal.*

# Questions

---

*Find the complement of the functions  $F1 = x'yz' + x'y'z$  and  $F2 = x(y'z' + yz)$  taking their duals and complementing each literal.*

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

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

Complement each literal:  $(x + y' + z)(x + y + z') = F'_1$ .

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

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

Complement each literal:  $x' + (y + z)(y' + z') = F'_2$ .

# Boolean Formula

---

Each Boolean formula means a Boolean function as well as a logic circuit.

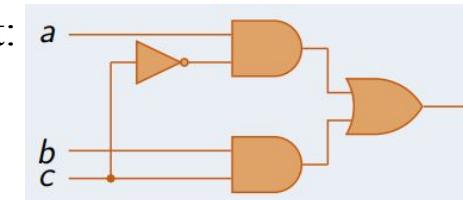
## Syntax Rules for Boolean Formulas

1. A Boolean constant (0 or 1) is a Boolean Formula
2. A Boolean variable (say  $x$ ) is a Boolean formula
3. If  $P$  and  $Q$  are Boolean formulas then so are:
  - a.  $(P \cdot Q)$
  - b.  $(P + Q)$
  - c.  $P'$

Example of Boolean Formula:  $((a \cdot c') + (b \cdot c))$

- From rule 2, Boolean variable  $a$  is a Boolean formula
- From rule 2, Boolean variable  $b$  is a Boolean formula
- From rule 2, Boolean variable  $c'$  is a Boolean formula
- From rule 3c and step (iii) above  $c'$ , is a Boolean formula
- From rule 3a, and steps (i) and (iv) above,  $(a \cdot c')$  is a Boolean formula
- From rule 3a, and steps (ii) and (iii) above,  $(b \cdot c)$  is a Boolean formula
- From rule 3b, and steps (v) and (vi) above,
- $((a \cdot c') + (b \cdot c))$  is a Boolean formula

The Boolean formula can be converted into a combinational logic circuit:



# Logic Minimization

---

From Truth Table to Boolean Formula and its Minimization

- Given a combinational logic circuit or Boolean formula, we have learnt to construct its truth table
- **But, given a truth table, how to construct a Boolean formula (or combinational logic circuit) for it?**
- **Also, as there are multiple Boolean formulas / logic circuits for each truth table, how to pick the minimal one?**
- Above problem is called **logic minimization**
  - many metrics: smallest, fastest, least power cosumption
  - our metric: smallest two level Sum of Products formula
  - may be more than one solution

# Questions

---

- *Consider Boolean functions of four inputs. What is the number of rows in the truth table of a four input Boolean function?*
  
- *What is the total number of Boolean functions of four inputs? Specify your answer as an integer.*

# Questions

---

- *Consider Boolean functions of four inputs. What is the number of rows in the truth table of a four input Boolean function?*

A four-input Boolean function's truth table will have 16 rows. ( $2^4$ )

- *What is the total number of Boolean functions of four inputs? Specify your answer as an integer.*

There are  $2^{2^N}$  different Boolean functions on n Boolean variables

Here = 65536

# Questions

---

- *Is a logic gate . . .*
  - *A Boolean function? or*
  - *A digital electronic circuit?*

# Questions

---

- *Is a logic gate...?*
  - *A Boolean function? or*
  - *A digital electronic circuit?*

It is both

This wonderful fact enables us to create the machines that perform mathematics, which we call computers.

# Questions



- There are 16 different truth tables for Boolean functions of two variables. List each truth table. Give each one a short descriptive name (such as OR, NAND, and so on).

# Questions

---

- *There are 16 different truth tables for Boolean functions of two variables. List each truth table. Give each one a short descriptive name (such as OR, NAND, and so on).*

A	B	$Y_0$	$Y_1$	$Y_2$	$Y_3$	$Y_4$	$Y_5$	$Y_6$	$Y_7$	$Y_8$	$Y_9$	$YA$	$YB$	$YC$	$YD$	$YE$	$YF$
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
		Zeros	NOR	$A' B$	NOT A	$AB'$	NOT B	XOR	NAND	AND	XNOR	B	$A' + B$	$A$	$A+B'$	OR	Ones

# Questions

---

- simplify the Boolean expression:

$$x(x' + y)$$

- A. 1
- B.  $x + y$
- C.  $x \cdot y$
- D.  $y$

# Questions

---

- simplify the Boolean expression:

$$x(x' + y)$$

- A. 1
- B.  $x + y$
- C.  $x \cdot y$
- D. y

Ans: C



**PES**  
UNIVERSITY

CELEBRATING 50 YEARS

**THANK YOU**

---

**Team DDCO  
Department of Computer Science and Engineering**



# DIGITAL DESIGN AND COMPUTER ORGANIZATION

---

## Combinational logic

Department of Computer Science and Engineering

# DIGITAL DESIGN AND COMPUTER ORGANIZATION

---

## canonical and Standard forms

Department of Computer Science and Engineering

# canonical and Standard forms

## From Truth Table to Boolean Formula and its Minimization

- Given a combinational logic circuit or Boolean formula, we have learnt to construct its truth table
- But, given a truth table, how to construct a Boolean formula (or combinational logic circuit) for it?
- Also, as there are multiple Boolean formulas / logic circuits for each truth table,
- how to pick the minimal one?

Above problem is called **logic minimization**

????

# canonical and Standard forms

## From Truth Table to Boolean Formula and its Minimization

- Given a combinational logic circuit or Boolean formula, we have learnt to construct its truth table
- **But, given a truth table, how to construct a Boolean formula (or combinational logic circuit) for it?**
- **Also, as there are multiple Boolean formulas / logic circuits for each truth table,**
- **how to pick the minimal one?**

Above problem is called **logic minimization**

- › many metrics: smallest, fastest, least power cosumption
- › our metric: smallest two level Sum of Products formula
- › may be more than one solution

# canonical and Standard forms

## Canonical and Standard form (T1: section 2.6)

- A binary variable may appear either in its normal form ( $x$ ) or in its complement form ( $x'$ ). Now consider two binary variables  $x$  and  $y$  combined with an AND operation. Since each variable may appear in either form, there are four possible combinations:  $xy$  ,  $x'y'$  ,  $x'y$  , and  $xy'$ . Each of these four AND terms is called a minterm, or a standard product.
- In a similar manner,  $n$  variables can be combined to form  $2^n$  minterms. The binary numbers from 0 to  $2^n-1$  are listed under the  $n$  variables.

# canonical and Standard forms

## Canonical and Standard form (T1: section 2.6)

**Table 2.3**

*Minterms and Maxterms for Three Binary Variables*

			Minterms	
<b>x</b>	<b>y</b>	<b>z</b>	<b>Term</b>	<b>Designation</b>
0	0	0	$x'y'z'$	$m_0$
0	0	1	$x'y'z$	$m_1$
0	1	0	$x'yz'$	$m_2$
0	1	1	$x'yz$	$m_3$
1	0	0	$xy'z'$	$m_4$
1	0	1	$xy'z$	$m_5$
1	1	0	$xyz'$	$m_6$
1	1	1	$xyz$	$m_7$

# canonical and Standard forms

## Canonical and Standard form (T1: section 2.6)

In a similar fashion, n variables forming an OR term, with each variable being primed or unprimed, provide  $2^n$  possible combinations, called **maxterms, or standard sums**.

**Table 2.3**  
*Minterms and Maxterms for Three Binary Variables*

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

# canonical and Standard forms

## Canonical and Standard form (T1: section 2.6)

It is important to note that

- (1) each maxterm is obtained from an OR term of the n variables, with each variable being unprimed if the corresponding bit is a 0 and primed if a 1, and
  - (2) each maxterm is the complement of its corresponding minterm and vice versa.
- A Boolean function can be expressed algebraically from a given truth table by forming a minterm for each combination of the variables that produces a 1 in the function and then taking the OR of all those terms.

# canonical and Standard forms

## Canonical and Standard form (T1: section 2.6)

Table 2.4

*Functions of Three Variables*

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

$f_1$  in Table 2.4 is determined by expressing the combinations 001, 100, and 111 Since each one of these minterms results in  $f_1=1$ , we have

$$\begin{aligned}f_1 &= x'y'z + xy'z' + xyz \\&= m_1 + m_4 + m_7\end{aligned}$$

$$\begin{aligned}f_2 &= x'yz + xy'z + xyz' + xyz \\&= m_3 + m_5 + m_6 + m_7\end{aligned}$$

## Sum of Min Terms

# LOGIC MINIMIZATION

## Canonical and Standard form (T1: section 2.6)

Now consider the complement of a Boolean function. It may be read from the truth table by forming a minterm for each combination that produces a 0 in the function and then ORing those terms. The complement of f1 is read as

$$F_1' = x'y'z' + x'yz' + x'yz + xy'z + xyz'$$

Which is

$$\begin{aligned}f_1 &= (x + y + z)(x + y' + z)(x' + y + z')(x' + y' + z) \\&= M_0 \cdot M_2 \cdot M_3 \cdot M_5 \cdot M_6\end{aligned}$$

Similarly

$$\begin{aligned}f_2 &= (x + y + z)(x + y + z')(x + y' + z)(x' + y + z) \\&= M_0 M_1 M_2 M_4\end{aligned}$$

**Product of max terms**

# canonical and Standard forms

## Canonical and Standard form (T1: section 2.6)



- The procedure for obtaining the product of maxterms directly from the truth table is as follows: Form a maxterm for each combination of the variables that produces a 0 in the function, and then form the AND of all those maxterms.
- **Boolean functions expressed as a sum of minterms or product of maxterms are said to be in canonical form**

# canonical and Standard forms

## Canonical and Standard form

- Application of SOP and POS form:

**SOP Form:**

**Fire Alarm System**

Fire alarm activates if:

Smoke is detected (S)

Temperature is high (T)

CO gas is detected (C)

Fire Alarm = S + T + C

This is a **classic SOP** — the output is true if **any one danger is present.**

**POS Form:**

**Traffic Light Control**

Used to control light states **only when certain combinations of sensors are false.**

For example, a green light should **not** be given if any pedestrian button is pressed **or** vehicle is waiting.

POS is used to describe “green light condition” as:

Green =  $(P' + V')$  → where P = pedestrian, V = vehicle.

# canonical and Standard forms

## Canonical and Standard form

### Sum of Minterms

- The minterms whose sum defines the Boolean function are those which give the 1's of the function in a truth table
- Since the function can be either 1 or 0 for each minterm, and since there are  $2^n$  minterms, one can calculate all the functions that can be formed with n variables to be  $2^{2^n}$
- Express the Boolean function  $F = A + B'C$  as a sum of minterms. The function has three variables: A, B, and C.

# canonical and Standard forms

## Canonical and Standard form

### Sum of Minterms

- Express the Boolean function  $F = A + B'C$  as a sum of minterms. The function has three variables: A, B, and C.
- $F = A'B'C + AB'C + AB'C + ABC' + ABC = m_1 + m_4 + m_5 + m_6 + m_7$
- When a Boolean function is in its sum-of-minterms form, it is sometimes convenient to express the function in the following brief notation:
- $F(A, B, C) = \sum (1, 4, 5, 6, 7)$
- **The  $\sum$  summation symbol stands for the OR ing of terms;**
- An alternative procedure for deriving the minterms of a Boolean function is to obtain the truth table of the function directly from the algebraic expression and then read the minterms from the truth table

# canonical and Standard forms

## Canonical and Standard form

### Sum of Minterms

The truth table shown in Table 2.5 can be derived directly from the algebraic expression by listing the eight binary combinations under variables A, B, and C and inserting 1's under F for those combinations for which  $A=1$  and  $B'C=01$ . From the truth table, we can then read the five minterms of the function to be 1, 4, 5, 6, and 7.

**Table 2.5**  
*Truth Table for  $F = A + B'C$*

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

# canonical and Standard forms



## Canonical and Standard form

### Product of Maxterms

Each of the  $2^{2^n}$  functions of n binary variables can be also expressed as a product of maxterms.

To express a Boolean function as a product of maxterms, it must first be brought into a form of OR terms. This may be done by using the distributive law,  $x+yz=(x+y)(x+z)$ .

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

# canonical and Standard forms

## Canonical and Standard form

### Product of Maxterms

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

First, convert the function into OR terms by using the distributive law:

$$F=xy+x'z$$

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

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

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

Each OR term is missing one variable;

$$=M0M2M4M5$$

$$F(x, y, z) = \pi(0, 2, 4, 5)$$

$\pi$  denotes the ANDing of maxterms; the numbers are the indices of the maxterms of the function.

$$x' + y = x' + y + zz' = (x' + y + z)(x' + y + z')$$

$$x + z = x + z + yy' = (x + y + z)(x + y' + z)$$

$$y + z = y + z + xx' = (x + y + z)(x' + y + z)$$

# canonical and Standard forms

## Conversion between Canonical Forms

The complement of a function expressed as the sum of minterms equals the sum of min terms missing from the original function. This is because the original function is expressed by those minterms which make the function equal to 1, whereas its complement is a 1 for those minterms for which the function is a 0. As an example, consider the function  $F(A, B, C) = \sum(1, 4, 5, 6, 7)$

This function has a complement that can be expressed as

$$F'(A, B, C) = \pi(0, 2, 3) = m_0 + m_2 + m_3$$

Now, if we take the complement of  $F'$  by DeMorgan's theorem, we obtain  $F$  in a different form:

$$F = (m_0 + m_2 + m_3)' = m'_0 \cdot m'_2 \cdot m'_3 = M_0 M_2 M_3 = \pi(0, 2, 3)$$

$$m'_j = M_j$$

# canonical and Standard forms

## Conversion between Canonical Forms

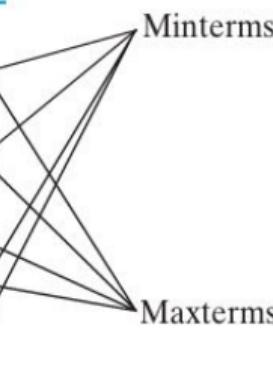
Consider, for example, the Boolean expression  $F = xy + x'z$

$$F(x, y, z) = \sum(1, 3, 6, 7)$$

$$F(x, y, z) = \pi(0, 2, 4, 5)$$

**Table 2.6**  
*Truth Table for  $F = xy + x'z$*

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



## Standard Forms-quick pointers

There are two types of standard forms: the sum of products and products of sums. The sum of products is a Boolean expression containing AND terms, called product terms, with one or more literals each. The sum denotes the ORing of these terms.

## Standard Forms-quick pointers

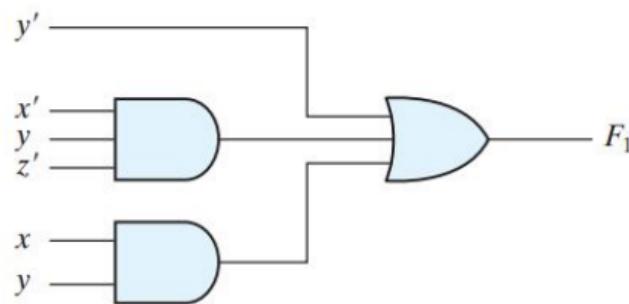
The product is an AND operation. The use of the words product and sum stems from the similarity of the AND operation to the arithmetic product (multiplication) and the similarity of the OR operation to the arithmetic sum (addition). The gate structure of the product-of-sums expression consists of a group of OR gates for the sum terms (except for a single literal), followed by an AND gate, as shown in Fig. 2.3 (b). This standard type of expression results in a two-level structure of gates.

# canonical and Standard forms

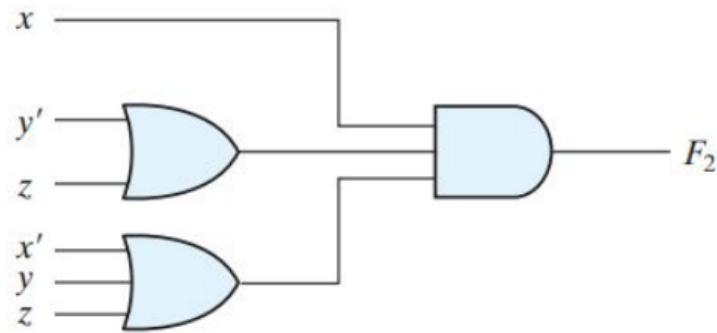
## Canonical forms- two level implementation

$$F_1 = y' + xy + x'y'z'$$

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



(a) Sum of Products



(b) Product of Sums

**FIGURE 2.3**

Two-level implementation

# canonical and Standard forms

## Canonical forms- two level implementation



A Boolean function may be expressed in a nonstandard form. For example, the function

$$F_3 = AB + C(D + E)$$

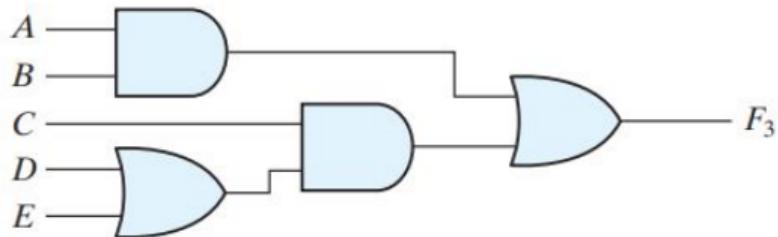
is neither in sum-of-products nor in product-of-sums form. The implementation of this expression is shown in Fig. 2.4 (a) and requires two AND gates and two OR gates. There are three levels of gating in this circuit.

It can be changed to a standard form by using the distributive law to remove the parentheses:

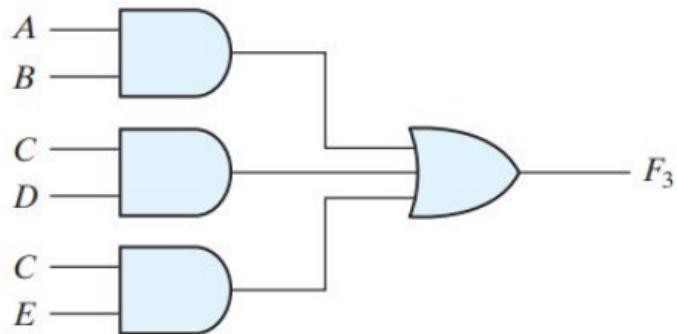
$$F3 = AB + C(D + E) = AB + CD + CE$$

# Canonical and Standard forms

## Canonical forms- three level implementation



$$(a) AB + C(D + E)$$



$$(b) AB + CD + CE$$

**FIGURE 2.4**

Three- and two-level implementation

# canonical and Standard forms

## Canonical forms- three level implementation



In general, a two-level implementation is preferred because it produces the least amount of delay through the gates when the signal propagates from the inputs to the output.

# canonical and Standard forms

## Questions



Given the Boolean function  $f(x, y, z) = \sum m(3, 5, 6, 7)$  (i.e., in SOP form), its equivalent POS (product-of-sum) form is:

- A)  $\prod M(3, 5, 6, 7)$
- B)  $\prod M(0, 1, 2, 4)$
- C)  $\prod M(0, 2, 4)$
- D)  $\prod M(1, 3, 5)$

# canonical and Standard forms

## Questions

---



Given the Boolean function  $f(x, y, z) = \sum m(3, 5, 6, 7)$  (i.e., in SOP form), its equivalent POS (product-of-sum) form is:

- A)  $\prod M(3, 5, 6, 7)$
- B)  $\prod M(0, 1, 2, 4)$
- C)  $\prod M(0, 2, 4)$
- D)  $\prod M(1, 3, 5)$

**Answer: B**

Because the POS corresponds to the maxterms *not included* in the minterm list: {0, 1, 2, 4}

The maxterm corresponding to the binary combination  $X = 1, Y = 0, Z = 1$  is:

- A)  $x + y + z$
- B)  $x' + y' + z'$
- C)  $x' + y + z'$
- D)  $x + y' + z$

# canonical and Standard forms

---



**The maxterm corresponding to the binary combination  
 $X = 1, Y = 0, Z = 1$  is:**

- A)  $x + y + z$
- B)  $x' + y' + z'$
- C)  $x' + y + z'$
- D)  $x + y' + z$

**Answer: C**

# THANK YOU

---



**Team DDCO**  
Department of Computer Science



# DIGITAL DESIGN AND COMPUTER ORGANIZATION

## Gate-Level Minimization : Karnaugh Maps

**Team DDCO**

**Department of Computer Science and Engineering**

# Gate-Level Minimization and Combinational logic Introduction :Chapter 3:

## 3.1,3.2,3.3,3.5



Gate-level minimization is the design task of finding an optimal gate-level implementation of the Boolean functions describing a digital circuit.

### Logic minimizations

Boolean Identities

K-map.(Karnaugh map)

Simplest algebraic expression is an algebraic expression with a **minimum number of terms** and with the **smallest possible number of literals in each term**. This expression produces a circuit diagram with a minimum number of gates and the minimum number of inputs to each gate.

# Introduction

---

- The complexity of the digital logic gates that implement a Boolean function is directly related to the complexity of the algebraic expression from which the function is implemented.
- Boolean expressions may be simplified by algebraic means. However, this procedure of minimization is awkward because it lacks specific rules to predict each succeeding step in the manipulative process.
- **The map method provides a simple, straightforward procedure for minimizing Boolean functions.**
- This method may be regarded as a pictorial form of a truth table. The map method is also known as the *Karnaugh map* or *K-map*

# Karnaugh Map

---

- **K-Map** is a **visual tool** used to simplify Boolean expressions.
- Introduced by **Maurice Karnaugh in 1953**, based on earlier work from 1881.
- Each **square in the K-map represents a minterm** of a Boolean function.
- A Boolean function can be expressed as a **sum of minterms** – this makes it easy to map onto a K-map.
- **Adjacent squares** represent minterms that differ by **only one literal**.
- Uses our brain's **pattern recognition ability** to simplify logic.
- Groups of 1s (e.g., 1, 2, 4, 8...) are formed to find **common literals**, which helps in **minimization**.

# Two-Variable K-Map

- There are four minterms for two variables; hence, the map consists of four squares, one for each minterm.
- The map is redrawn in (b) to show the relationship between the squares and the two variables  $x$  and  $y$ .
- The 0 and 1 marked in each row and column designate the values of variables.

$$m_1 + m_2 + m_3 = x'y' + xy' + xy = x + y$$

$m_0$	$m_1$
$m_2$	$m_3$

(a)

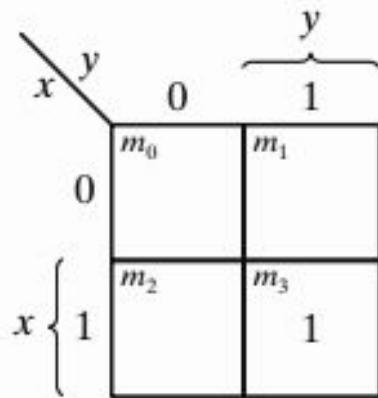
$x \backslash y$        $\overbrace{y}$   
 0      1

$m_0$	$m_1$
$x'y'$	$x'y$
$m_2$	$m_3$
$xy'$	$xy$

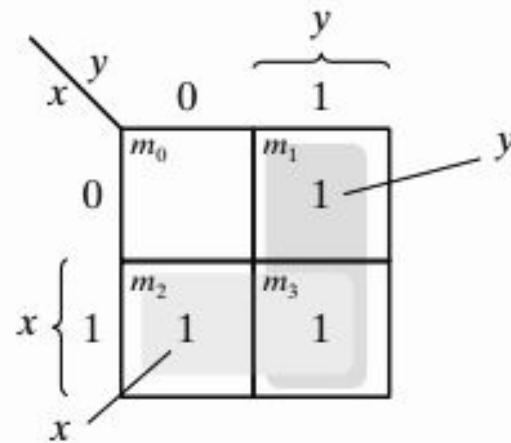
(b)

# Two-Variable K-Map

Representation of functions in the map: The function  $x + y$  is represented in the map as follows:



(a)  $xy$



(b)  $x + y$

# Two-Variable K-Map

---

- The three squares could also have been determined from the intersection of variable x in the second row and variable y in the second column, which encloses the area belonging to x or y .
- In each example, the minterms at which the function is asserted are marked with a 1.

# Three-Variable K-Map

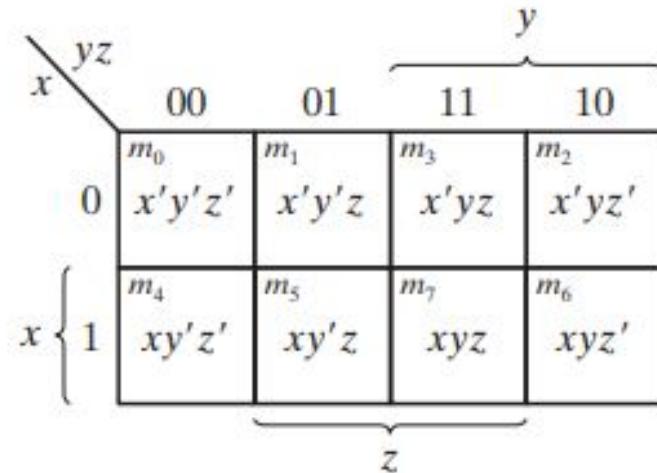
---

- The characteristic of this sequence is that only one bit changes in value from one adjacent column to the next.
- The map drawn in part (b) is marked with numbers in each row and each column to show the relationship between the squares and the three variables.
- **Gray code** ensures that only one variable changes between each pair of adjacent cells.

# Three-Variable K-Map

$m_0$	$m_1$	$m_3$	$m_2$
$m_4$	$m_5$	$m_7$	$m_6$

(a)

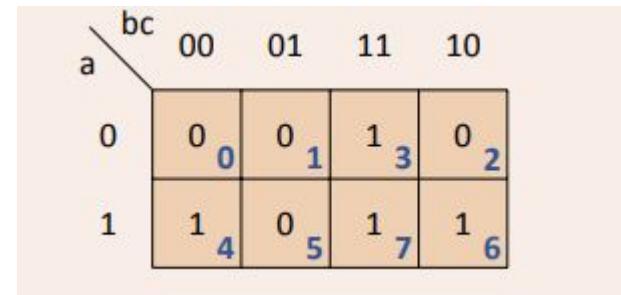
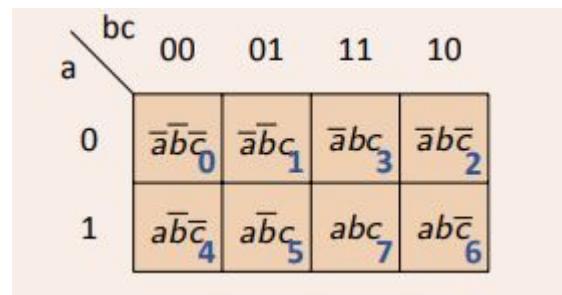


(b)

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

# Three-Variable K-Map

a	b	c	y	minterm
0	0	0	0	$\bar{a}\bar{b}\bar{c}$
0	0	1	0	$\bar{a}\bar{b}c$
0	1	0	0	$\bar{a}bc$
0	1	1	1	$\bar{a}bc$
1	0	0	1	$a\bar{b}\bar{c}$
1	0	1	0	$a\bar{b}c$
1	1	0	1	$ab\bar{c}$
1	1	1	1	$abc$



- Each square corresponds to a row of the truth table
- Any two adjacent squares differ only in one literal
- Achieved using two rows and binary order 00,01,11,10
- Notion of “wrap-around”: far left and right squares are adjacent

# K-Map Method

## K-Map Implicants

- Implicant
  - ▶ K-Map area composed of squares containing 1's
  - ▶ Area is square or rectangular (wraparound allowed)
  - ▶ No. of squares in area is a power of two (1, 2, 4, ...)
  - ▶ Each implicant corresponds to a product of literals
    - ★ Double the area, one less literal
- Prime implicant
  - ▶ Implicant having largest number of squares obeying above rules
- Essential prime implicant
  - ▶ Prime implicant containing a square not in any other prime implicant

## K-Map Method

- Include all required prime implicants
  - ▶ Include all essential prime implicants
  - ▶ Include other prime implicants such that:
    - ★ Each square containing 1 is covered
    - ★ Boolean formula is minimal (may not be unique)
- Convert required implicants to Boolean formula
  - ▶ Each implicant is a product of literals
  - ▶ Include literals which do not change over its area

# K-Map Method

---

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

- (1) all the minterms of the function are covered when we combine the squares,
- (2) the number of terms in the expression is minimized, and
- (3) there are no redundant terms (i.e., minterms already covered by other terms)

A prime implicant is a product term obtained by combining the maximum possible number of adjacent squares in the map.

If a minterm in a square is covered by only one prime implicant, that prime implicant is said to be essential.

The prime implicants of a function can be obtained from the map by combining all possible maximum numbers of squares.

# K-Map Example

- Prime implicants: **green** colour
- Essential prime implicants: **blue** colour
- Required prime implicants: **dark red** colour

		bc	00	01	11	10
		a	0	0	1	3
0	0	0	0	1	3	2
	1	1	0	1	7	6

a	b	c	y	minterm
0	0	0	0	$\bar{a}\bar{b}\bar{c}$
0	0	1	0	$\bar{a}\bar{b}c$
0	1	0	0	$\bar{a}b\bar{c}$
0	1	1	1	$\bar{a}bc$
1	0	0	1	$a\bar{b}\bar{c}$
1	0	1	0	$a\bar{b}c$
1	1	0	1	$ab\bar{c}$
1	1	1	1	$abc$

# K-Map Example

- Prime implicants: green colour
- Essential prime implicants: blue colour
- Required prime implicants: dark red colour

		bc	00	01	11	10
		a	0	0	1	0
0	0	0	0	1	1	0
	1	1	0	1	1	1

Labels below the map:

- Cell (0,0): 0
- Cell (0,1): 1
- Cell (0,2): 3
- Cell (0,3): 2
- Cell (1,0): 4
- Cell (1,1): 5
- Cell (1,2): 7
- Cell (1,3): 6

# K-Map Example

---

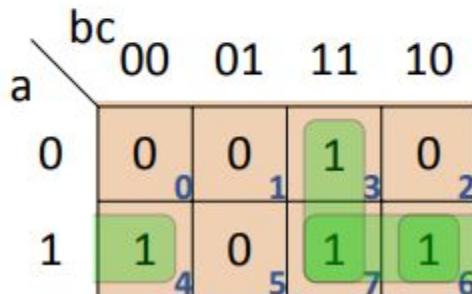
- Prime implicants: **green** colour
- Essential prime implicants: **blue** colour
- Required prime implicants: **dark red** colour

		bc 00	01	11	10
		a 0	0	1	0
a	0	0	1	1	2
	1	1	0	1	1

4 5 6 7

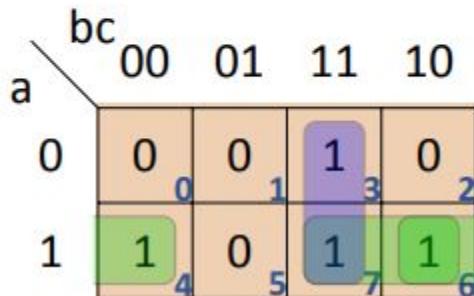
# K-Map Example

- Prime implicants: **green** colour
- Essential prime implicants: **blue** colour
- Required prime implicants: **dark red** colour
- Three prime implicants



# K-Map Example

- Prime implicants: **green** colour
- Essential prime implicants: **blue** colour
- Required prime implicants: **dark red** colour
- Three prime implicants



# K-Map Example

- Prime implicants: **green** colour
- Essential prime implicants: **blue** colour
- Required prime implicants: **dark red** colour
- Three prime implicants
- Two essential prime implicants

		bc 00	01	11	10
		a 0	0	0	1
a 1	0	0	1	3	2
	1	1	0	1	1

4 5 6 7

# K-Map Example

- Prime implicants: **green** colour
- Essential prime implicants: **blue** colour
- Required prime implicants: **dark red** colour
- Three prime implicants
- Two essential prime implicants

		bc 00	01	11	10	
		a 0	0	1	3	2
a 1	0	1	0	1	1	
	1	4	5	7	6	

# K-Map Example

- Prime implicants: **green** colour
- Essential prime implicants: **blue** colour
- Required prime implicants: **dark red** colour
- Three prime implicants
- Two essential prime implicants
- Required prime implicants:  $bc$ ,  $a\bar{c}$

		bc 00	01	11	10	
		a 0	0	1	3	2
a 1	0	1	0	1	1	
	1	4	5	7	6	

# K-Map Example

- Prime implicants: green colour
- Essential prime implicants: blue colour
- Required prime implicants: dark red colour
- Three prime implicants
- Two essential prime implicants
- Required prime implicants:  $bc, a\bar{c}$
- Boolean formula:

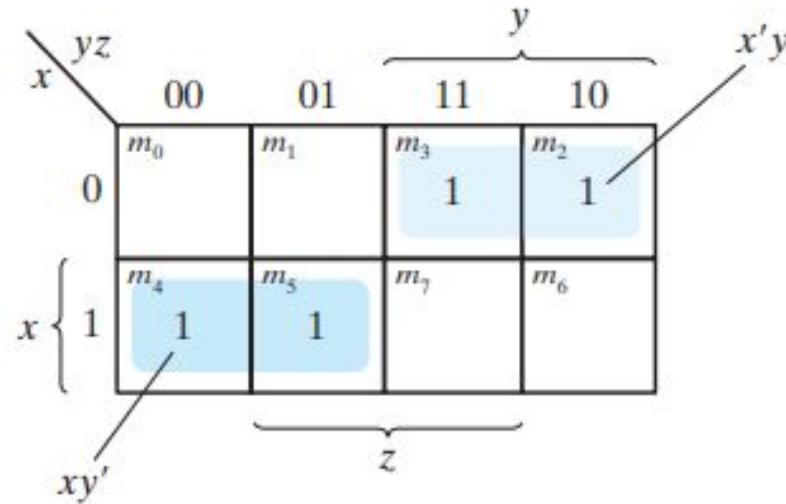
$$f(a, b, c) = a\bar{c} + bc$$

		bc 00	01	11	10
		a 0	0 0	1 1	0 2
a 1	0	1 4	0 5	1 7	1 6
	1				

# Three Variable K-Map

Simplify the Boolean function

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



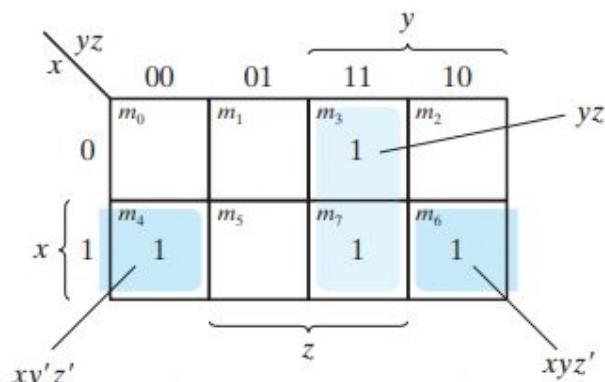
$$F = x'y + xy'$$

# Three Variable K-Map

Simplify the Boolean function

$$F(x, y, z) = \Sigma(3, 4, 6, 7)$$

$$F = yz + xz'$$



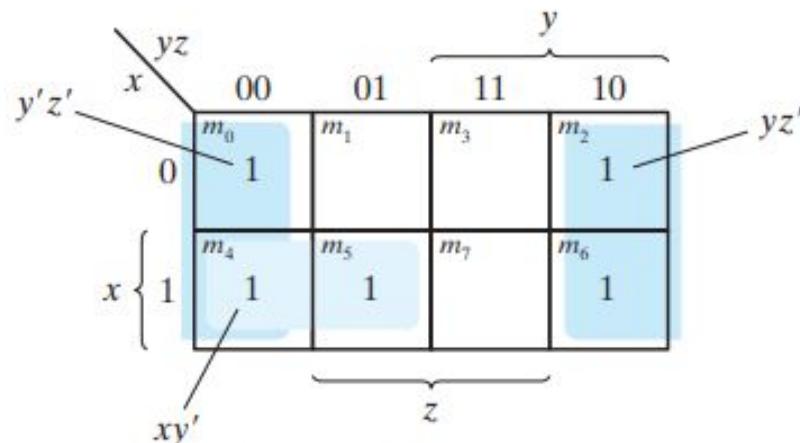
Note:  $xy'z' + xyz' = xz'$

# Three Variable K-Map

Simplify the Boolean function

$$F(x, y, z) = \Sigma(0, 2, 4, 5, 6)$$

$$F = z' + xy'$$



Note:  $y'z' + yz' = z'$

# Quick Pointers

---

- The number of adjacent squares that may be combined must always represent a number that is a power of two, such as 1, 2, 4, and 8.
- As more adjacent squares are combined, we obtain a product term with fewer literals.
- One square represents one minterm, giving a term with three literals.
- Two adjacent squares represent a term with two literals.
- Four adjacent squares represent a term with one literal.
- Eight adjacent squares encompass the entire map and produce a function that is always equal to 1.

# Example

For the Boolean function

$$F = A'C + A'B + AB'C + BC$$

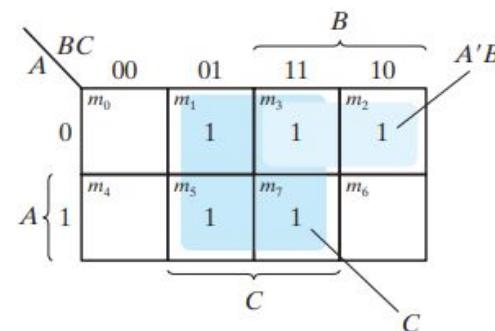
- (a) Express this function as a sum of minterms.
- (b) Find the minimal sum-of-products expression.

sum-of-minterms form as

$$F(A, B, C) = \Sigma(1, 2, 3, 5, 7)$$

The sum-of-products expression, as originally given, has too many terms. It can be simplified, as shown in the map, to an expression with only two terms:

$$F = C + A'B$$



# Four Variable K-Map

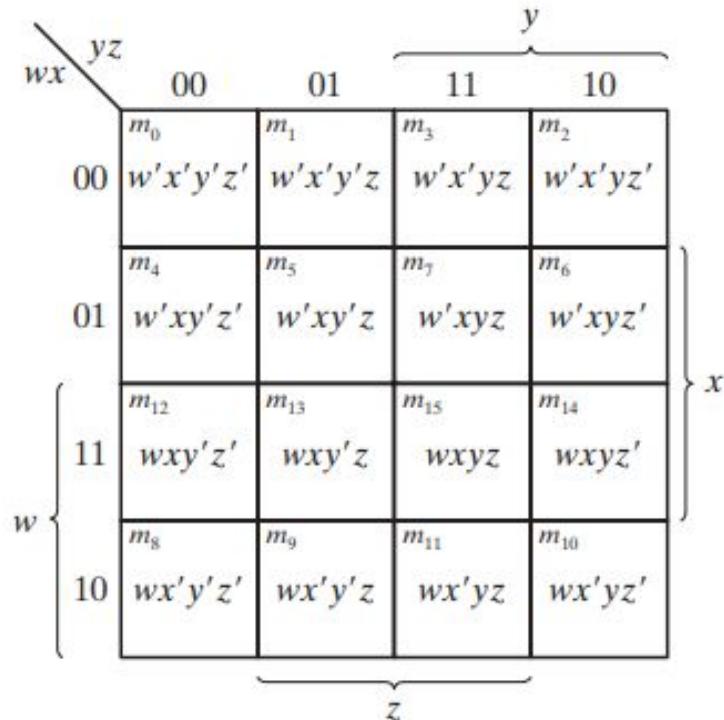
---

- The map minimization of four-variable Boolean functions is similar to the method used to minimize three-variable functions. Adjacent squares are defined to be squares next to each other.
- For example,  $m_0$  and  $m_2$  form adjacent squares, as do  $m_3$  and  $m_{11}$ .
- The combination of adjacent squares that is useful during the simplification process is easily determined from inspection of the four-variable map:

# Four Variable K-Map

$m_0$	$m_1$	$m_3$	$m_2$
$m_4$	$m_5$	$m_7$	$m_6$
$m_{12}$	$m_{13}$	$m_{15}$	$m_{14}$
$m_8$	$m_9$	$m_{11}$	$m_{10}$

(a)



(b)

# Quick Pointers

---

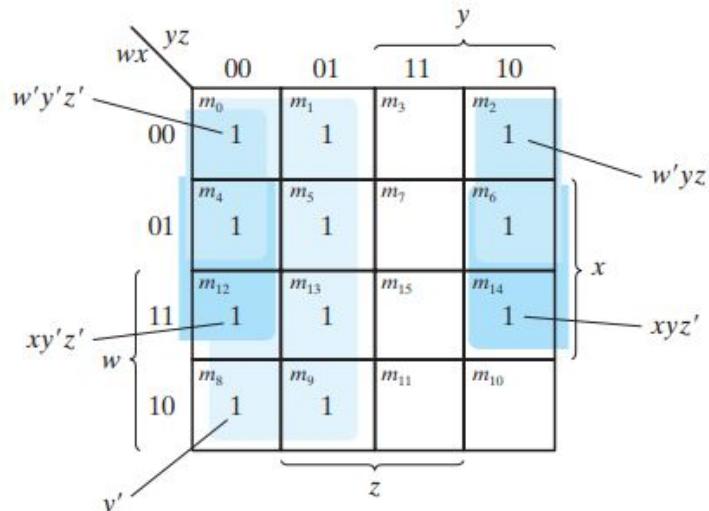
- One square represents one minterm, giving a term with four literals.
- Two adjacent squares represent a term with three literals.
- Four adjacent squares represent a term with two literals.
- Eight adjacent squares represent a term with one literal.
- Sixteen adjacent squares produce a function that is always equal to 1.

# Example

Simplify the Boolean function

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

$$F = y' + w'z' + xz'$$



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

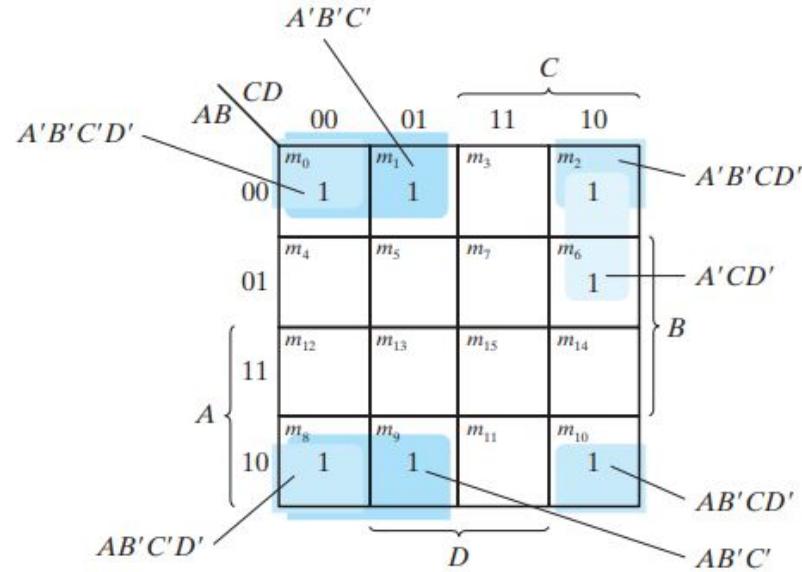
# Example

---

Simplify the Boolean function

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

# Example



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

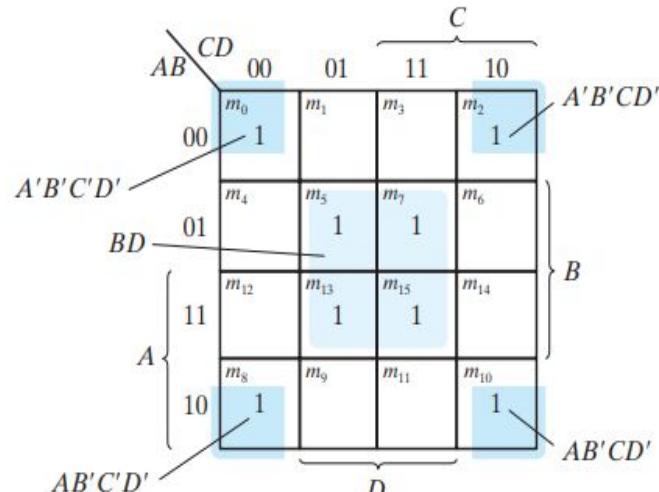
**FIGURE 3.10**

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

# Simplification using Prime Implicants

$$(a) F(w, x, y, z)$$

$$= \Sigma(0, 2, 5, 7, 8, 10, 13, 15)$$



$$\text{Note: } A'B'C'D' + A'B'CD' = A'B'D'$$

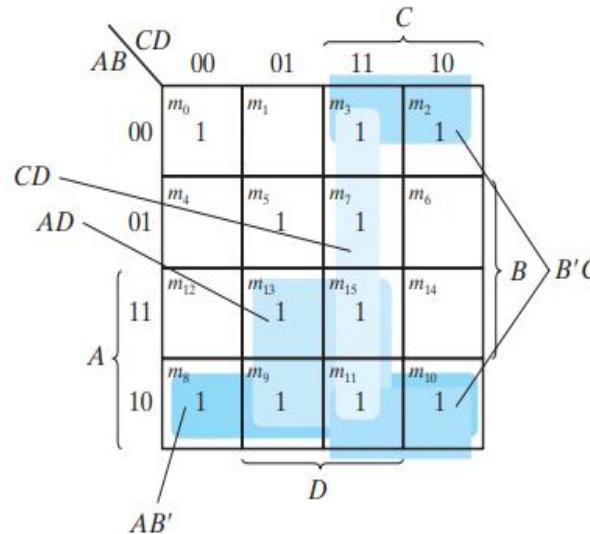
$$AB'C'D' + AB'CD' = AB'D'$$

$$A'B'D' + AB'D' = B'D'$$

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

$$(b) F(w, x, y, z)$$

$$= \Sigma(0, 2, 3, 5, 7, 8, 9, 10, 11, 13, 15)$$



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

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

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

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

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

# Another K-Map Example

## K-Map Minimization Example

	$a$	$b$	$y$
	0	0	1
	0	1	0
	1	0	1
	1	1	1

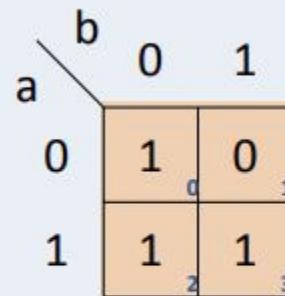
Two Input Truth Table

# Another K-Map Example

K-Map Example (two inputs)

a	b	y
0	0	1
0	1	0
1	0	1
1	1	1

Two Input Truth Table

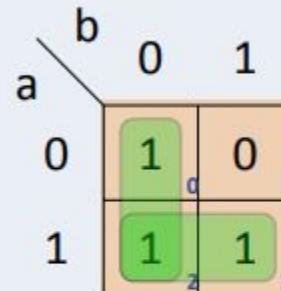


# Another K-Map Example

K-Map Example (two inputs)

a	b	y
0	0	1
0	1	0
1	0	1
1	1	1

Two Input Truth Table



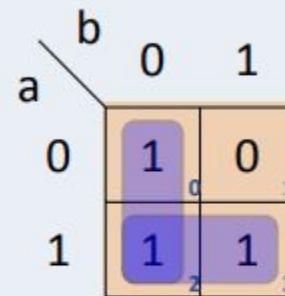
- Two prime implicants

# Another K-Map Example

K-Map Example (two inputs)

a	b	y
0	0	1
0	1	0
1	0	1
1	1	1

Two Input Truth Table



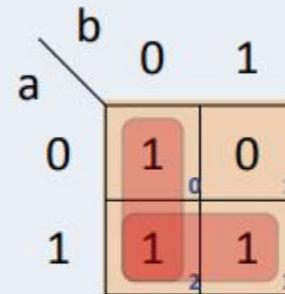
- Two prime implicants
- Two essential prime implicants

# Another K-Map Example

K-Map Example (two inputs)

a	b	y
0	0	1
0	1	0
1	0	1
1	1	1

Two Input Truth Table



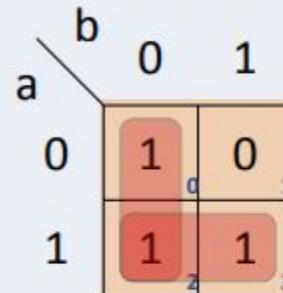
- Two prime implicants
- Two essential prime implicants
- Two required prime implicants

# Another K-Map Example

K-Map Example (two inputs)

a	b	y
0	0	1
0	1	0
1	0	1
1	1	1

Two Input Truth Table



- Two prime implicants
- Two essential prime implicants
- Two required prime implicants
- Minimized Boolean formula:  
 $f(a, b, c, d) = \bar{b} + a$

# Another K-Map Example

## K-Map Minimization Example

$a$	$b$	$c$	$y$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

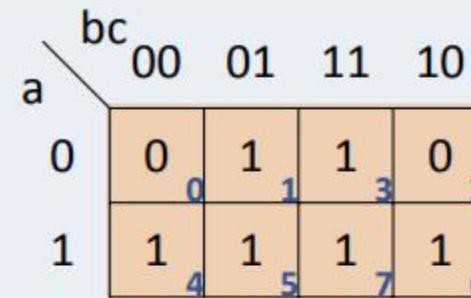
Truth table

# Another K-Map Example

K-Map Minimization Example

a	b	c	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Truth table

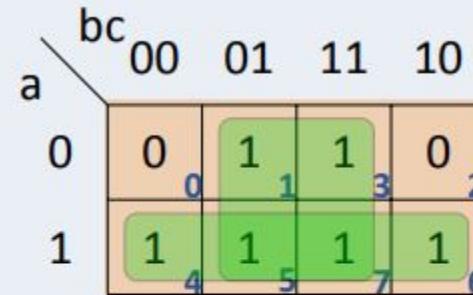


# Another K-Map Example

K-Map Minimization Example

a	b	c	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Truth table



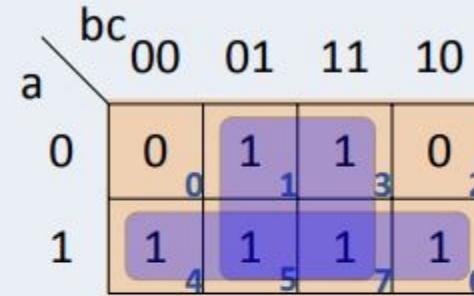
- Two prime implicants

# Another K-Map Example

K-Map Minimization Example

$a$	$b$	$c$	$y$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Truth table



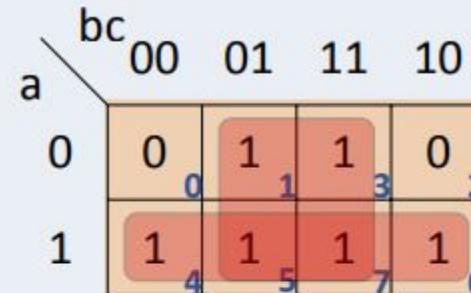
- Two prime implicants
- Two essential prime implicants

# Another K-Map Example

K-Map Minimization Example

a	b	c	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Truth table



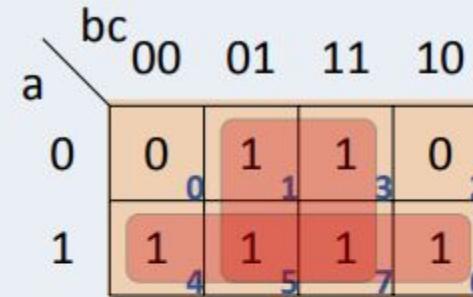
- Two prime implicants
- Two essential prime implicants
- Two required prime implicants:  $c, a$

# Another K-Map Example

K-Map Minimization Example

a	b	c	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Truth table



- Two prime implicants
- Two essential prime implicants
- Two required prime implicants:  $c$ ,  $a$
- Minimal Boolean formula:  

$$f(a, b, c) = c + a$$

# Yet Another K-Map Example

## K-Map Minimization Example

$a$	$b$	$c$	$y$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

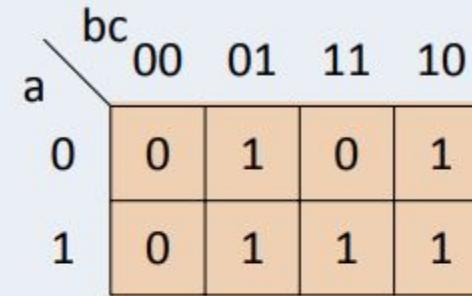
Truth table

# Yet Another K-Map Example

K-Map Minimization Example

$a$	$b$	$c$	$y$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Truth table

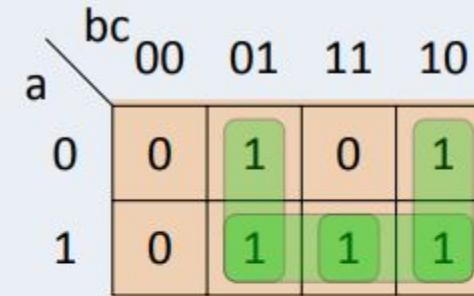


# Yet Another K-Map Example

K-Map Minimization Example

a	b	c	y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Truth table



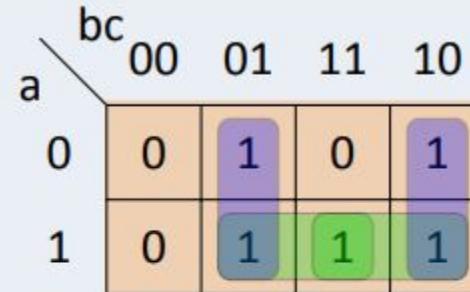
- Four prime implicants

# Yet Another K-Map Example

K-Map Minimization Example

a	b	c	y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Truth table



- Four prime implicants
- Two essential prime implicants

# Yet Another K-Map Example

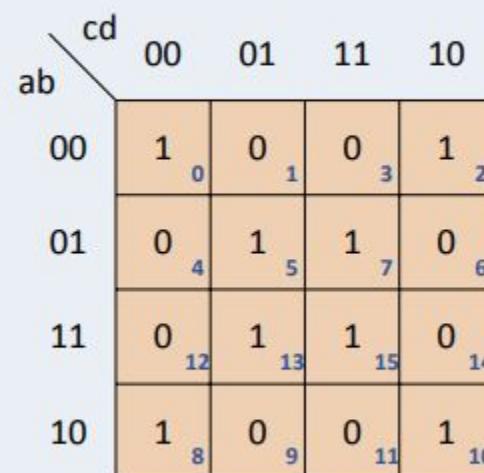
## K-Map Minimization Example

a	b	c	d	y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

# Yet Another K-Map Example

K-Map Example (four inputs)

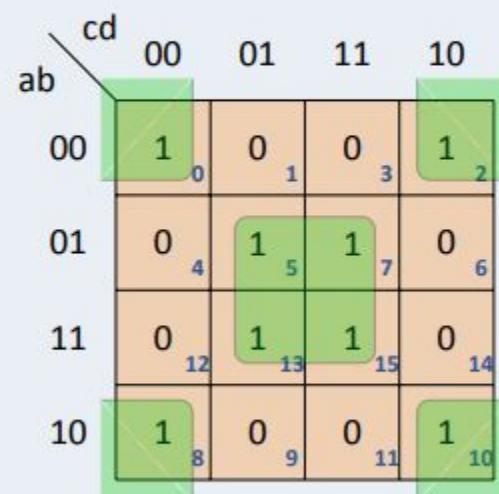
a	b	c	d	y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1



# Yet Another K-Map Example

K-Map Example (four inputs)

a	b	c	d	y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1



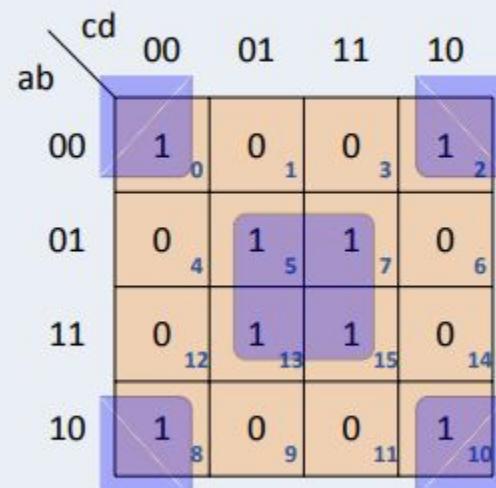
- Two prime implicants

# Yet Another K-Map Example

K-Map Example (four inputs)

a	b	c	d	y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

Four input truth table



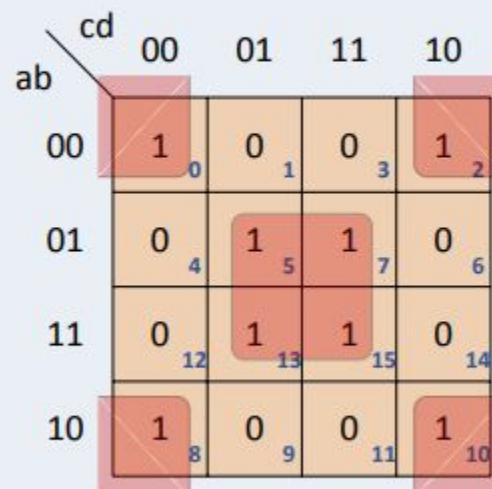
- Two prime implicants
- Two essential prime implicants

# Yet Another K-Map Example

K-Map Example (four inputs)

a	b	c	d	y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

Four Input Truth Table

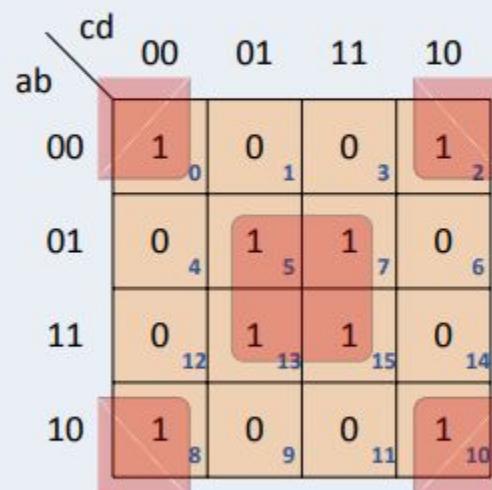


- Two prime implicants
- Two essential prime implicants
- Two required prime implicants:  $bd$ ,  $\overline{bd}$

# Yet Another K-Map Example

K-Map Example (four inputs)

a	b	c	d	y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1



- Two prime implicants
- Two essential prime implicants
- Two required prime implicants:  $bd$ ,  $\overline{bd}$
- Minimized Boolean formula:  

$$f(a, b, c, d) = bd + \overline{b} \overline{d}$$

# Gate-Level Minimization and Combinational logic Don't-care conditions.(section 3.6)

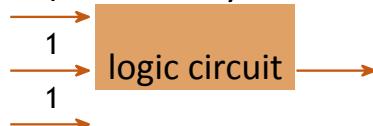


- Functions that have **unspecified outputs** for some input combinations are called incompletely specified functions .
- In most applications, we simply don't care what value is assumed by the function for the unspecified minterms.
- For this reason, it is customary to call the unspecified minterms of a function don't-care conditions

# K-MAPS

## Don't Cares

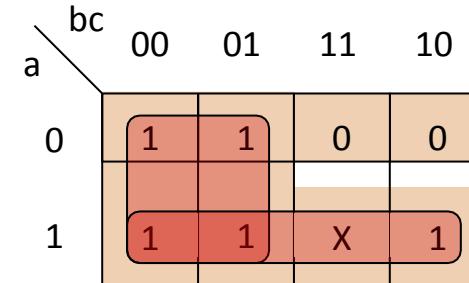
- Suppose we are asked to design a three input logic circuit all whose inputs can never be 1 simultaneously



guaranteed can't happen

$a$	$b$	$c$	$y$
0	0	0	1
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	X

Truth table



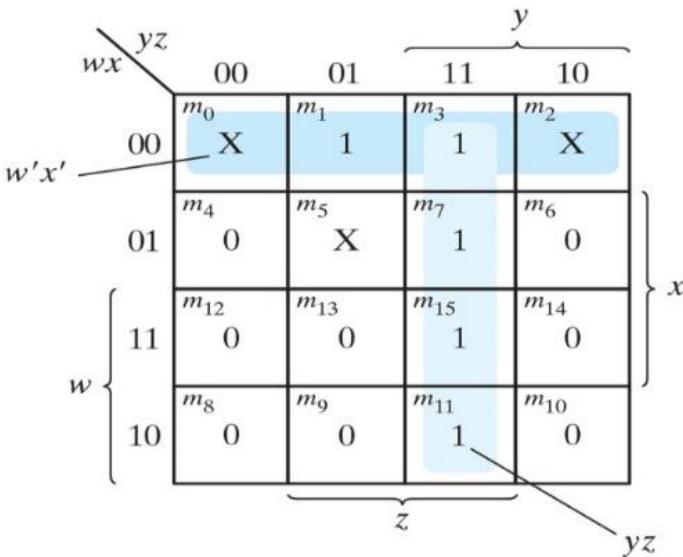
- Case 0:  $y = \bar{b} + \bar{a}a$
- Case 1:  $y = \bar{b} + a$
- Either 0 or 1 can result in smaller formula
- So initially write  $X$  in truth table and K-Map

### Don't Care

Output(s) we **don't care** about are denoted by  $X$ , can be treated as either 0 or 1

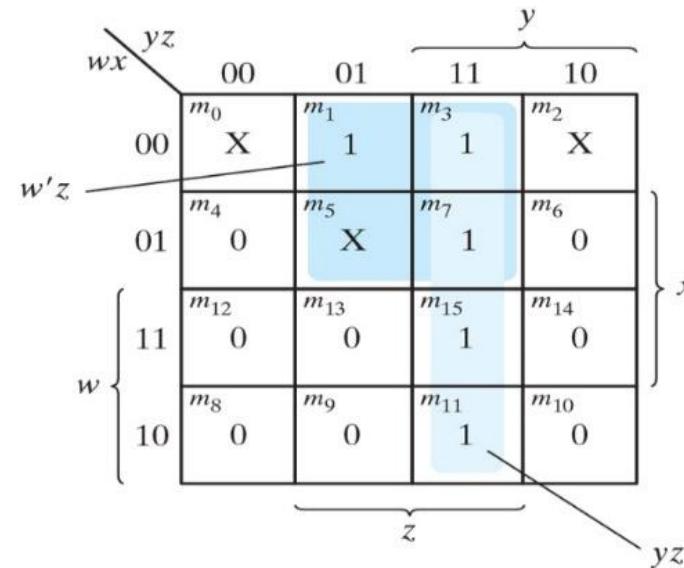
# Gate-Level Minimization and Combinational logic Don't-care conditions.

$$(a) F(w, x, y, z) = \sum m(1, 3, 7, 11, 15) + d(0, 2, 5)$$



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

$$(b) F(w, x, y, z) = \sum m(1, 3, 7, 11, 15) + d(0, 2, 5)$$



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

# Gate-Level Minimization and Combinational logic Product of sums simplification.

---



- The minimized Boolean functions derived from the map in all previous examples were expressed in sum-of-products form.
- With a minor modification, the product-of-sums form can be obtained.
- The procedure for obtaining a minimized function in product-of-sums form follows from the basic properties of Boolean functions.

# Gate-Level Minimization and Combinational logic Product of sums simplification.



- The 1's placed in the squares of the map represent the minterms of the function.
- The minterms not included in the standard sum-of-products form of a function denote the complement of the function.
- From this observation, we see that the complement of a function is represented in the map by the squares not marked by 1's
- If we mark the empty squares by 0's and combine them into valid adjacent squares, we obtain a simplified sum-of-products expression of the complement of the function (i.e., of  $F'$ ).
- The complement of  $F'$  gives us back the function  $F$  in product-of-sums form (a consequence of DeMorgan's theorem)

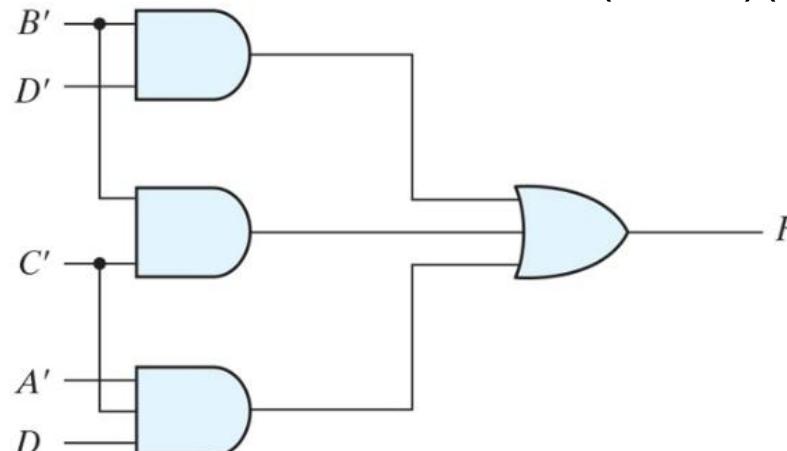
# Gate-Level Minimization and Combinational logic

Product of sums simplification.

## Gate implementations of the function –

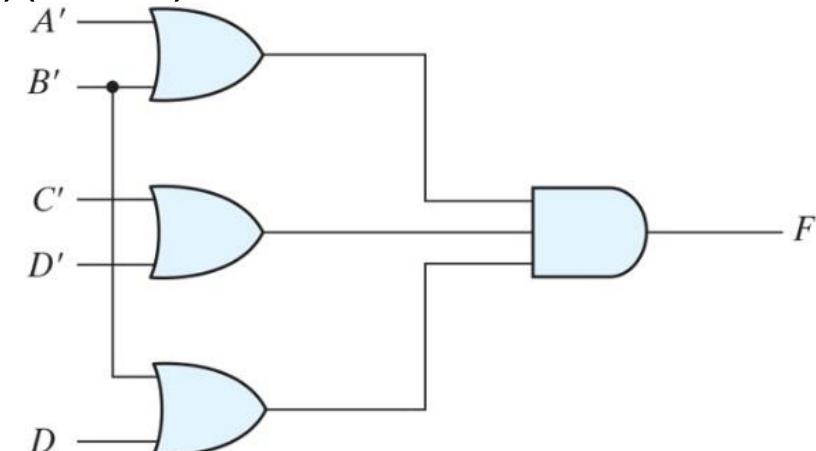
$$F(A, B, C, D) = (0, 1, 2, 5, 8, 9, 10)$$

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



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

SOP



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

POS

# Real-Life Applications of K-Maps

---

Application: Seatbelt Warning System in Cars:

Trigger a seatbelt warning buzzer based on the driver's seat occupancy, seatbelt status, and car ignition.

Variable

A

B

C

W

Meaning

Seat Occupied (1 = Yes, 0 = No)

Seatbelt Buckled (1 = Yes, 0 = No)

Ignition ON (1 = ON, 0 = OFF)

Warning buzzer ON (1 = ON, 0 = OFF)- output

# Real-Life Applications of K-Maps

$A \ B \downarrow \ C \rightarrow$	$C = 0$	$C = 1$
$A=0, B=0$	0	0
$A=0, B=1$	0	0
$A=1, B=1$	0	0
$A=1, B=0$	0	1

$$W = A \cdot B' \cdot C$$

Circuit:

- 1.NOT gate to invert  $B \rightarrow B'$
- 2.3-input AND gate with inputs  $A, B'$  and  $C$
- 3.Output drives buzzer (via transistor if needed)

# Real-Life Applications of K-Maps

---

Digital Circuits: Karnaugh maps are widely used in the design of digital circuits. The simplified expressions obtained from K-Maps can be easily translated into logic gates, making it easier to design and implement the circuit.

Computer memory: K-Maps are used in the design of computer memory. The simplified expressions obtained from K-Maps help in reducing the size and complexity of the memory circuit.

Communication systems: K-Maps are used in the design of communication systems. The simplified expressions obtained from K-Maps help in reducing the complexity and improving the efficiency of the communication system.

# Real-Life Applications of K-Maps

---

Consumer electronics: K-Maps are used in the design of consumer electronics such as televisions, radios, and other electronic devices. The simplified expressions obtained from K-Maps help in reducing the size and complexity of electronic devices.

Automotive electronics: K-Maps are used in the design of automotive electronics such as engine control units, braking systems, and other electronic systems. The simplified expressions obtained from K-Maps help in reducing the size and complexity of the electronic systems, making them more efficient and reliable.

Used in error detection like Parity Checkers, Cyclic Redundancy Check (CRC) etc.

# Try it Out

---

## Minimize using K-Maps

- $f(a, b, c) = \Sigma(0, 3, 5)$
- $f(a, b, c) = \Sigma(0, 1, 2, 4, 5, 6)$
- $f(a, b, c) = \Sigma(0, 2, 3, 4, 6, 7)$
- $f(a, b) = \Sigma(0, 1, 2, 3)$
- $f(a, b, c, d) = \Sigma(0, 1, 5, 7, 15, 14, 10)$
- $g(w,x,y,z) = \Sigma(1,3,6,12,15) + d(0,2,5)$
- $s(w,x,y,z) = \Sigma(0,1,2,4,5,7,9) + d(3,8,15)$



**PES**  
UNIVERSITY

CELEBRATING 50 YEARS

**THANK YOU**

---

**Team DDCO  
Department of Computer Science and Engineering**



# DIGITAL DESIGN AND COMPUTER ORGANIZATION

---

## Gate Level Minimization: NAND and NOR Implementation

Department of Computer Science and Engineering

# DIGITAL DESIGN AND COMPUTER ORGANIZATION

---

## NAND and NOR implementation

Department of Computer Science and Engineering

# Gate-Level Minimization

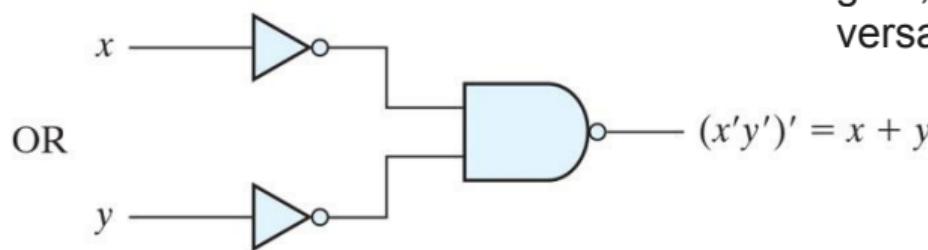
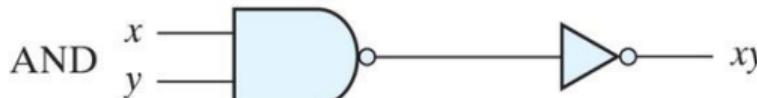
## NAND and NOR implementation(Universal Gates)



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

# Gate-Level Minimization

## NAND and NOR implementation(T1-3.6)



Universal logic gates, NAND and NOR, are essential building blocks in digital electronics. They can create any other logic gate, making them highly versatile for circuit design

NAND



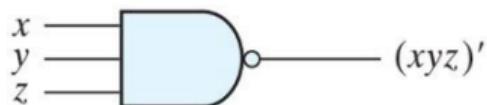
$$Y = \overline{AB}$$

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

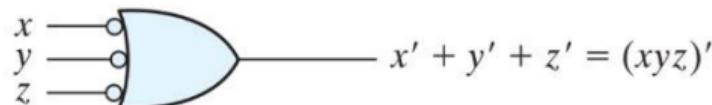
Logic operations with NAND gates.

# Gate-Level Minimization

## NAND and NOR implementation



(a) AND-invert



(b) Invert-OR

Two graphic symbols for a **three-input NAND gate**.

# Gate-Level Minimization

## NAND and NOR implementation

---



- The AND-invert symbol has been defined previously and consists of an AND graphic symbol followed by a small circle negation indicator referred to as a bubble.
- It is possible to represent a NAND gate by an OR graphic symbol that is preceded by a bubble in each input.
- The **invert-OR symbol for the NAND gate follows DeMorgan's theorem** and the convention that the negation indicator (bubble) denotes complementation.

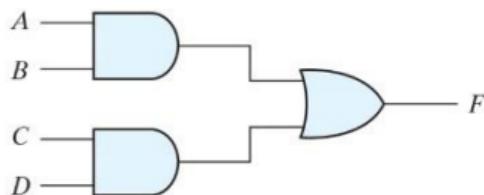
The implementation of Boolean functions with NAND gates requires that the functions be in sum-of-products form.

Implement the function  $F = AB + CD$  with

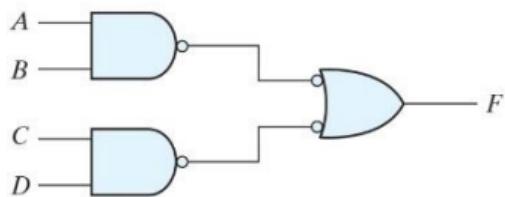
- And ,OR gate
- NAND gates only(AND-Invert only)
- AND -Invert and Invert-OR

## Gate-Level Minimization

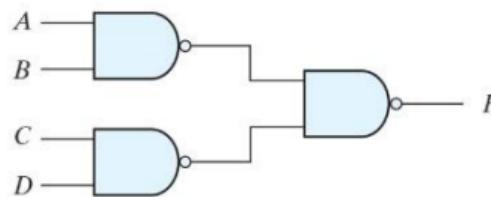
### NAND and NOR implementation( 2 level Implementation)



(a)



(b)



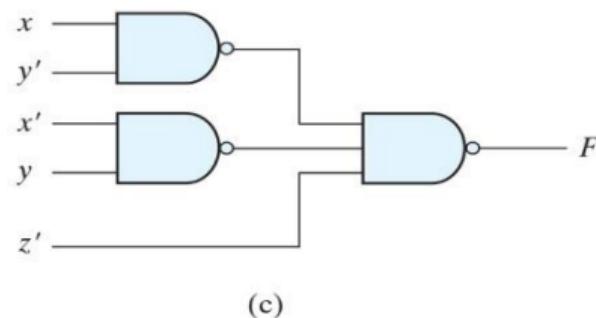
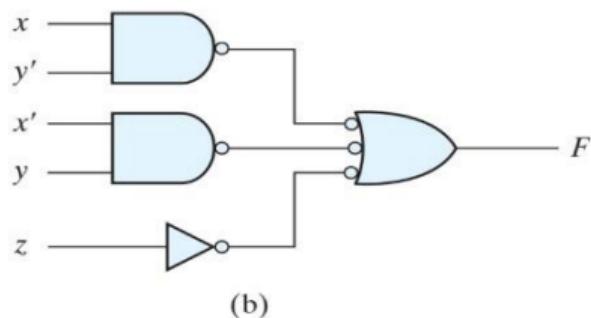
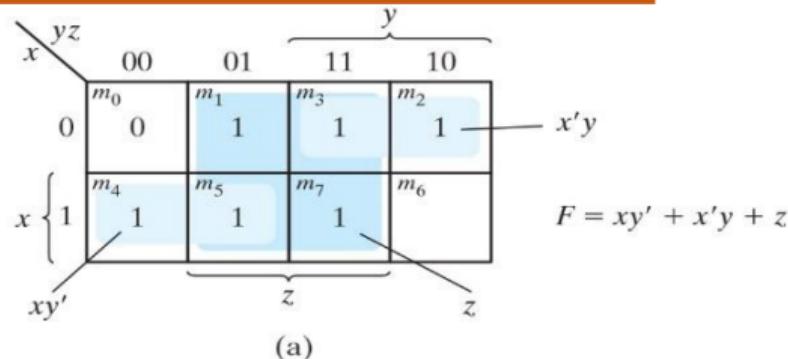
(c)

Three ways to implement  $F = AB + CD$ .

Implement the following Boolean function with NAND gates:

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

# Gate-Level Minimization and Combinational logic NAND and NOR implementation



Solution to Solved Example

# Gate-Level Minimization and Combinational logic NAND and NOR implementation



The procedure for obtaining the logic diagram from a Boolean function is as follows:

1. Simplify the function and express it in sum-of-products form.
2. Draw a NAND gate for each product term of the expression that has at least two literals. The inputs to each NAND gate are the literals of the term. This procedure produces a group of first-level gates.
3. Draw a single gate using the AND-invert or the invert-OR graphic symbol in the second level, with inputs coming from outputs of first-level gates.
4. A term with a single literal requires an inverter in the first level. However, if the single literal is complemented, it can be connected directly to an input of the second level NAND gate

# Gate-Level Minimization and Combinational logic

## NAND and NOR implementation

### (multilevel NAND gate)



The standard form of expressing Boolean functions results in a two-level implementation. There are occasions, however, when the design of **digital systems results in gating structures with three or more levels**

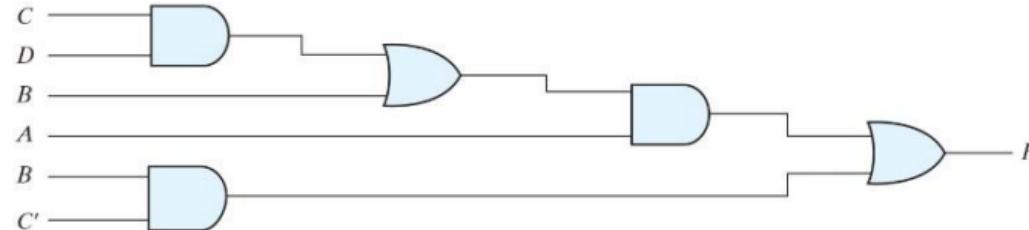
The most common procedure in the design of multilevel circuits is to express the Boolean function in terms of AND, OR, and complement operations. The function can then be implemented with AND and OR gates. After that, if necessary, it can be converted into an all-NAND circuit.

Implementing  $F = A(CD + B) + BC'$  using NAND gate( AND invert, Invert OR) . Implement the same using AND-OR gates

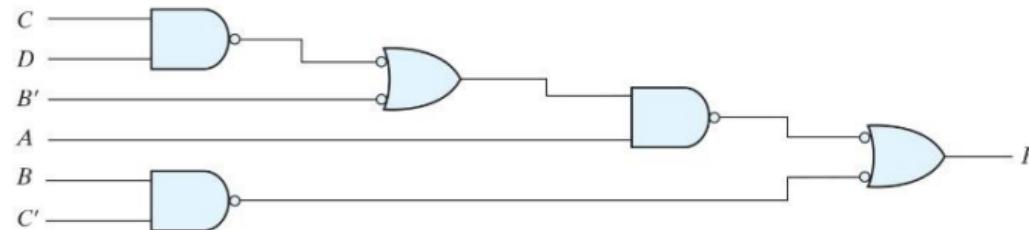
Draw the Circuit .

# Gate-Level Minimization

## NAND and NOR implementation(multilevel NAND gate)



(a) AND-OR gates



(b) NAND gates

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

# Gate-Level Minimization and Combinational logic

## NAND and NOR implementation(multilevel NAND gate)



Although it is possible to remove the parentheses and reduce the expression into a standard sum-of-products form, we choose to implement it as a multilevel circuit for illustration

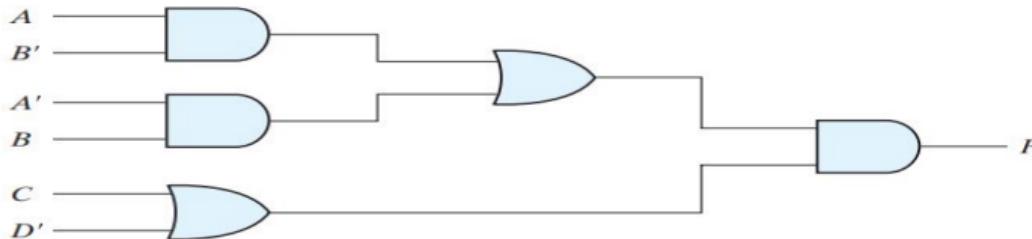
The general procedure for converting a multilevel AND–OR diagram into an all-NAND diagram using mixed notation is as follows:

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

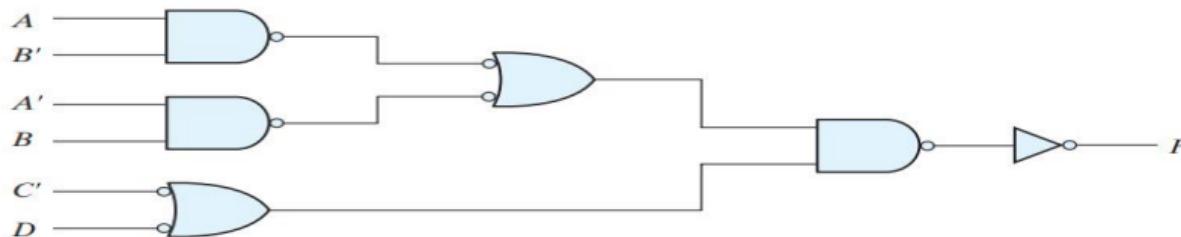
Implement  $F = (AB' + A'B)(C + D')$  using AND-OR gates, NAND gates

# Gate-Level Minimization and Combinational logic

## NAND and NOR implementation(multilevel NAND gate)



(a) AND-OR gates



(b) NAND gates

**FIGURE 3.21**

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

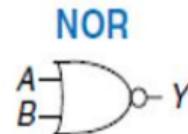
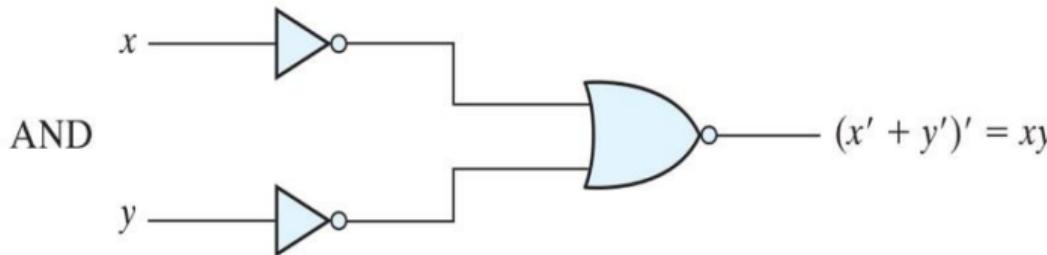
# Gate-Level Minimization and Combinational logic NOR implementation

---



The NOR operation is the dual of the NAND operation. Therefore, all procedures and rules for NOR logic are the **duals** of the corresponding procedures and rules developed for NAND logic. The NOR gate is another universal gate that can be used to implement any Boolean function

# Gate-Level Minimization and Combinational logic NOR implementation



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

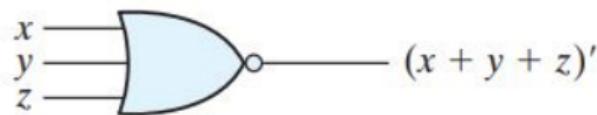
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

Logic operations with NOR gates.

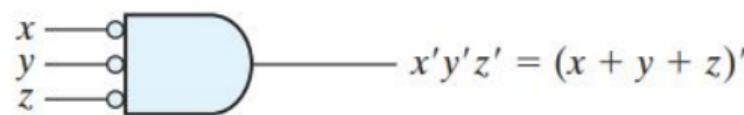
# Gate-Level Minimization and Combinational logic NOR implementation



## Logic operations with NOR gates



(a) OR-invert



(b) Invert-AND

**FIGURE 3.23**

Two graphic symbols for the NOR gate

The procedure for converting a multilevel AND-OR diagram to an all-NOR diagram. we must convert each OR gate to an OR-invert symbol and each AND gate to an invert-AND symbol

# Gate-Level Minimization and Combinational logic NOR implementation

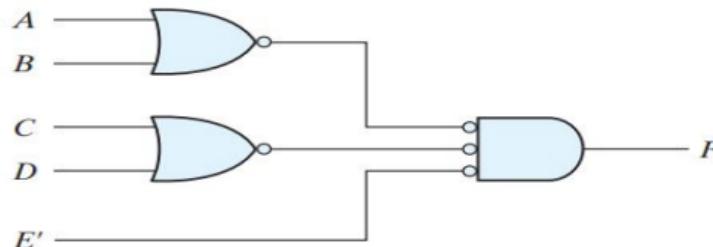
---



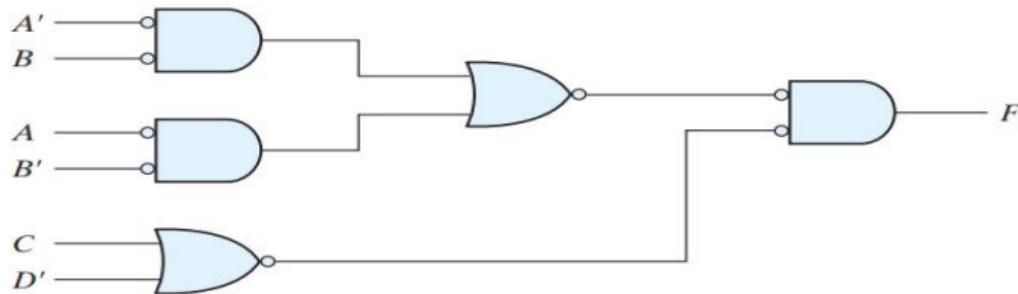
Implementing  $F = (A + B)(C + D)E$

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

# Gate-Level Minimization and Combinational logic NAND and NOR implementation



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



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

# Exclusive-OR Function (T1: 3.8)

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

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

The exclusive-OR is equal to 1 if only  $x$  is equal to 1 or if only  $y$  is equal to 1 (i.e.,  $x$  and  $y$  differ in value), but not when both are equal to 1 or when both are equal to 0. The exclusive-NOR, also known as equivalence, performs the following Boolean operation:



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

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

**XNOR**



$$Y = \overline{A \oplus B}$$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

# Exclusive-OR Function

The following identities apply to the exclusive-OR operation:

$$x \oplus 0 = x$$

$$x \oplus 1 = x'$$

$$x \oplus x = 0$$

$$x \oplus x' = 1$$

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

Any of these identities can be proven with a truth table or by replacing the  $\oplus$  operation by its equivalent Boolean expression. Also, it can be shown that the exclusive-OR operation is both commutative and associative; that is,

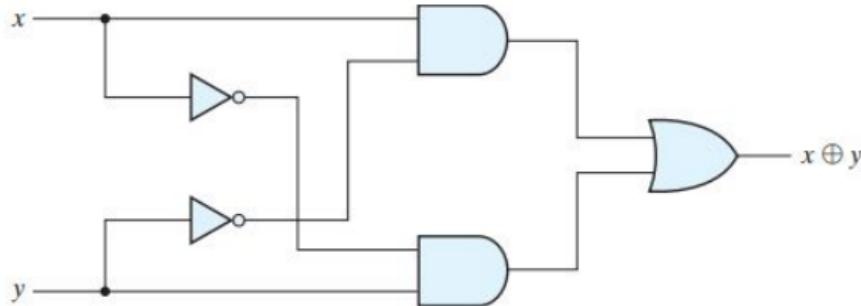
$$A \oplus B = B \oplus A$$

and

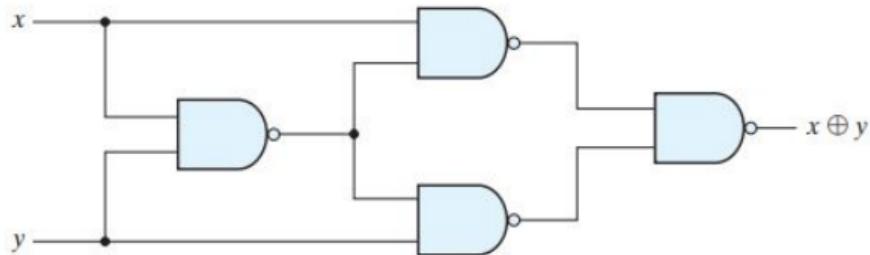
$$(A \oplus B) \oplus C = A \oplus (B \oplus C) = A \oplus B \oplus C$$

# Exclusive-OR Implementation

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



(a) Exclusive-OR with AND-OR-NOT gates



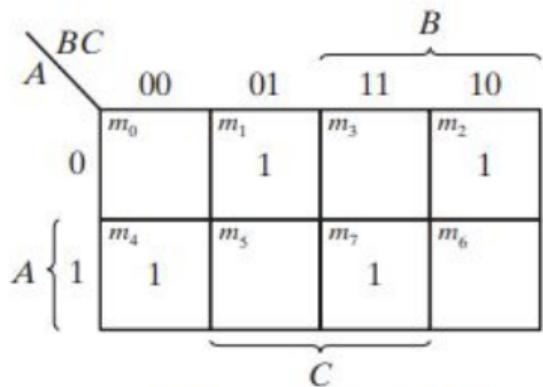
(b) Exclusive-OR with NAND gates

# Odd function - XOR for 3 variables

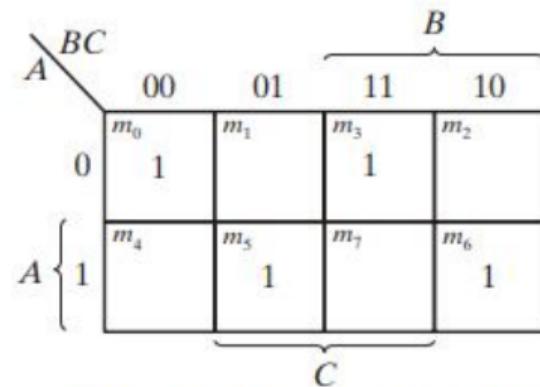
---

- Multiple-variable exclusive-OR operation is defined as an odd function.
- XOR – 3 variables: A XOR B XOR C
- The Boolean expression clearly indicates that the three-variable exclusive-OR function is equal to 1 if only one variable is equal to 1 or if all three variables are equal to 1.
- Multiple variable XOR – ODD Function: The odd function is identified from the four minterms whose binary values have an odd number of 1's- ODD parity.

# Odd function - XOR for 3 variables



(a) Odd function  $F = A \oplus B \oplus C$

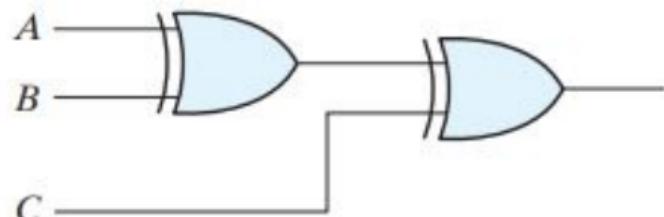


(b) Even function  $F = (A \oplus B \oplus C)'$

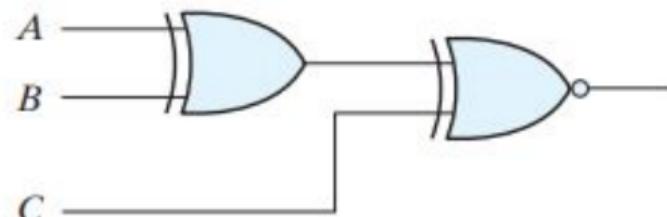
**FIGURE 3.31**

Map for a three-variable exclusive-OR function

# Odd function - XOR for 3 variables



(a) 3-input odd function



(b) 3-input even function

**FIGURE 3.32**

Logic diagram of odd and even functions

# Odd function - XOR for 4 variables

---

- There are 16 minterms for a four-variable Boolean function.
- Half of the minterms have binary numerical values with an odd number of 1's;
- The other half of the minterms have binary numerical values with an even number of 1's

# Odd function - XOR for 4 variables

$$\begin{aligned}A \oplus B \oplus C \oplus D &= (AB' + A'B) \oplus (CD' + C'D) \\&= (AB' + A'B)(CD + C'D') + (AB + A'B')(CD' + C'D) \\&= \Sigma(1, 2, 4, 7, 8, 11, 13, 14)\end{aligned}$$

		CD		C			
		00	01	11	10		
AB		m <sub>0</sub>	m <sub>1</sub>	m <sub>3</sub>	m <sub>2</sub>		
A	00	m <sub>0</sub>	1				
	01	m <sub>4</sub>	m <sub>5</sub>	m <sub>7</sub>	m <sub>6</sub>		
	11	m <sub>12</sub>	m <sub>13</sub>	m <sub>15</sub>	m <sub>14</sub>		
	10	m <sub>8</sub>	m <sub>9</sub>	m <sub>11</sub>	m <sub>10</sub>		

(a) Odd function  $F = A \oplus B \oplus C \oplus D$

		CD		C			
		00	01	11	10		
AB		m <sub>0</sub>	m <sub>1</sub>	m <sub>3</sub>	m <sub>2</sub>		
A	00	1					
	01		m <sub>5</sub>	m <sub>7</sub>	m <sub>6</sub>		
	11	m <sub>12</sub>	m <sub>13</sub>	m <sub>15</sub>	m <sub>14</sub>		
	10	m <sub>8</sub>	m <sub>9</sub>	m <sub>11</sub>	m <sub>10</sub>		

(b) Even function  $F = (A \oplus B \oplus C \oplus D)'$

# Parity Generation and Checking

- Exclusive-OR functions are very useful in systems requiring error detection and correction.
- A parity bit is an extra bit included with a binary message to make the number of 1's either odd or even. The message, including the parity bit, is transmitted and then checked at the receiving end for errors.
- The circuit that generates the parity bit in the transmitter is called a parity generator and a parity checker.



# Parity Generation and Checking

Parity bit P must be generated to make the number of 1's even if even parity is considered.

*Even-Parity-Generator Truth Table*

Three-Bit Message			Parity Bit
x	y	z	P
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

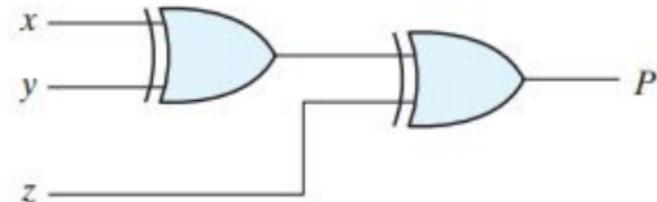
# Parity Generation and Checking

---

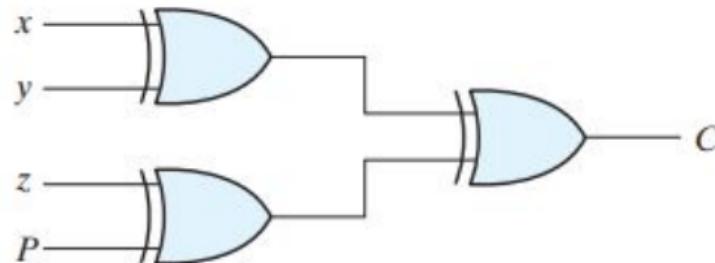
- A parity bit is an extra bit included with a binary message to make the number of 1's either odd or even. The message, including the parity bit, is transmitted and then checked at the receiving end for errors
- An error is detected if the checked parity does not correspond with the one transmitted. The circuit that generates the parity bit in the transmitter is called a parity generator. The circuit that checks the parity in the receiver is called a parity checker.

# Parity Generation and Checking

Logic diagram of a parity generator and checker.



(a) 3-bit even parity generator



(b) 4-bit even parity checker

# Parity Generation and Checking

Parity check

$$A \text{ XOR } B \text{ XOR } C \text{ XOR } P$$

Advantage- use same gates for parity generator and checker

*Even-Parity-Checker Truth Table*

<b>Four Bits Received</b>				<b>Parity Error Check</b>
<b>x</b>	<b>y</b>	<b>z</b>	<b>P</b>	<b>c</b>
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

# Applications

---

- Alarm Systems

Scenario:

Industrial machines often have safety alarms that trigger if certain unsafe conditions occur (e.g., overheat, high pressure, door open).

## Why NAND/NOR?

Safety logic can be built with a single type of universal gate:

- NAND can be configured to act as OR to trigger alarms when any unsafe condition is present.
- NAND can also be configured to act as AND to ensure alarms only trigger if multiple sensor fail simultaneously.

# Applications

---

## Processor Design (Control Logic)

Scenario: In CPUs, the control unit decides which micro-operation to perform based on opcode and status flags.

### Why NAND/NOR?

At the hardware level, all combinational control logic can be implemented using only NAND or only NOR gates — reducing design complexity in IC fabrication.

# THANK YOU

---



**Team DDCO**  
Department of Computer Science



# DIGITAL DESIGN AND COMPUTER ORGANIZATION

## Combinational Logic

---

**Team DDCO**

**Department of Computer Science and Engineering**

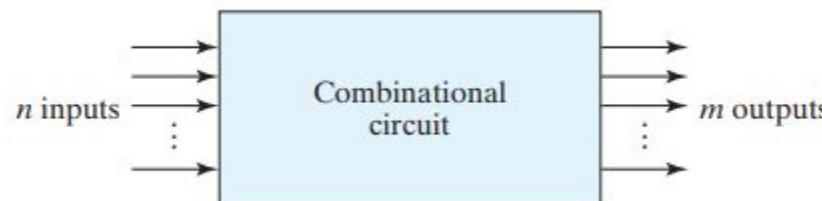
# Introduction

---

- There are two types of Logic Circuits:
  - Combinational
  - Sequential
- Combinational Circuits consists of logic gates whose output at any time is function of the **present inputs only**.
- Sequential Circuits along with logic gates use storage elements. The value stored in these elements is due to a previous combination of inputs. Thus the present output of a sequential circuits depends upon **the present and the past inputs**.

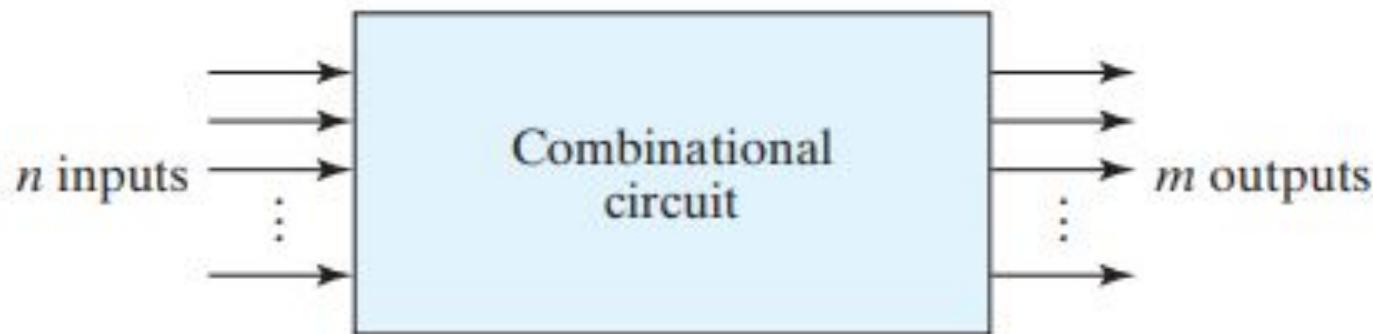
# Combinational Circuits

- A combinational circuit consists of input variables, logic gates and output variables.
- For  $n$  input variables there will be a total  **$2^n$  input combinations** and the pattern of  $m$  outputs w.r.t the different combination can be listed in the form of a truth table.
- A combinational circuit can be represented by a truth table having  **$n$  inputs &  $m$  outputs** or it can be represented by a Boolean function having  $m$  equations one for each output variable.

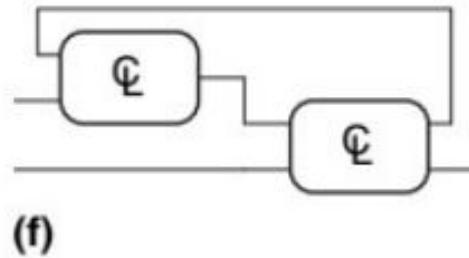
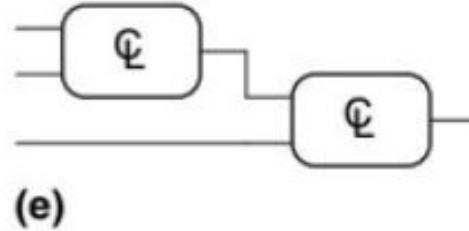
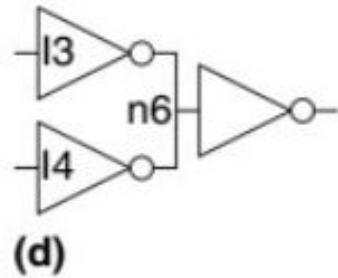
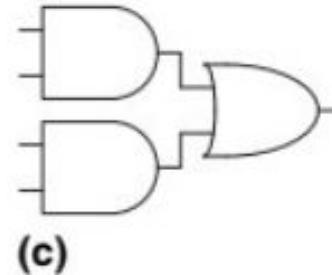
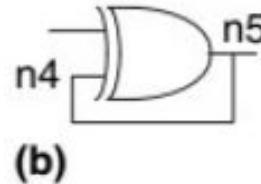
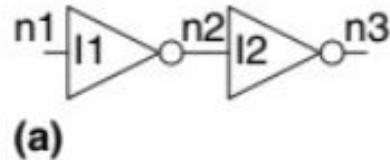


# Combinational Circuits

Block diagram of combinational circuit



# Identify the Circuit



# Identify the Circuit

---

- A. is **combinational**. It is constructed from two combinational circuit elements (inverters I1 and I2). It has three nodes: n1, n2, and n3. n1 is an input to the circuit and to I1; n2 is an internal node, which is the output of I1 and the input to I2; n3 is the output of the circuit and of I2.
- B. is **not combinational**, because there is a cyclic path: the output of the XOR feeds back to one of its inputs.
- C. is **combinational**.
- D. is **not combinational**, because node n6 connects to the output terminals of both I3 and I4.
- E. is **combinational**, illustrating two combinational circuits connected to form a larger combinational circuit.
- F. is **not combinational**, does not obey the rules of combinational composition because it has a cyclic path through the two elements.

# Analysis Procedure

---

- The analysis starts with a given logic diagram and culminates with
  - **A set of Boolean functions**
  - **A Truth Table**
  - **or, possibly, an explanation of the circuit operation**
- The first step in the analysis is to make sure that the given circuit is **combinational and not sequential.**
- The diagram of a **combinational** circuit has logic gates with **no feedback paths or memory elements.**
- A feedback path is a connection from the output of one gate to the input of a second gate whose output forms part of the input to the first gate.

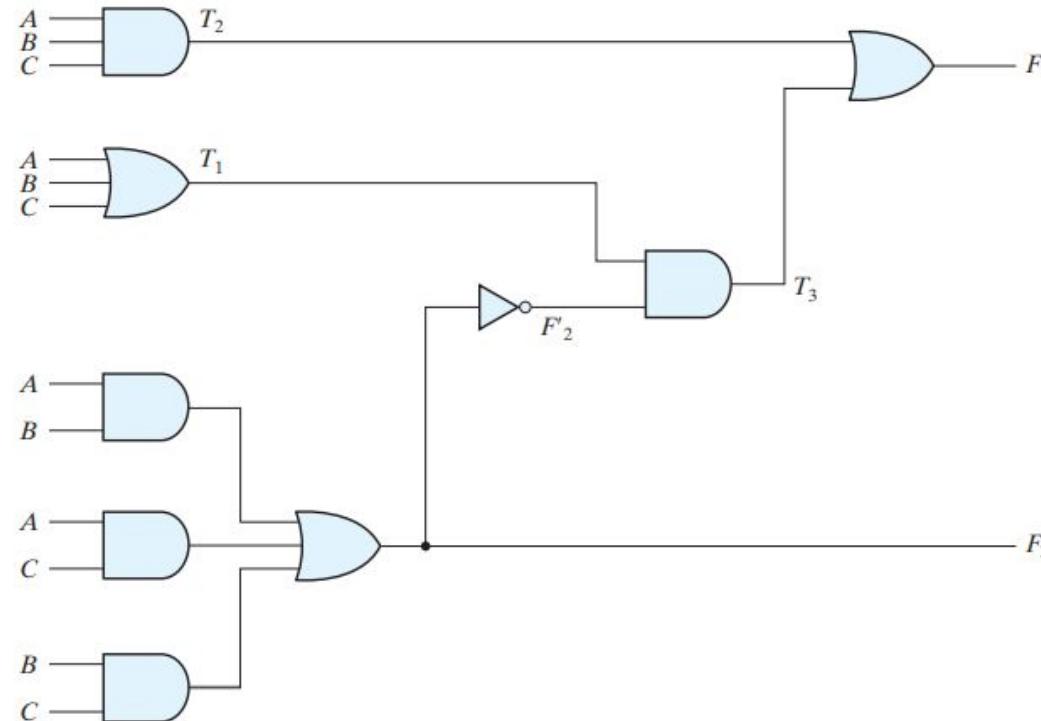
# Analysis Procedure

---

To obtain the output Boolean functions from a logic diagram, we proceed as follows:

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

# Analysis Procedure



**FIGURE 4.2**  
Logic diagram for analysis example

# Analysis Procedure

---

The analysis of the combinational circuit of Fig. 4.2 illustrates the proposed procedure. We note that the circuit has three binary inputs— $A$ ,  $B$ , and  $C$ —and two binary outputs— $F_1$  and  $F_2$ . The outputs of various gates are labeled with intermediate symbols. The outputs of gates that are a function only of input variables are  $T_1$  and  $T_2$ . Output  $F_2$  can easily be derived from the input variables. The Boolean functions for these three outputs are

$$F_2 = AB + AC + BC$$

$$T_1 = A + B + C$$

$$T_2 = ABC$$

Next, we consider outputs of gates that are a function of already defined symbols:

$$T_3 = F'_2 T_1$$

$$F_1 = T_3 + T_2$$

To obtain  $F_1$  as a function of  $A$ ,  $B$ , and  $C$ , we form a series of substitutions as follows:

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

# Analysis Procedure

---

The derivation of the truth table for a circuit is a straightforward process once the output Boolean functions are known. To obtain the truth table directly from the logic diagram without going through the derivations of the Boolean functions, we proceed as follows:

1. Determine the number of input variables in the circuit. For  $n$  inputs, form the  $2^n$  possible input combinations and list the binary numbers from 0 to  $(2^n - 1)$  in a table.
2. Label the outputs of selected gates with arbitrary symbols.
3. Obtain the truth table for the outputs of those gates which are a function of the input variables only.
4. Proceed to obtain the truth table for the outputs of those gates which are a function of previously defined values until the columns for all outputs are determined.

# Analysis Procedure

**Table 4.1**  
*Truth Table for the Logic Diagram of Fig. 4.2*

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

# Design Procedure

---

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

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

# Code Conversion Example

---

It is sometimes necessary to use the **output of one system as the input to another**. A conversion circuit must be inserted between the two systems if each uses different codes for the same information. Thus, a code converter is a circuit that makes the two systems compatible even though each uses a different binary code

## Code Conversion Example

*Design a circuit that converts from a BCD code to excess-3 code.*

# Code Conversion Example

---

- Some circuits use excess-3 code and we need to feed this circuit with the right input.
- In excess-3 code, numbers are represented as decimal digits, and each digit is represented by four bits as the digit value plus 3

BCD stands for **Binary Coded Decimal**. Binary coded decimal (BCD) is a system of writing numerals that assigns a four-digit binary code to each digit 0 through 9 in a decimal (base-10) numeral.

BCD code is also known as the 8 4 2 1 code.

In truth table 10-15 is don't care output.

# Code Conversion Example

*Truth Table for Code Conversion Example*

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

# Code Conversion Example

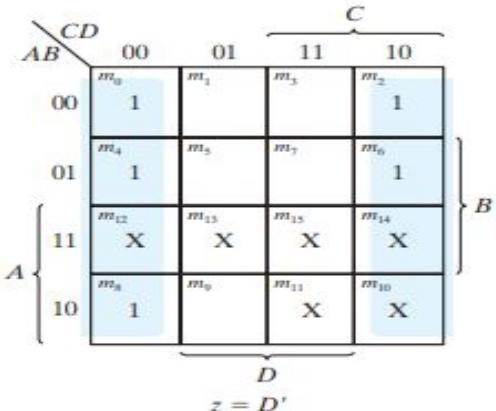
Maps for BCD-to-excess-3 code converter

$$z = D'$$

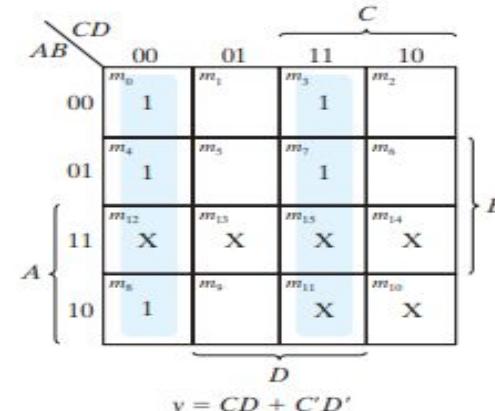
$$y = CD + C'D' = CD + (C + D)'$$

$$\begin{aligned} x &= B'C + B'D + BC'D' = B'(C + D) + BC'D' \\ &= B'(C + D) + B(C + D)' \end{aligned}$$

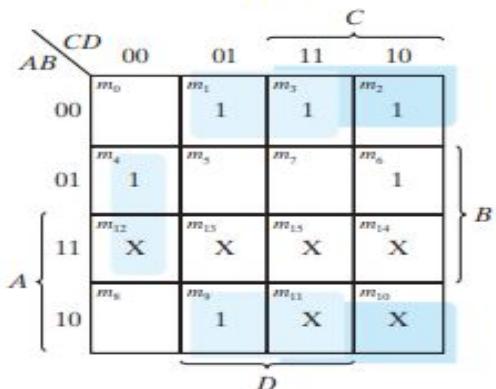
$$w = A + BC + BD = A + B(C + D)$$



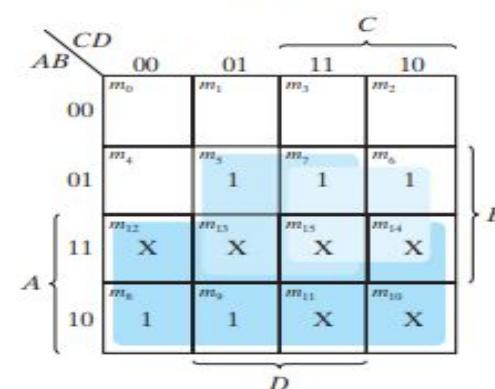
$$x = B'C + B'D + BC'D'$$



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

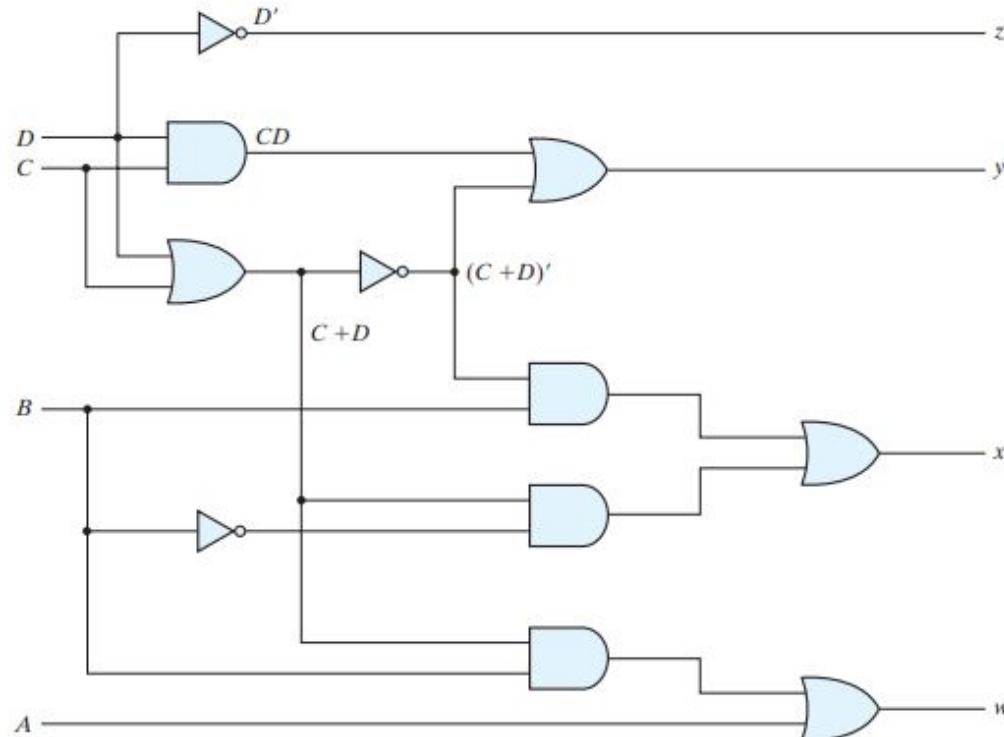


$$x = B'C + B'D + BC'D'$$



$$w = A + BC + BD$$

# Code Conversion Example



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

# Code Conversion Example

---

Not counting input inverters, the implementation in sum-of-products form requires seven AND gates and three OR gates (without simplification).

The implementation of the figure in the previous slide requires four AND gates, four OR gates, and one inverter (after simplification)

Thus, the three-level logic circuit requires fewer gates, all of which in turn require no more than two inputs.

# 8 4 -2 -1 Code and BCD Code

	8	4	-2	-1	BCD Code			
Dec.	A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	0	0
1	0	1	1	1	0	0	0	1
2	0	1	1	0	0	0	1	0
3	0	1	0	1	0	0	1	1
4	0	1	0	0	0	1	0	0
5	1	0	1	1	0	1	0	1
6	1	0	1	0	0	1	1	0
7	1	0	0	1	0	1	1	1
8	1	0	0	0	1	0	0	0
9	1	1	1	1	1	0	0	1

# Gray code

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

# Think about it

---

In Gray code, **only one bit changes at a time compared to binary**. What is the **Gray code equivalent** of the binary number 1101?

- A) 1111
- B) 1011
- C) 1000
- D) 1010

# Think about it

---

In Gray code, **only one bit changes at a time** compared to binary. What is the **Gray code equivalent** of the binary number 1101?

- A) 1111
- B) 1011
- C) 1000
- D) 1010

**Answer:** B) 1011

**Explanation:**

Binary bit position	Binary bit	Rule	Gray bit
1 (MSB)	1	Copy as-is	1
2	1	$1 \oplus 1 = 0$	0
3	0	$1 \oplus 0 = 1$	1
4 (LSB)	1	$0 \oplus 1 = 1$	1

# Think about it

---

A combinational logic circuit is designed to convert a **4-bit BCD input** into **Gray code**. If the BCD input is restricted to digits 0–9, how many valid Gray code outputs are expected?

- A) 10
- B) 15
- C) 9
- D) 16

# Think about it

---

A combinational logic circuit is designed to convert a **4-bit BCD** input into **Gray code**. If the BCD input is restricted to digits 0–9, how many valid Gray code outputs are expected?

- A) 10
- B) 15
- C) 9
- D) 16

**Answer:** A) 10

**Explanation:**

Since BCD digits range from 0000 to 1001 (0 to 9), only **10 valid inputs** exist. So, the circuit will produce 10 valid Gray code outputs corresponding to these.

# Think about it

---

In a BCD to Excess-3 code converter, what is the output for the BCD input **0101 (5)**?

- A) 1000
- B) 0110
- C) 1010
- D) 1100

# Think about it

---

In a BCD to Excess-3 code converter, what is the output for the BCD input **0101 (5)**?

- A) 1000
- B) 0110
- C) 1010
- D) 1100

**Answer:** A) 1000

**Explanation:**

Excess-3 code = BCD + 0011.

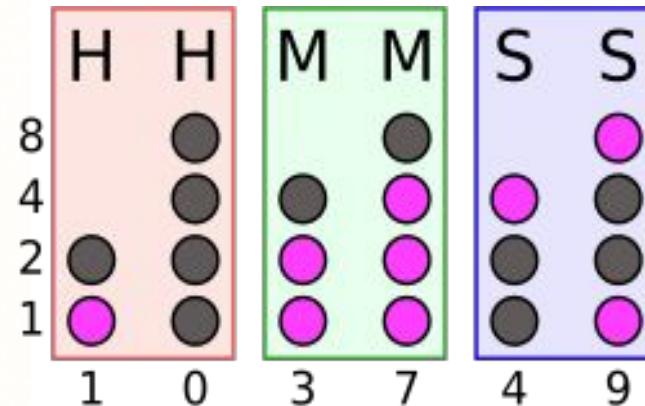
$\text{BCD}(0101) + 0011 = 1000 \text{ (8)}$ .

So, the output is **1000**.

# Applications

## Code Conversion or Real Device

Code Conversion	Real Device
Binary to Gray	 Rotary Encoder in a Mouse or Robot Wheel
Decimal to BCD	 Digital Clock
BCD to Excess-3	 Old Electronic Calculators
Binary to ASCII	 Keyboard Typing
Binary to Parity	 Pen Drive Data Transfer



10:37:49

# THANK YOU

---

**Team DDCO**  
**Department of Computer Science and Engineering**



# DIGITAL DESIGN AND COMPUTER ORGANIZATION

**Binary Adder Subtractor,  
Decimal Adder**

---

**Team DDCO**  
**Department of Computer Science and Engineering**

# DIGITAL DESIGN AND COMPUTER ORGANIZATION

---



## Adder Subtractor Decimal Adder

Department of Computer Science and Engineering

### Binary Adder-Subtractor

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

### Binary Adder

- Simple addition consists of four possible elementary operations:

$$0+0 = 0$$

$$0+1 = 1$$

$$1+0 = 1$$

$$1+1 = 10 \leftarrow \text{Sum has Carry}$$

Half Adder (two '2' input bits)

Full Adder (three '3' input bits)

# Combinational logic

## Adder Subtractor

---

### Half Adder

- The simplified Boolean functions for the two outputs can be obtained directly from the truth table. The simplified sum-of-products expressions are

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

$$C = xy$$

# Combinational logic

## Adder Subtractor

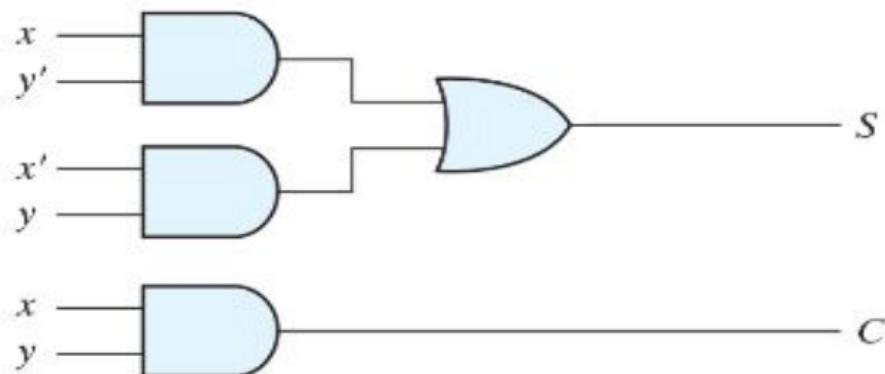
Truth Table of Half Adder

<b>x</b>	<b>y</b>	<b>c</b>	<b>s</b>
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

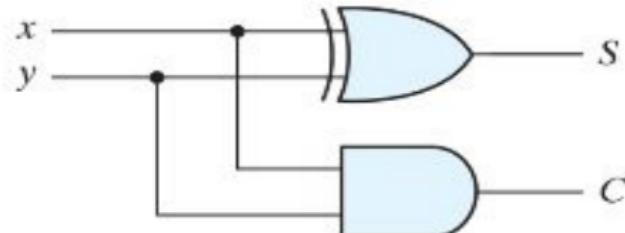
# Combinational logic

## Adder Subtractor

Implementation of half adder.



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



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

# Combinational logic

## Adder Subtractor

---

### Full Adder

- A full adder is a combinational circuit that forms the arithmetic sum of three bits. It consists of three inputs and two outputs. Two of the input variables, denoted by  $x$  and  $y$ , represent the two significant bits to be added.

# Combinational logic

## Adder Subtractor

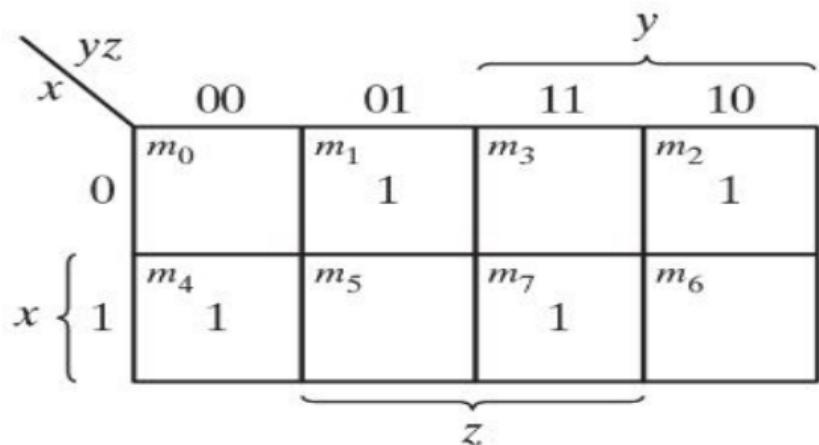
### Truth Table of Full Adder

<b>x</b>	<b>y</b>	<b>z</b>	<b>c</b>	<b>s</b>
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

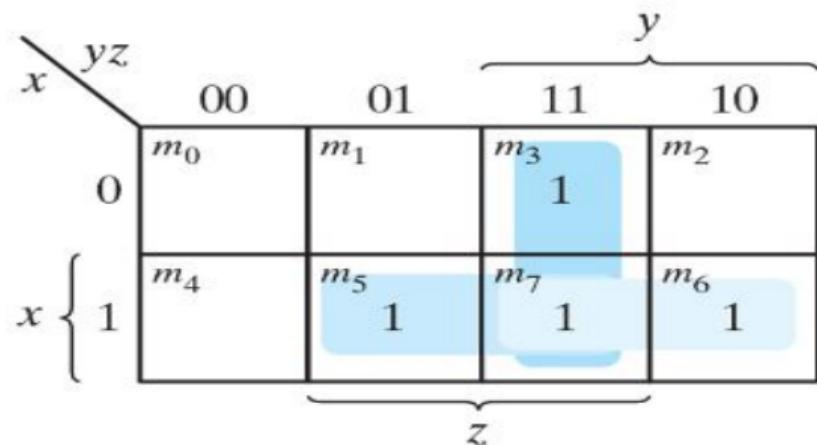
# Combinational logic

## Adder Subtractor

K-map for full adder.



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



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

## Adder Subtractor

### Full Adder

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

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

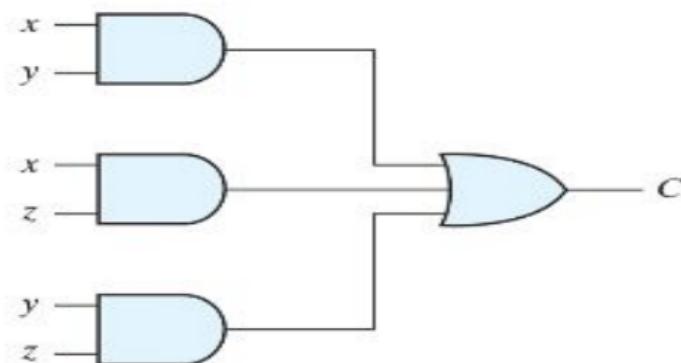
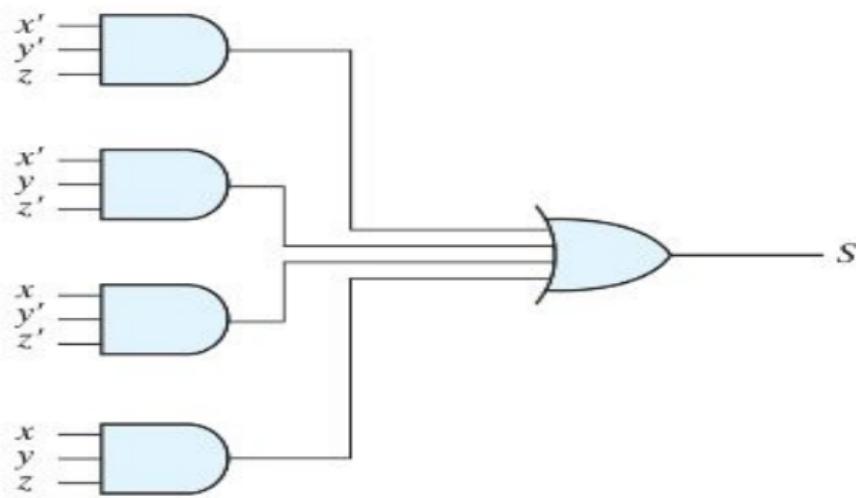
The carry output is

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

# Combinational logic

## Adder Subtractor

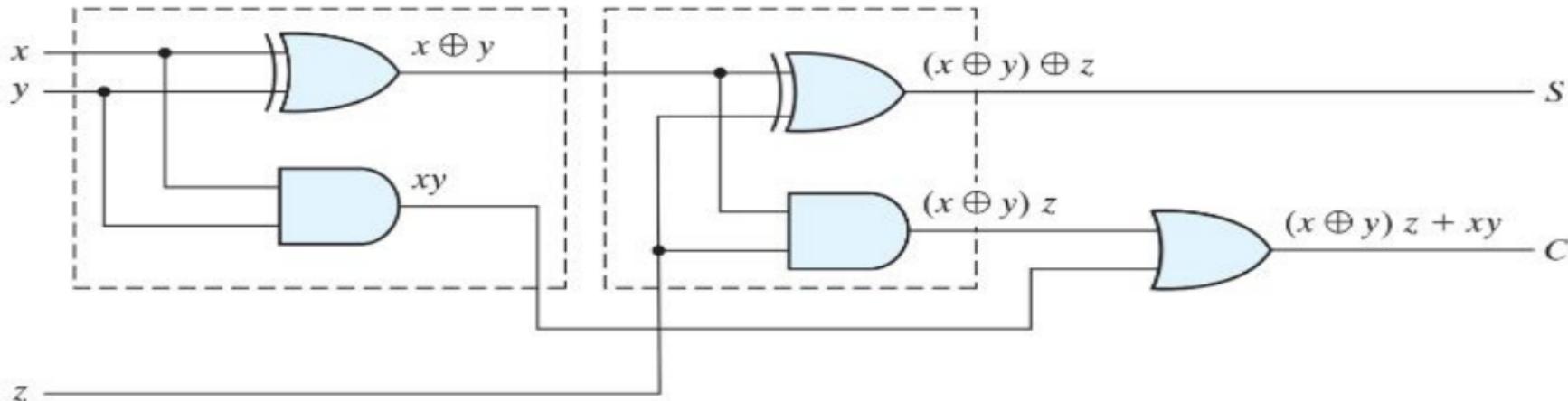
Implementation of full adder in sum-of-products form.



# Combinational logic

## Adder Subtractor

Implementation of full adder with two half adders and an OR



# Combinational logic

## Adder Subtractor

---

### Full Adder

- Full adders are used in the Arithmetic Logic Unit (ALU) of processors to perform binary addition in devices like computers, smartphones, and calculators.

### Binary Adder

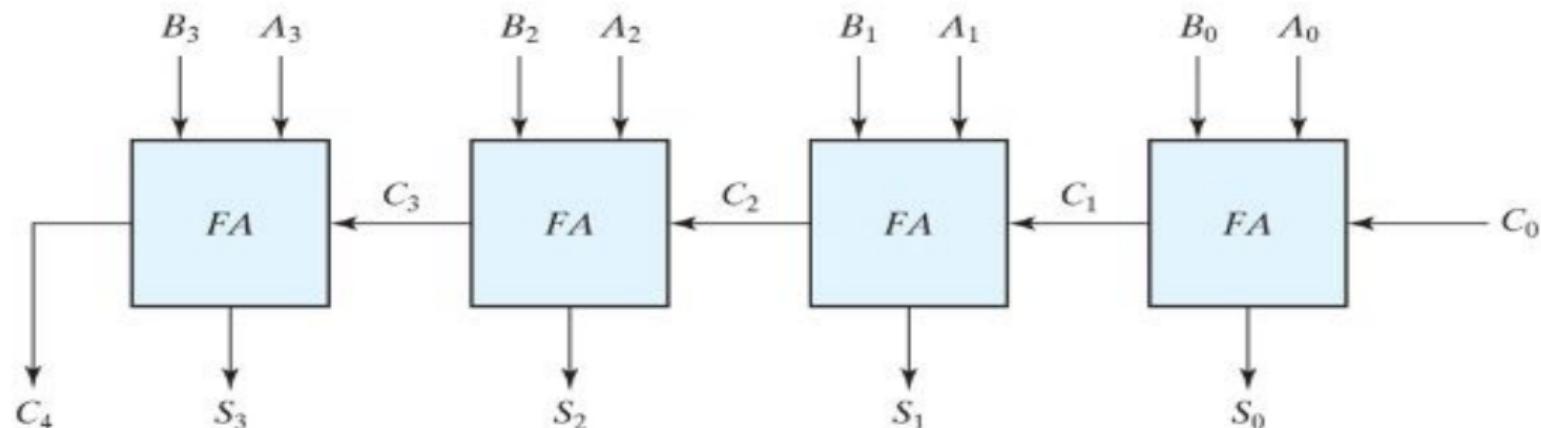
A binary adder is a digital circuit that produces the arithmetic sum of two binary numbers. It can be constructed with full adders connected in cascade, with the output carry from each full adder connected to the input carry of the next full adder in the chain

Addition of n-bit numbers requires a chain of n full adders or a chain of one-half adder and  $n - 1$  full adders.

# Combinational logic

## Adder Subtractor

Implementation of Four-bit binary adder can be implemented by using 4 full adders  $2^9 = 512$  entries (classical method, by using adders it can be avoided)



## Adder Subtractor

### Binary Adder

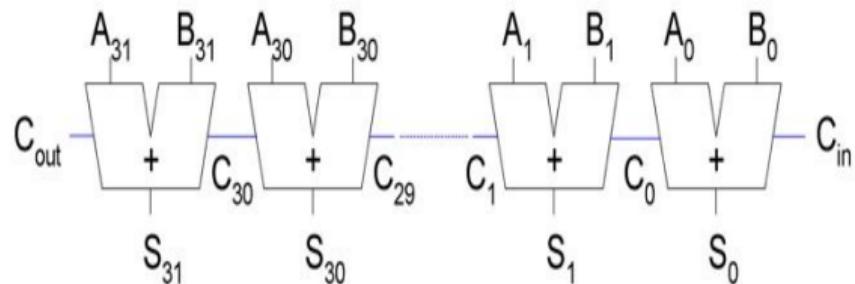
- Consider a 4-bit adder. Let's denote  $A_i$  and  $B_i$  as the input bits of the two binary numbers and  $C_i$  is the carry from previous bit addition of the numbers.
- We also let  $S_i$  be the sum and  $C_{i+1}$  the carry output in the current addition.

<b>Subscript <math>i</math>:</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>0</b>	
Input carry	0	1	1	0	$C_i$
Augend	1	0	1	1	$A_i$
Addend	0	0	1	1	$B_i$
Sum	1	1	1	0	$S_i$
Output carry	0	0	1	1	$C_{i+1}$

# Combinational logic

## Ripple Carry Adder

- Chain 1-bit adders together
- Carry ripples through entire chain
- Disadvantage: **slow**
- In ripple carry adders, for each adder block, the two bits that are to be added are available instantly.
- However, each adder block waits for the carry to arrive from its previous block.



## Ripple Carry Adder

---

- The Cout of one stage acts as the Cin of the next stage
- The ith block waits for the (i-1) th block to produce its carry.
- So there will be a considerable time delay which is carry propagation delay.
- The propagation time is equal to the propagation delay of each adder block, multiplied by the number of adder blocks in the circuit.
- Time complexity:

$$t_{\text{ripple}} = N t_{\text{FA}}$$

where  $t_{\text{FA}}$  is the delay of a  
1-bit full adder

For example, if each full adder stage has a propagation delay of 20 nanoseconds, then will reach its final correct value after 80 ( $20 \times 4$ ) nanoseconds.

The situation gets worse, if we extend the number of stages for adding more number of bits.

The ripple carry adder has the disadvantage of being slow when  $N$  is large

Ripple-carry addition requires  $\Theta(n)$  time because of the rippling of carry bits through the circuit.

Instead of generating and propagating carry bit-by-bit, **can we generate all of them in parallel** and break the sequential chain?

CLA is another type of carry propagate adder that solves this problem by **dividing the adder into *blocks*** and providing circuitry to quickly determine the carry out of a block as soon as the carry in is known.

Thus it is said to **look ahead** across the blocks rather than waiting to ripple through all the full adders inside a block

In Ripple Carry Adder, each full adder has to wait for its carry-in from its previous stage full adder.

Thus,  $n^{\text{th}}$  full adder has to wait until all **(n-1) full adders** have completed their operations.

This causes a delay and makes ripple carry adder extremely slow.

**The situation becomes worst when the value of n becomes very large.**

To overcome this disadvantage, Carry Look Ahead Adder comes into play

**Carry Look Ahead Adder is an improved version of the ripple carry adder.**

It generates the carry-in of each full adder simultaneously without causing any delay.

## Carry Propagation

---

- In addition of two binary numbers in parallel implies that all the bits of the augend and addend are available for computation at the same time.
- The total propagation time is equal to the propagation delay of a typical gate, times the number of gate levels in the circuit.
- Carry propagation time is an important attribute of the adder because it limit the speed with which two numbers are added. Consider the circuit of the full adder

# Combinational logic

## Carry Propagation

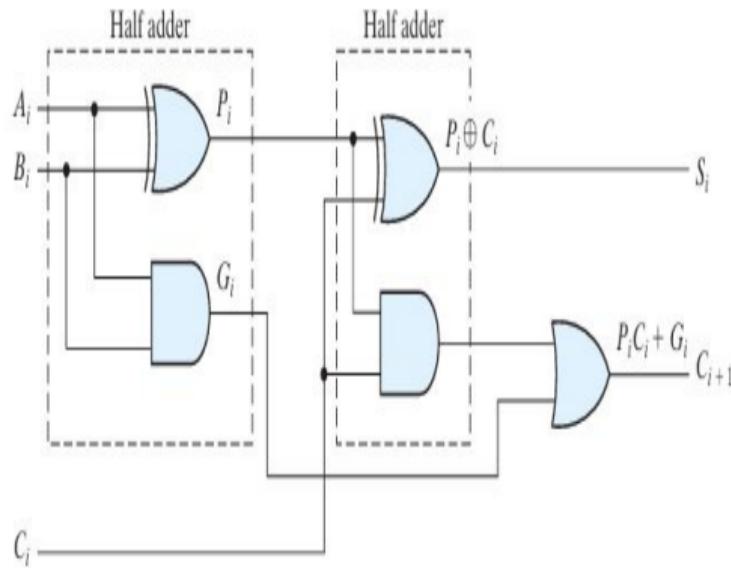


FIGURE 4.10

Full adder with  $P$  and  $G$  shown

- A block is said to generate a carry if it produces a carry out independent of the carry in to the block
- Column  $i$  will generate a carry out if  $A_i$  AND  $B_i$  are both 1.  
$$G_i = A_i \cdot B_i$$
- $G_i$  produces the carry when both  $A_i$ ,  $B_i$  are 1 regardless of the input carry.
- The block is said to propagate a carry if it produces a carry out whenever there is a carry in to the block
- The  $i$ th column will propagate a carry  $C_i$ , if either  $A_i$  or  $B_i$  is 1.  
Thus,  $P_i = A_i \oplus B_i$ .

## Carry Propagation

---

Full adder with P and G shown.

These two signals are common to all half adders and depend on only the input augend and addend bits. The signal from the input carry  $C_i$  to the output carry  $C_{i+1}$  propagates through an AND gate and an OR gate, which constitute two gate levels. If there are four full adders in the adder, the output carry  $C_4$  would have  $2 * 4 = 8$  gate levels from  $C_0$  to  $C_4$ . For an  $n$ -bit adder, **there are  $2 n$  gate levels for the carry to propagate from input to output.**

- Consider the circuit of the full adder. If we define two new binary variables

$$P_i = A_i \oplus B_i$$

$$G_i = A_i B_i$$

output sum and carry can respectively be expressed as

$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + P_i C_i$$

- $P_i$ =carry propagate
- $G_i$ =Carry generate
- Carry lookahead logic-most widely used technique to reduce the carry delay time

### Carry Propagation

$C_0$  = input carry

$C_1 = G_0 + P_0C_0$

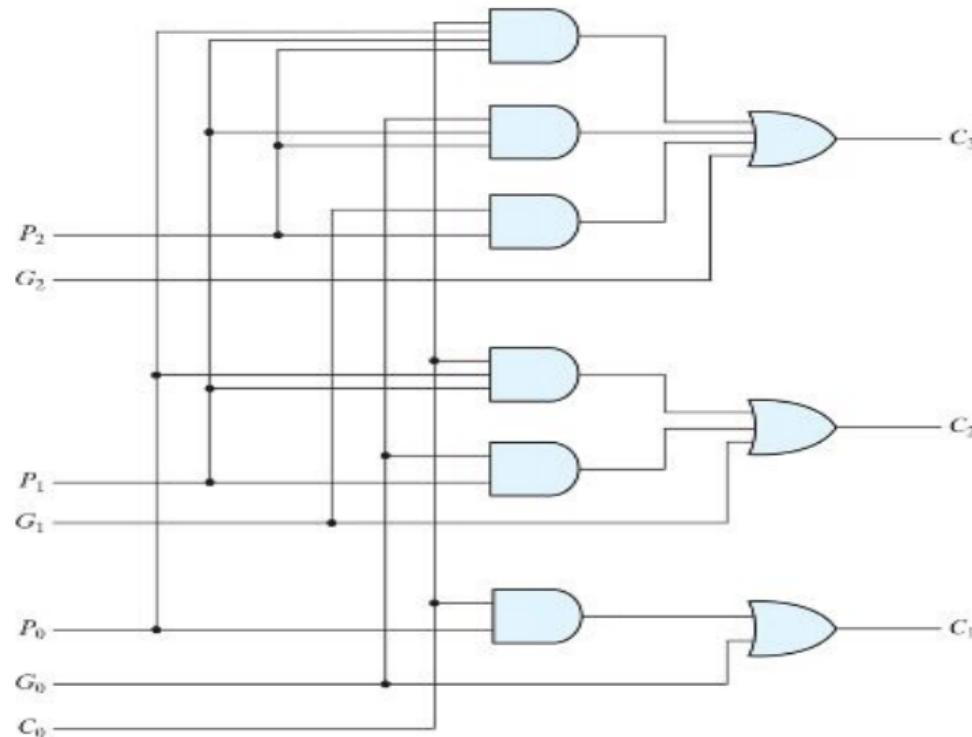
$C_2 = G_1 + P_1C_1 = G_1 + P_1(G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0$

$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 = P_2P_1P_0C_0$

# Combinational logic

## Carry Propagation

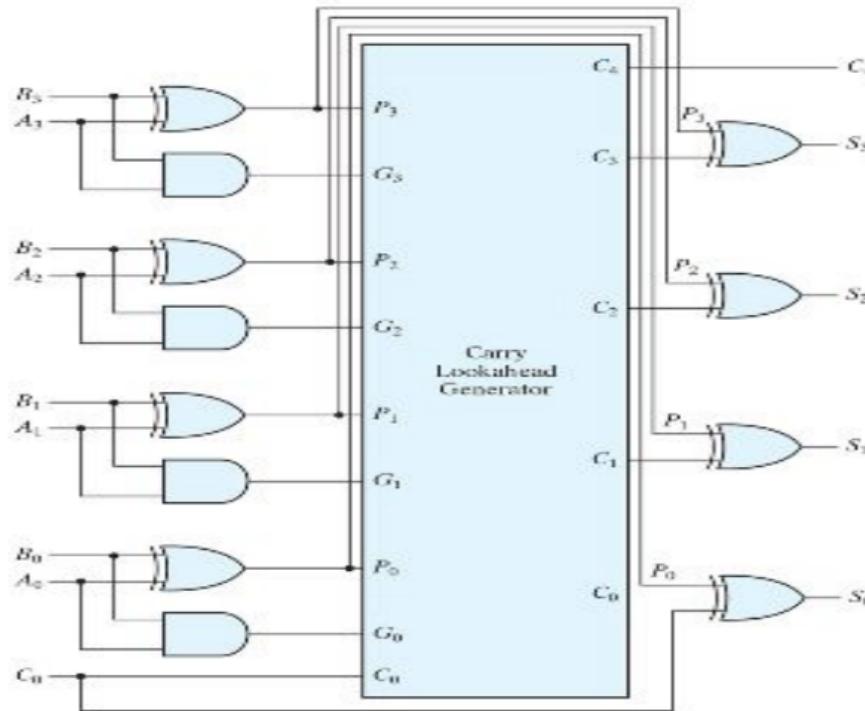
### Logic diagram of carry lookahead generator



# Combinational logic

## Carry Propagation

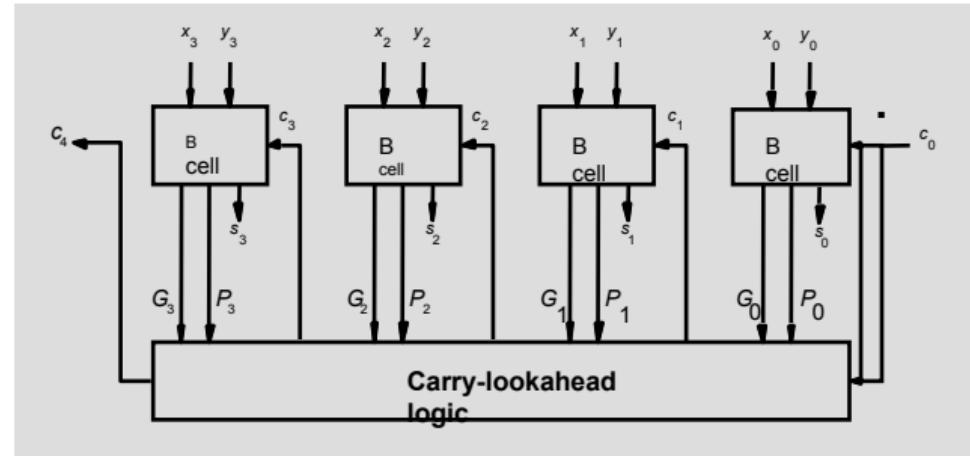
Four-bit adder with carry lookahead.



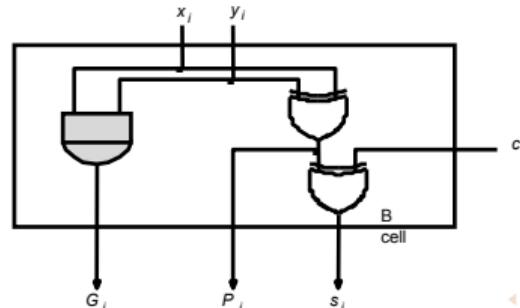
# Combinational logic

## Carry Look Ahead Adder

### 4 bit Carry-lookahead adder



4-bit  
carry-lookahead  
adder

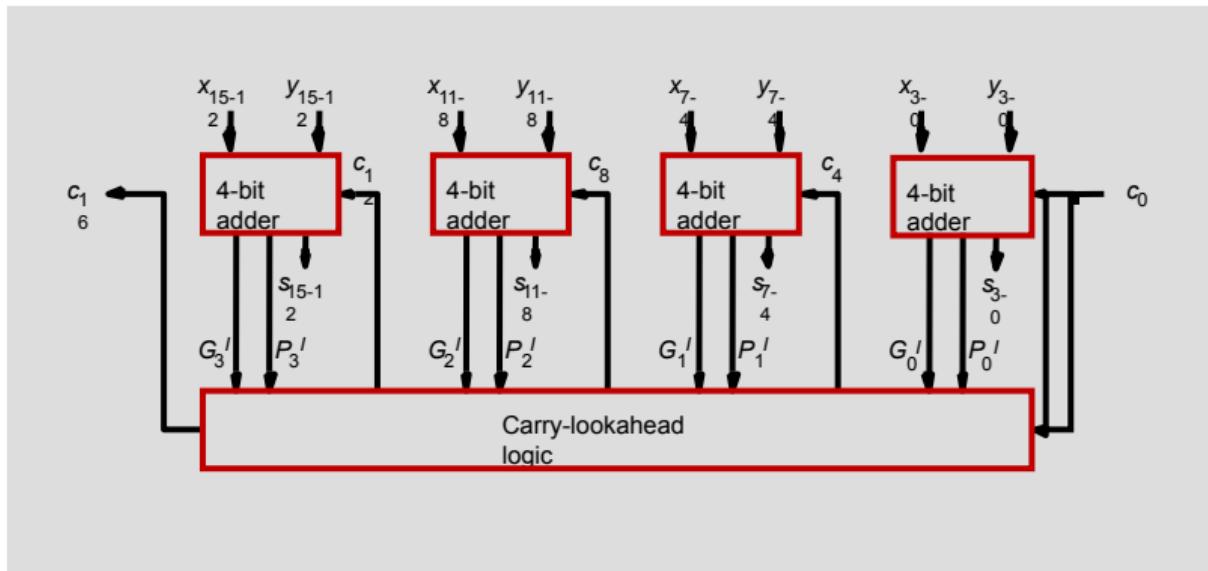


B-cell for a single stage

# Combinational logic

## Carry Look Ahead Adder

### 16-bit Block Carry-Look ahead adder



### Binary Subtractor

- Binary Subtraction is the process of finding the difference between two binary numbers. Instead of directly subtracting, it is implemented by adding the 2's complement of the subtrahend to the minuend
- This method allows the same hardware circuit to perform both addition and subtraction, making it efficient and widely used in digital computers and processors.

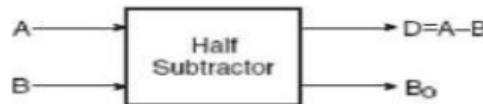
# Combinational logic

## Adder Subtractor

### Binary Subtractor

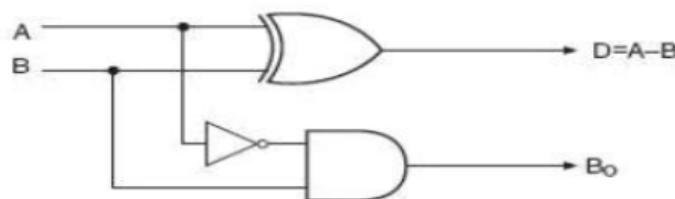
$$D = \overline{A} \cdot B + A \cdot \overline{B}$$

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



A	B	D	B <sub>o</sub>
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Half Subtractor

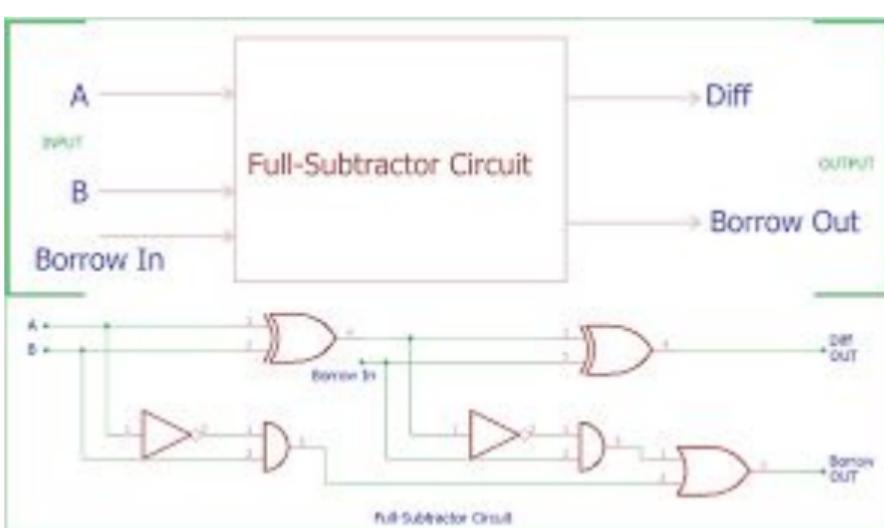


# Combinational logic

## Adder Subtractor

### Full subtractor

Inputs			Outputs	
A	B	Borrow <sub>in</sub>	Diff	Borrow
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1



### Adder-subtractor circuit

$A - B$  is equal to  $A + 2\text{'s complement of } B = a + (1\text{'s complement of } B + 1)$

For unsigned numbers, that gives  $A - B$  if  $A \geq B$  or the 2's complement of  $(B - A)$  if  $A < B$ . For signed numbers, the result is  $A - B$ , provided that there is no overflow.

# ADDER, SUBTRACTOR, OVERFLOW - 3

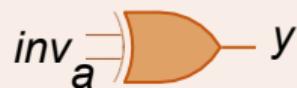
## Two's Complement Adder / Subtractor

### XOR as Controlled Inverter

- Truth table:

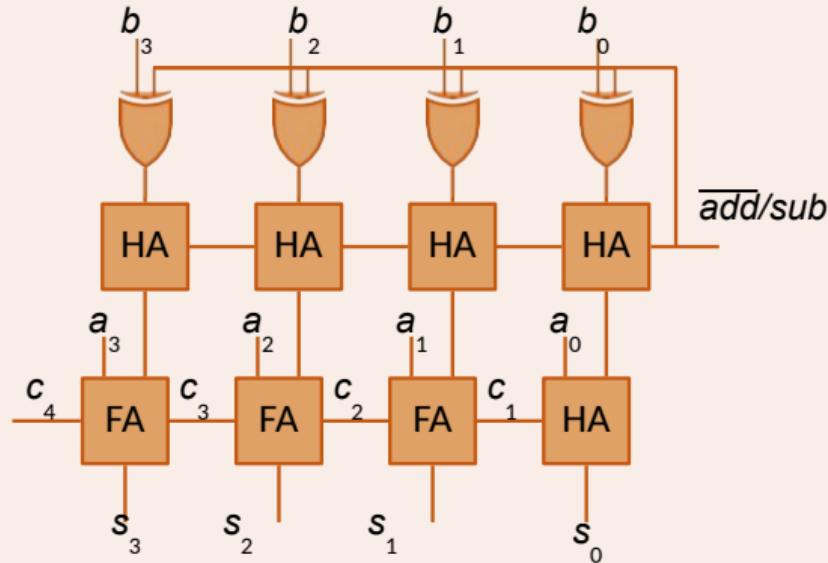
inv	a	y
0	0	0
0	1	1
1	0	1
1	1	0

- Symbol:



- When  $inv = 0$ ,  $y = a$
- When  $inv = 1$ ,  $y = \bar{a}$

### Two's Complement Adder / Subtractor

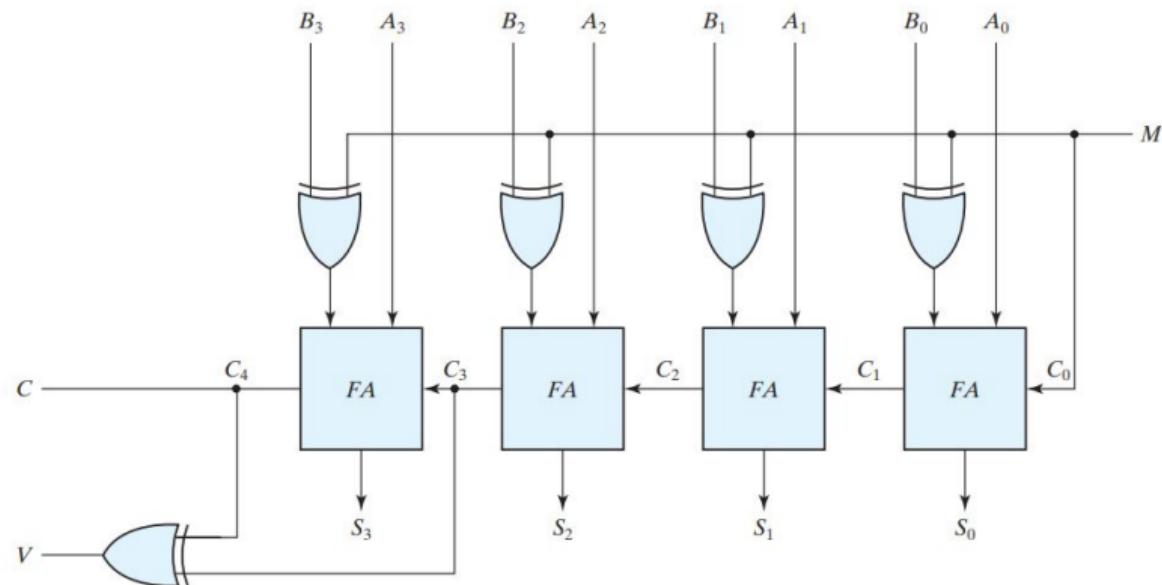


- $add/sub$  denotes: \_\_\_\_\_

} Addition when  $add/sub = 0$   
Subtraction when  $add/sub = 1$

# Combinational logic

## Adder Subtractor



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

### Overflow condition

When two numbers with n digits each are added and the sum is a number occupying n + 1 digits, we say that an overflow occurred

The detection of an overflow after the addition of two binary numbers depends on whether the numbers are considered to be signed or unsigned.

When two **unsigned numbers are added, an overflow is detected from the end carry out of the most significant position.**

EX A-B= 1001 -0110= 1\_0011

Signed numbers

MSB(leftmost bit)- sign

Negative numbers are represented in 2's complement.

Overflow will not occur if opposite sign

Overflow will occur if same sign

# Combinational logic

## Overflow Condition

carries:      0    1

$$\begin{array}{r} +70 \\ +80 \\ \hline +150 \end{array} \qquad \begin{array}{r} 0 \ 1000110 \\ 0 \ 1010000 \\ \hline 1 \ 0010110 \end{array}$$

carries:      1    0

$$\begin{array}{r} -70 \\ -80 \\ \hline -150 \end{array} \qquad \begin{array}{r} 1 \ 0111010 \\ 1 \ 0110000 \\ \hline 0 \ 1101010 \end{array}$$

+70 and +80 are eight bit numbers Range: +127 to -128

$$\begin{array}{r} 70 \quad + \ 80 = 150 - \text{not in range} \\ -70 - 80 = -150 \text{ not in range} \end{array}$$

Note that the eight-bit result that should have been positive has a negative sign bit (i.e., the eighth bit) and the eight-bit result that should have been negative has a positive sign bit. If, however, the carry out of the sign bit position is taken as the sign bit of the result, then the nine-bit answer so obtained will be correct. But since the answer cannot be accommodated within eight bits, we say that an overflow has occurred

# ADDER, SUBTRACTOR, OVERFLOW - 4

## Two's complement addition

No overflow when  
numbers have opposite signs  
(or one/both are zero)

Overflow can occur when

When both positive  
msb is 1 ( $c_{msb}$  is 1)

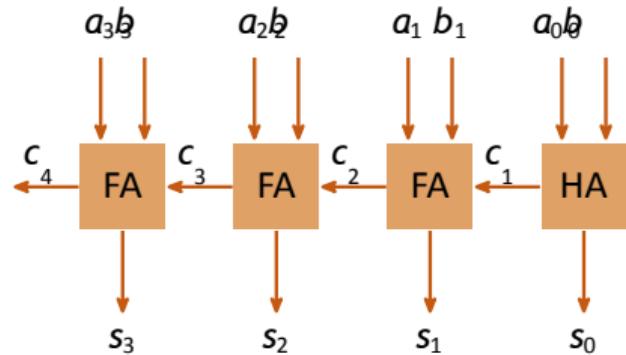
$c_{msb-1}$  is 1 and  $c_{msb}$  is 0

When both negative

msb is 0 ( $c_{msb-1}$  is 0)

$c_{msb-1}$  is 0 and  $c_{msb}$  is 1

$$\text{overflow} = c_{msb} \oplus c_{msb-1}$$



# Gate-Level Minimization and Combinational logic

## Adder Subtractor with Overflow Detection

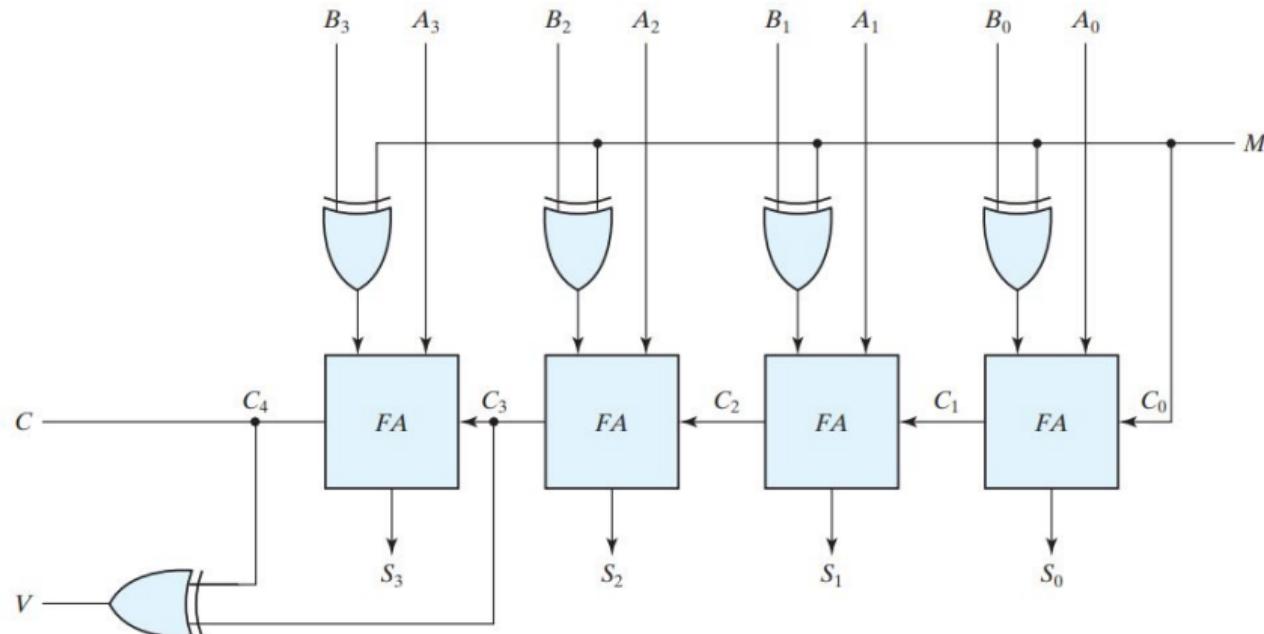


FIGURE 4.13

Four-bit adder-subtractor (with overflow detection)

### Decimal Adder- BCD adder

Binary to BCD conversion

BCD-0 to 9

Add 6 to convert the number from Binary to BCD(8421 representation )  
if it exceeds 9

The number 6 is chosen because it is the smallest value that, when added to a binary number between 10 (1010) and 15 (1111), produces a valid BCD representation.

BCD addition

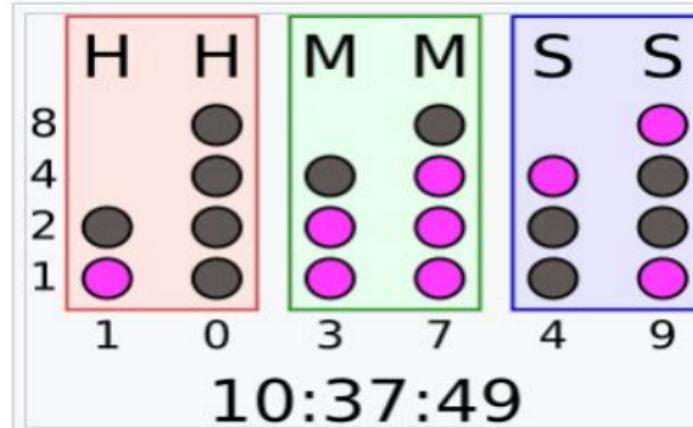
45+28

M. Morris Mano, Michael D. Ciletti, *Digital Design: With an Introduction to the Verilog HDL*, 5th ed., Prentice Hall, 2012, Section 4.6.

# Combinational logic

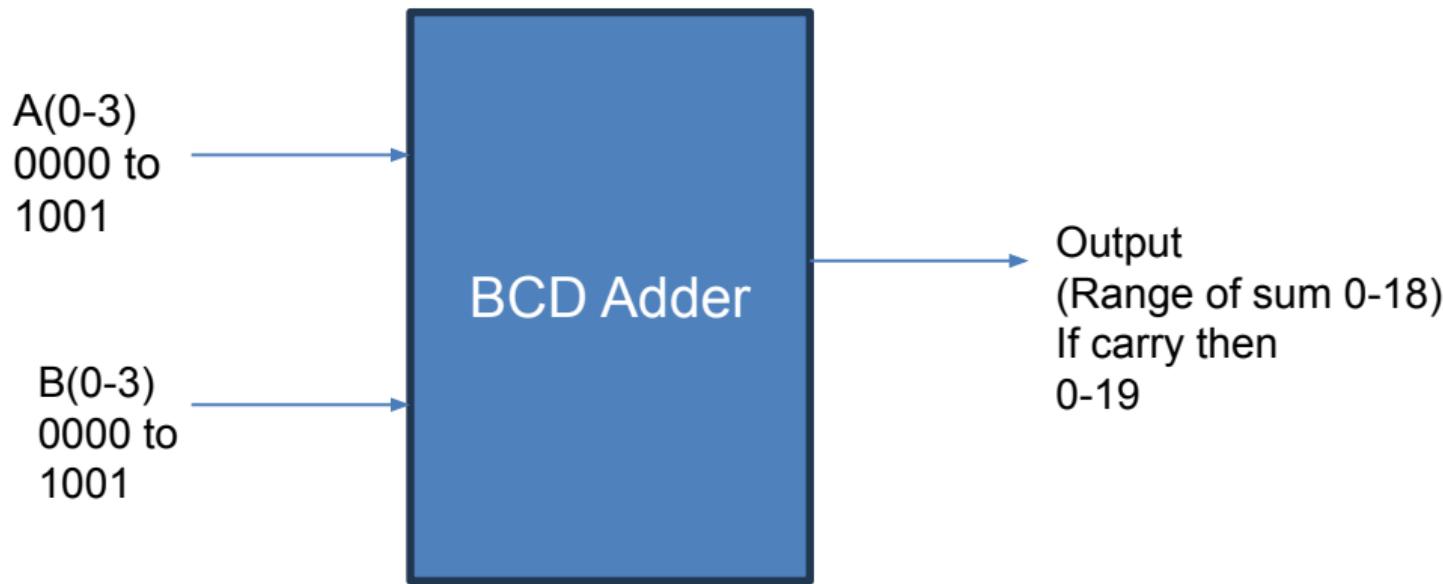
## Decimal Adder-BCD Adder

Why BCD?



Reading a **binary-coded decimal** clock: Add the values of each column of **LEDs** to get six decimal digits. There are two columns each for hours, minutes and seconds.

### BCD adder



# Combinational logic

## Decimal Adder-BCD Adder

Since each input digit does not exceed 9, the output sum cannot be greater than  $9 + 9 + 1 = 19$ , the 1 in the sum being an input carry

Table 4.5  
*Derivation of BCD Adder*

Binary Sum					BCD Sum					Decimal
K	Z <sub>8</sub>	Z <sub>4</sub>	Z <sub>2</sub>	Z <sub>1</sub>	C	S <sub>8</sub>	S <sub>4</sub>	S <sub>2</sub>	S <sub>1</sub>	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

# Combinational logic

## Decimal Adder-BCD Adder

Since each input digit does not exceed 9, the output sum cannot be greater than  $9 + 9 + 1 = 19$ , the 1 in the sum being an input carry

Table 4.5  
*Derivation of BCD Adder*

Binary Sum					BCD Sum					Decimal
K	Z <sub>8</sub>	Z <sub>4</sub>	Z <sub>2</sub>	Z <sub>1</sub>	C	S <sub>8</sub>	S <sub>4</sub>	S <sub>2</sub>	S <sub>1</sub>	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

## Decimal Adder-BCD Adder

---

In examining the contents of the table, it becomes apparent that when the binary sum is equal to or less than 1001, the corresponding BCD number is identical, and therefore no conversion is needed. When the binary sum is greater than 1001, we obtain an invalid BCD representation. The addition of binary 6 (0110) to the binary sum converts it to the correct BCD representation and also produces an output carry as required.

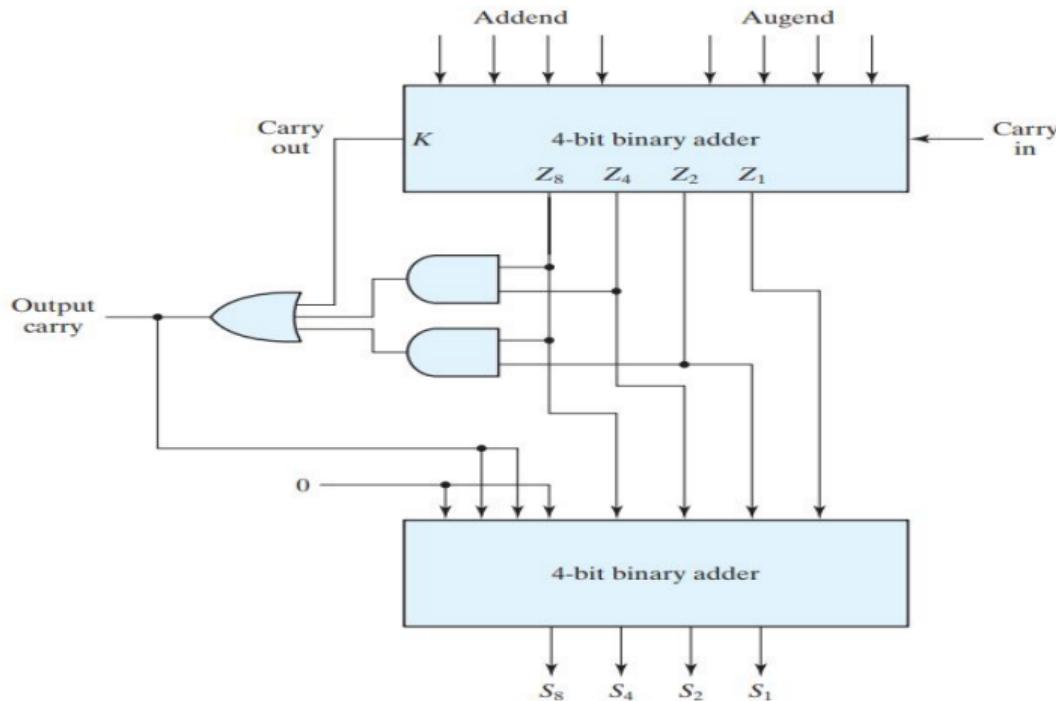
## Decimal Adder-BCD Adder

The logic circuit that detects the necessary correction can be derived from the entries in the table. It is obvious that a correction is needed when the binary sum has an output carry  $K=1$ . The other six combinations from 1010 through 1111 that need a correction have a 1 in position  $Z_8$ . To distinguish them from binary 1000 and 1001, which also have a 1 in position  $Z_8$ , we specify further that either  $Z_4$  or  $Z_2$  must have a 1. The condition for a correction and an output carry can be expressed by the Boolean function

$$C = K + Z_8 Z_4 + Z_8 Z_2$$

# Combinational logic

## Decimal Adder-BCD Adder



**FIGURE 4.14**  
Block diagram of a BCD adder

A BCD adder that adds two BCD digits and produces a sum digit in BCD . The two decimal digits, together with the input carry, are first added in the top four-bit adder to produce the binary sum. **When the output carry is equal to 0, nothing is added to the binary sum. When it is equal to 1, binary 0110 is added to the binary sum through the bottom four-bit adder.**

The output carry generated from the bottom adder can be ignored, since it supplies information already available at the output carry terminal. A decimal parallel adder that adds  $n$  decimal digits needs  $n$  BCD adder stages. The output carry from one stage must be connected to the input carry of the next higher order stage.

## Think about it

---

- . What is the Boolean expression used to detect correction in a BCD adder?
- A)  $C = K + Z_8Z_4 + Z_8Z_2$
  - B)  $C = K + Z_4Z_2$
  - C)  $C = K \cdot Z_8C$
  - D)  $C = Z_8 + Z_4 + Z_2$

In a 4-bit ripple carry adder, the worst-case carry propagation delay occurs when:

- A) All input bits are 0
- B) Only LSB has 1 and rest are 0
- C) All inputs are 1
- D) Inputs cause a carry to propagate through all stages

# Combinational logic

## Think about it

---

- . What is the Boolean expression used to detect correction in a BCD adder?
- A)  $C = K + Z_8Z_4 + Z_8Z_2$
  - B)  $C = K + Z_4Z_2$
  - C)  $C = K \cdot Z_8C$
  - D)  $C = Z_8 + Z_4 + Z_2$

Ans: A

In a 4-bit ripple carry adder, the worst-case carry propagation delay occurs when:

- A) All input bits are 0
  - B) Only LSB has 1 and rest are 0
  - C) All inputs are 1
  - D) Inputs cause a carry to propagate through all stages
- Answer: D) Inputs cause a carry to propagate through all stages

# Combinational logic

## Think about it

---

In 2's complement representation, what is the result of 1001–0110?

- A) 0011
- B) 1111
- C) 1011
- D) 0111

In a BCD adder, a correction factor is added when:

- A) The binary sum exceeds 1111
- B) The binary sum exceeds 1001
- C) The output carry is 0
- D) The binary sum is less than 1001

# Combinational logic

## Think about it

---

In 2's complement representation, what is the result of 1001–0110?

- A) 0011
- B) 1111
- C) 1011
- D) 0111

**Answer:** A) 0011

In a BCD adder, a correction factor is added when:

- A) The binary sum exceeds 1111
- B) The binary sum exceeds 1001
- C) The output carry is 0
- D) The binary sum is less than 1001

**Answer:** B) The binary sum exceeds 1001

# THANK YOU

---



**Team DDCO**  
Department of Computer Science



# DIGITAL DESIGN AND COMPUTER ORGANIZATION

## Binary Multipliers and Magnitude Comparator

---

**Team DDCO**  
**Department of Computer Science and Engineering**

# **DIGITAL DESIGN AND COMPUTER ORGANIZATION**

---



## **Binary Multipliers and Magnitude Comparator**

Department of Computer Science and Engineering

- Multiplication of binary numbers is performed in the same way as multiplication of decimal numbers.
- The **multiplicand is multiplied by each bit of the multiplier, starting from the least significant bit**. Each such multiplication forms a **partial product**. Successive partial products are shifted one position to the left.
- The final product is obtained from the sum of the partial products.

M. Morris Mano, Michael D. Ciletti, *Digital Design: With an Introduction to the Verilog HDL*, 5th ed., Prentice Hall, 2012, Section 4.7

# Combinational logic

## Binary Multipliers

$$\begin{array}{r} 1010 \\ \times 1011 \\ \hline 1010 \\ 1010 \\ 0000 \\ 1010 \\ \hline 1101110 \end{array}$$

→ Multiplicand

→ Multiplier

→ Partial product 1

→ Partial product 2

→ Partial product 3

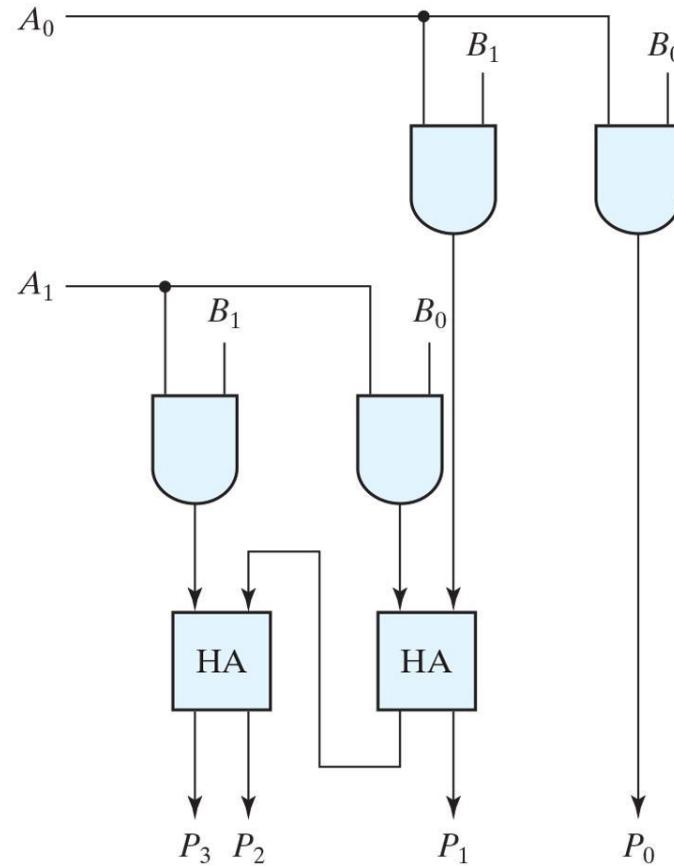
→ Partial product 4

# Combinational logic

## Binary Multipliers

Two-bit by two-bit binary multiplier.

$$\begin{array}{r} & B_1 & B_0 \\ & \quad & \quad \\ \begin{array}{r} A_1 \\ A_0 \end{array} & \hline & \begin{array}{r} A_0B_1 \\ A_0B_0 \end{array} \\ & \quad & \quad \\ \begin{array}{r} A_1B_1 \\ A_1B_0 \end{array} & \hline & \begin{array}{r} P_2 \\ P_1 \\ P_0 \end{array} \\ P_3 & & & \end{array}$$



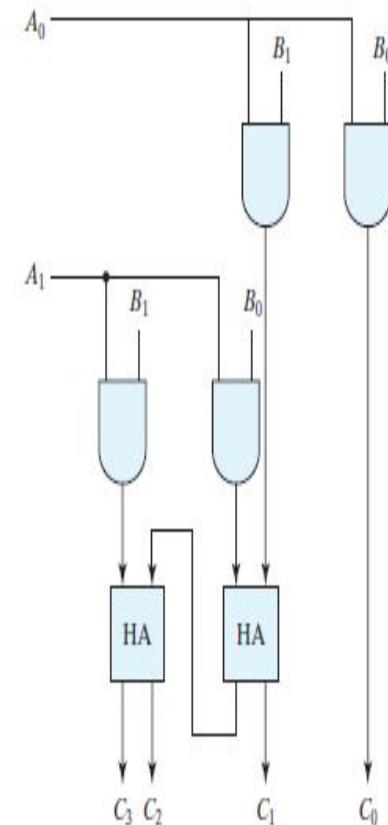
# Combinational logic

## Binary Multipliers

Consider the multiplication of two 2-bit numbers

- The multiplicand bits are  $B_1$  and  $B_0$ , the multiplier bits are  $A_1$  and  $A_0$ , and the product is  $C_3C_2C_1C_0$ .
- The first partial product is formed by multiplying  $B_1B_0$  by  $A_0$ .
- The multiplication of two bits such as  $A_0$  and  $B_0$  produces a 1 if both bits are 1; otherwise, it produces a 0. This is identical to an **AND** operation. Therefore, the partial product can be implemented with AND gates as shown in the diagram.

$$\begin{array}{r} B_1 \quad B_0 \\ A_1 \quad A_0 \\ \hline A_0B_1 \quad A_0B_0 \\ \hline C_3 \quad C_2 \quad C_1 \quad C_0 \end{array}$$

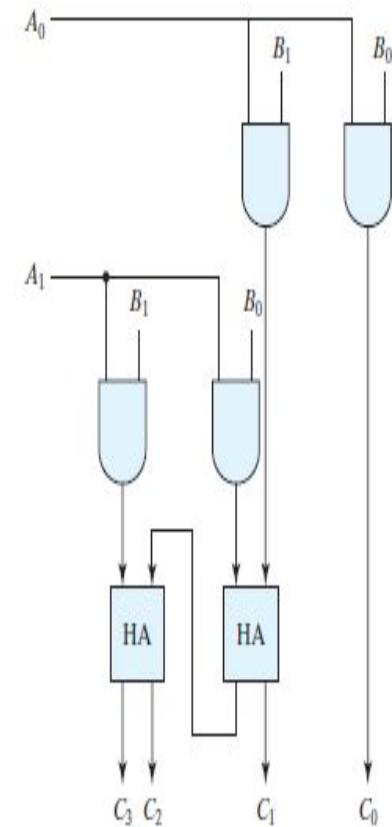


# Combinational logic

## Binary Multipliers

- The second partial product is formed by **multiplying  $B_1B_0$  by  $A_1$  and shifting one position to the left.**
- The **two partial products** are added with two **half-adder (HA) circuits**. Usually, there are more bits in the partial products and it is necessary to use full adders to produce the sum of the partial products.
- Note that the least significant bit of the product does not have to go through an adder, since it is formed by the output of the first AND gate.

$$\begin{array}{r} B_1 \quad B_0 \\ A_1 \quad A_0 \\ \hline A_0B_1 \quad A_0B_0 \\ \hline A_1B_1 \quad A_1B_0 \\ \hline C_3 \quad C_2 \quad C_1 \quad C_0 \end{array}$$



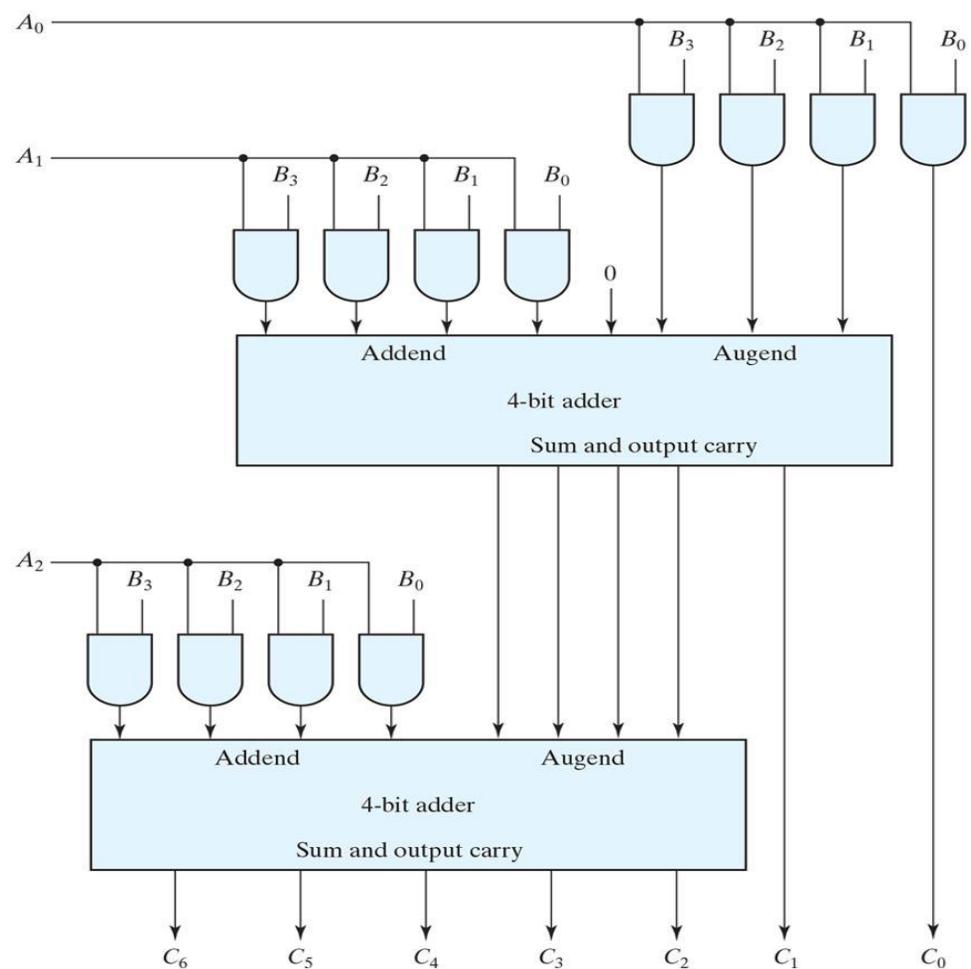
- A combinational circuit binary multiplier with more bits can be constructed in a similar fashion.
- A bit of the multiplier is ANDed with each bit of the multiplicand in as many levels as there are bits in the multiplier.
- The binary output in each level of AND gates is added with the partial product of the previous level to form a new partial product. The last level produces the product.
- For  $J$  multiplier bits and  $K$  multiplicand bits, we **need  $J * K$  AND gates** and
- **$(J - 1)K$  -bit adders to produce a product of  $(J + K)$  bits.**

# Combinational logic

## Binary Multipliers- $1011 \times 001$

Four-bit by three-bit binary multiplier.

- Consider a multiplier circuit that multiplies a binary number represented by four bits by a number represented by three bits.
- Let the multiplicand be represented by  $B_3B_2B_1B_0$  and the multiplier by  $A_2A_1A_0$ .
- Since  $K = 4$  and  $J = 3$ , we need 12 AND gates and two 4-bit adders to produce a product of seven bits



# Combinational logic

## Magnitude Comparators

- A *magnitude comparator* is a combinational circuit that compares two numbers  $A$  and  $B$  and determines their relative magnitudes. The outcome of the comparison is specified by three binary variables that indicate whether  $A > B$ ,  $A = B$ , or  $A < B$ .
- On the one hand, the circuit for comparing two  $n$ -bit numbers has  $2^{2n}$  entries in the truth table and becomes too cumbersome, even with  $n = 3$
- Consider two numbers,  $A$  and  $B$ , with four digits each. Write the coefficients of the numbers in descending order of significance:

$$A = A_3 \ A_2 \ A_1 \ A_0$$

$$B = B_3 \ B_2 \ B_1 \ B_0$$

- The two numbers are equal if all pairs of significant digits are equal:
- $A_3 = B_3, A_2 = B_2, A_1 = B_1$ , and  $A_0 = B_0$ .

M. Morris Mano, Michael D. Ciletti, *Digital Design: With an Introduction to the Verilog HDL*, 5th ed., Prentice Hall, 2012, Section 4.8

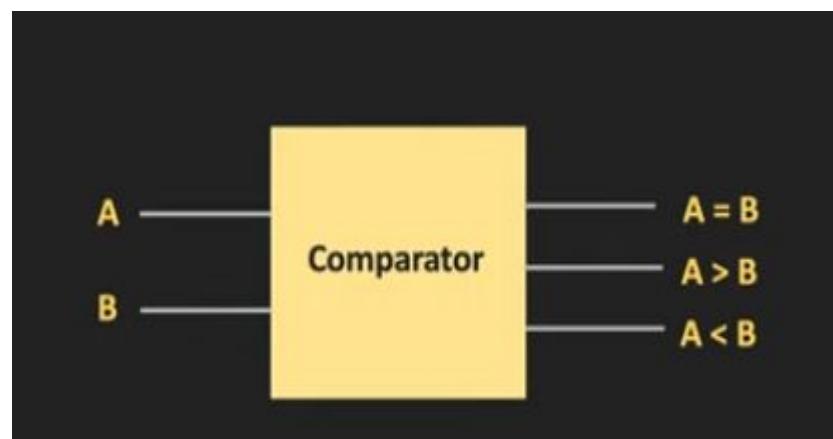
# Combinational logic

## Magnitude Comparators

- When the numbers are binary, the digits are either 1 or 0, and the equality of each pair of bits can be expressed logically with an exclusive-NOR function as

$$x_i = A_i B_i + A_i' B_i' \text{ for } i = 0, 1, 2, 3$$

where  $x_i = 1$  only if the pair of bits in position  $i$  are equal (i.e., if both are 1 or both are 0).



# Combinational logic

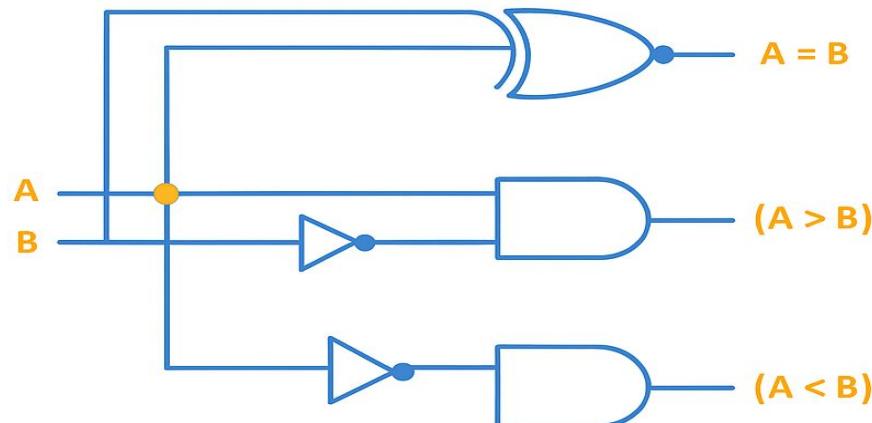
## Magnitude Comparators

### 1-bit Comparator

Truth Table

A	B	A = B	A > B	A < B
0	0	1	0	0
0	1	0	0	1
1	0	0	1	0
1	1	1	0	0

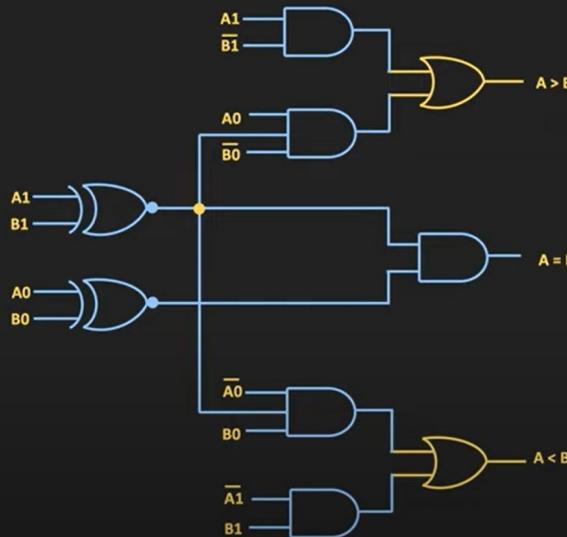
$$\text{Output} = \overline{\bar{A} \bar{B}} + AB$$



# 2 bit comparator

1-bit Comparator       $\rightarrow$  2 variables  $\rightarrow$  4 rows  
2-bit Comparator       $\rightarrow$  4 variables  $\rightarrow$  16 rows  
n-bit Comparator       $\rightarrow$   $2^n$  variables  $\rightarrow$   $2^{2n}$  rows

## 2-bit Comparator

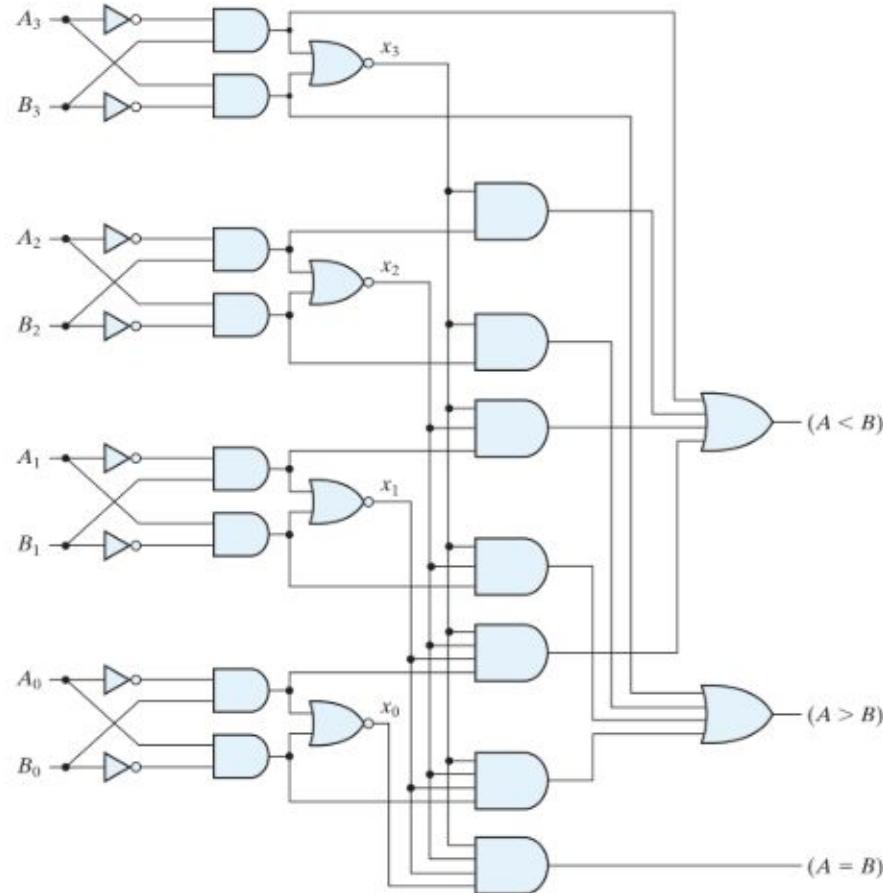


A1	A0	B1	B0	A > B	A < B	A = B
0	0	0	0	0	0	1
0	0	0	1	0	1	0
0	0	1	0	0	1	0
0	0	1	1	0	1	0
0	1	0	0	1	0	0
0	1	0	1	0	0	1
0	1	1	0	0	1	0
0	1	1	1	0	1	0
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	0	1
1	0	1	1	0	1	0
1	1	0	0	1	0	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	0	0	1

# Combinational logic

## Magnitude Comparators

### 4 Bit Comparator



**FIGURE 4.17**  
Four-bit magnitude comparator

$$A = A_3 A_2 A_1 A_0$$

$$B = B_3 B_2 B_1 B_0$$

$$A > B \quad \longrightarrow \quad A_3 > B_3$$

OR

If  $A_3 = B_3$  and  $A_2 > B_2$

OR

If  $A_3 = B_3$  and  $A_2 = B_2$  and  $A_1 > B_1$

OR

If  $A_3 = B_3$  and  $A_2 = B_2$  and  $A_1 = B_1$  and  $A_0 > B_0$

✓

$$(A > B) = A_3 \overline{B_3} + (A_3 \odot B_3) A_2 \overline{B_2} + (A_3 \odot B_3) (A_2 \odot B_2) A_1 \overline{B_1} +$$

$$(A_3 \odot B_3) (A_2 \odot B_2) (A_1 \odot B_1) A_0 \overline{B_0}$$

$$(A < B) = A'_3 B_3 + x_3 A'_2 B_2 + x_3 x_2 A'_1 B'_1 + x_3 x_2 x_1 A' n_0 B'_0$$

$$\mathbf{A} = \mathbf{A}_3 \ \mathbf{A}_2 \ \mathbf{A}_1 \ \mathbf{A}_0$$

$$\mathbf{B} = \mathbf{B}_3 \ \mathbf{B}_2 \ \mathbf{B}_1 \ \mathbf{B}_0$$

$$\mathbf{A} < \mathbf{B} \quad \longrightarrow \quad \mathbf{A}_3 < \mathbf{B}_3$$

**OR**

If  $\mathbf{A}_3 = \mathbf{B}_3$  and  $\mathbf{A}_2 < \mathbf{B}_2$

**OR**

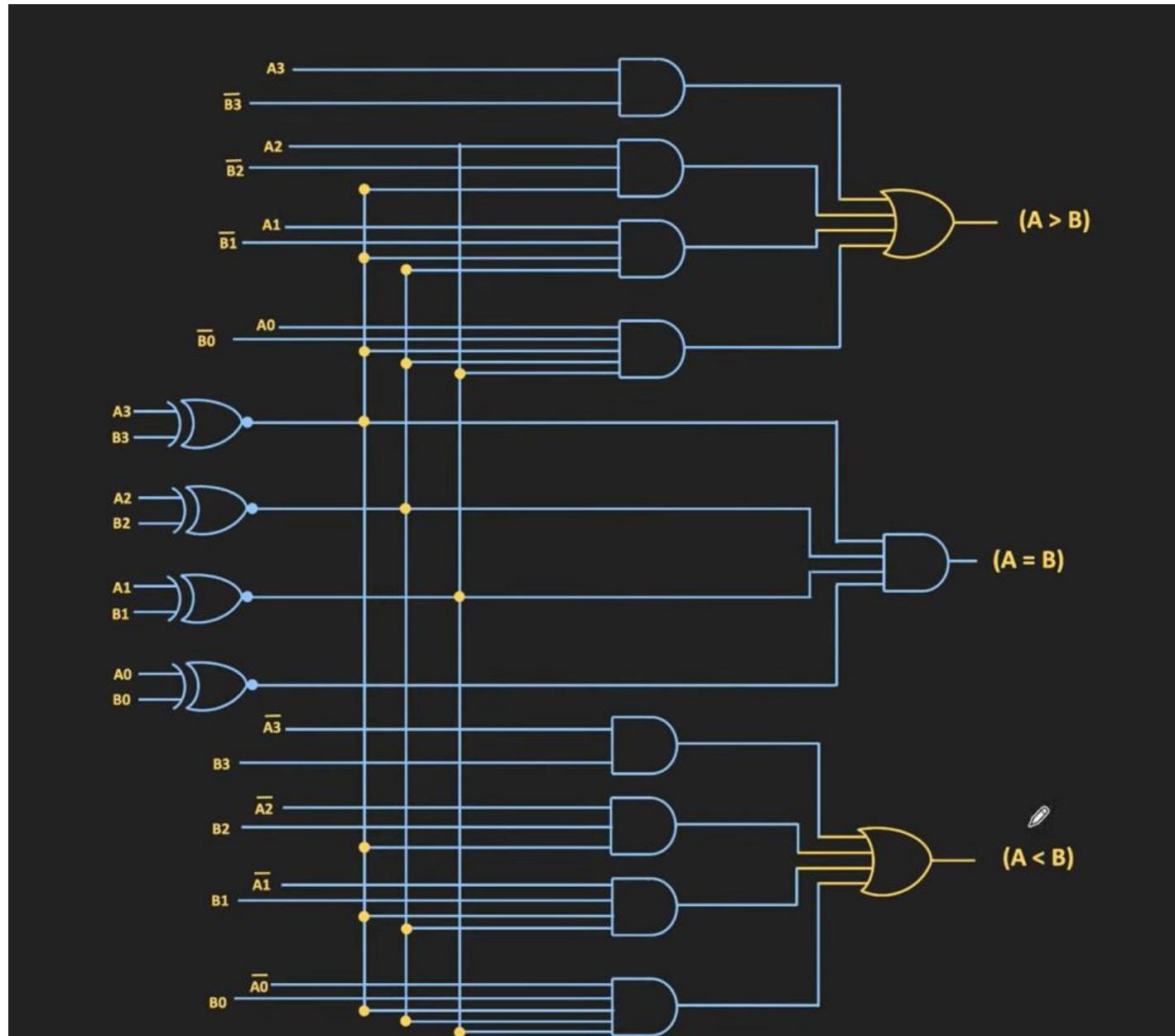
If  $\mathbf{A}_3 = \mathbf{B}_3$  and  $\mathbf{A}_2 = \mathbf{B}_2$  and  $\mathbf{A}_1 < \mathbf{B}_1$

**OR**

If  $\mathbf{A}_3 = \mathbf{B}_3$  and  $\mathbf{A}_2 = \mathbf{B}_2$  and  $\mathbf{A}_1 = \mathbf{B}_1$  and  $\mathbf{A}_0 < \mathbf{B}_0$

$$\begin{aligned}
 (\mathbf{A} < \mathbf{B}) = & \overline{\mathbf{A}_3} \ \mathbf{B}_3 + (\mathbf{A}_3 \odot \mathbf{B}_3) \overline{\mathbf{A}_2} \ \mathbf{B}_2 + (\mathbf{A}_3 \odot \mathbf{B}_3) (\mathbf{A}_2 \odot \mathbf{B}_2) \ \overline{\mathbf{A}_1} \ \mathbf{B}_1 + \\
 & (\mathbf{A}_3 \odot \mathbf{B}_3) (\mathbf{A}_2 \odot \mathbf{B}_2) (\mathbf{A}_1 \odot \mathbf{B}_1) \ \overline{\mathbf{A}_0} \ \mathbf{B}_0
 \end{aligned}$$

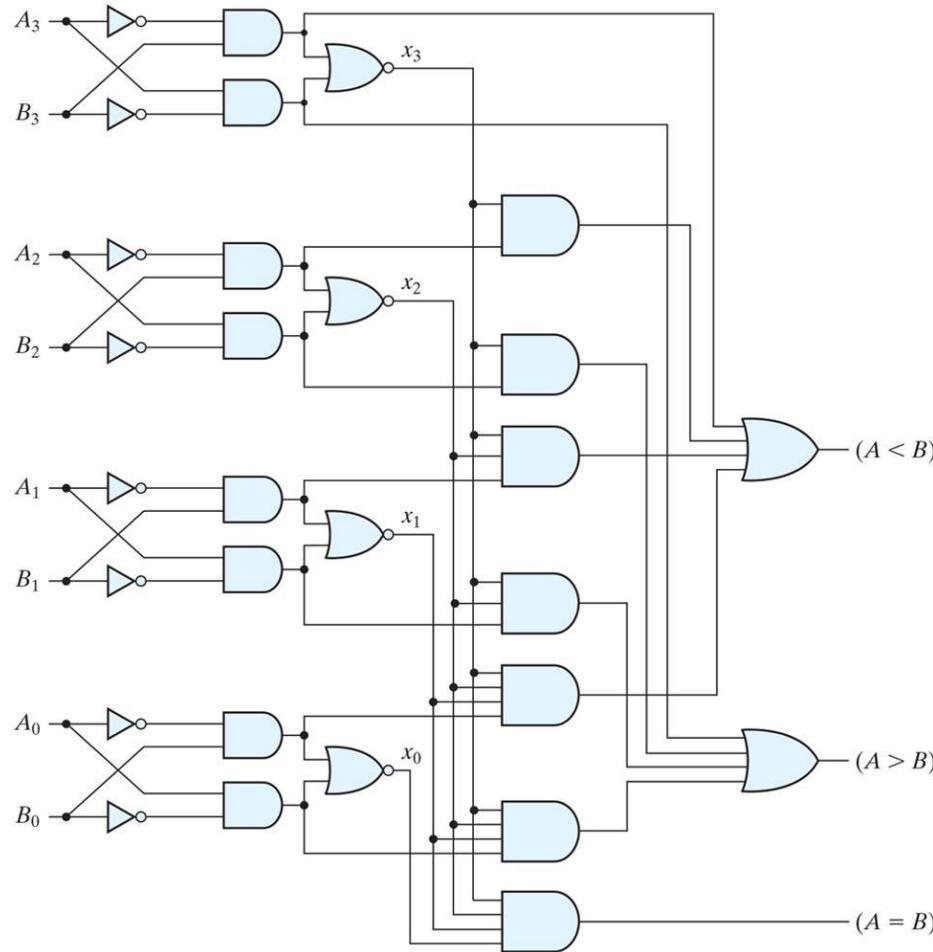
# Magnitude Comparator



# Combinational logic

## Magnitude Comparators

Four-bit Magnitude Comparator



# Combinational logic

## Magnitude Comparators

- For equality to exist, all  $x_i$  variables must be equal to 1, a condition that dictates an AND operation of all variables:

$$(A = B) = x_3x_2x_1x_0$$

- The *binary* variable ( $A = B$ ) is equal to 1 only if all pairs of digits of the two numbers are equal.
- To determine whether  $A$  is greater or less than  $B$ , we inspect the relative magnitudes of pairs of significant digits, starting from the most significant position.
- If the two digits of a pair are equal, we compare the next lower significant pair of digits.
- The comparison continues until a pair of unequal digits is reached. If the corresponding digit of  $A$  is 1 and that of  $B$  is 0, we conclude that  $A > B$ .
- If the corresponding digit of  $A$  is 0 and that of  $B$  is 1, we have  $A < B$ .

# Combinational logic

## Magnitude Comparators

- The sequential comparison can be expressed logically by the two Boolean functions

$$(A > B) = A_3B'_3 + x_3A_2B'_2 + x_3x_2A_1B'_1 + x_3x_2x_1A_0B'_0$$

$$(A < B) = A'_3B_3 + x_3A'_2B_2 + x_3x_2A'_1B_1 + x_3x_2x_1A'n_0B'_0$$

- The symbols  $(A > B)$  and  $(A < B)$  are binary output variables that are equal to 1 when  $A > B$  and  $A < B$ , respectively.
- The unequal outputs can use the same gates that are needed to generate the equal output
- The four  $x$  outputs are generated with exclusive-NOR circuits and are applied to an AND gate to give the output binary variable  $(A = B)$ .
- The other two outputs use the  $x$  variables to generate the Boolean functions listed previously. This is a multilevel implementation and has a regular pattern.

# Combinational logic

## Think about it

---

A 4-bit by 4-bit binary multiplier is implemented using AND gates and adders. How many AND gates are required?

- A.** 16
- B.** 8
- C.** 32
- D.** 64

For comparing two 4-bit numbers A and B, a combinational circuit gives three outputs  $A > B$ ,  $A = B$ , and  $A < B$ . If each bit comparison uses XNOR and AND/OR logic, how many XNOR gates are needed?

- A.** 4
- B.** 3
- C.** 2
- D.** 1

# Combinational logic

## Think about it

---

A 4-bit by 4-bit binary multiplier is implemented using AND gates and adders. How many AND gates are required?

- A. 16
- B. 8
- C. 32
- D. 64

**Answer:** A. 16

**Explanation:**  $4 \times 4 = 16$  partial products via AND gates.

For comparing two 4-bit numbers A and B, a combinational circuit gives three outputs  $A > B$ ,  $A = B$ , and  $A < B$ . If each bit comparison uses XNOR and AND/OR logic, how many XNOR gates are needed?

- A. 4
- B. 3
- C. 2
- D. 1

**Answer:** A. 4

**Explanation:** Each bit position requires 1 XNOR gate to check equality.

# Combinational logic

## Think about it

---



Describe a real-world application where a magnitude comparator is essential. Explain how the comparator's outputs are used in decision-making.

# Combinational logic

## Think about it

---



Describe a real-world application where a magnitude comparator is essential. Explain how the comparator's outputs are used in decision-making.

**Ans: Digital Thermostat in an Air Conditioning (AC) System**

A **digital thermostat** constantly monitors the **room temperature** and compares it with the **desired set temperature**. A **magnitude comparator** is used to perform this comparison.

A: Current room temperature (in binary)

B: Desired set temperature (in binary)

# Combinational logic

Think about it



## Comparator Outputs:

$A > B \rightarrow$  Room is **warmer** than desired

$A = B \rightarrow$  Room is at **desired temperature**

$A < B \rightarrow$  Room is **colder** than desired

### Comparator Output

### System Decision

$A > B$

Turn ON cooling (AC compressor starts)

$A = B$

Maintain current state (system remains idle)

$A < B$

Turn OFF cooling or turn ON heating (if available)

### Other Examples Where Comparators Are Used:

**Speed regulators** in vehicles (comparing set vs. actual speed)

**Digital counters** (checking if max value is reached)

**Elevator control systems** (comparing current floor with target floor)

**Battery chargers** (comparing voltage levels)



# THANK YOU

---



**Team DDCO**  
Department of Computer Science



**PES**

UNIVERSITY

CELEBRATING 50 YEARS

# DIGITAL DESIGN AND COMPUTER ORGANIZATION

## Decoders

---

**Team DDCO**

**Department of Computer Science and Engineering**

# **DIGITAL DESIGN AND COMPUTER ORGANIZATION**

---



## **Decoders**

Department of Computer Science and Engineering

# Combinational logic

## Decoders

---



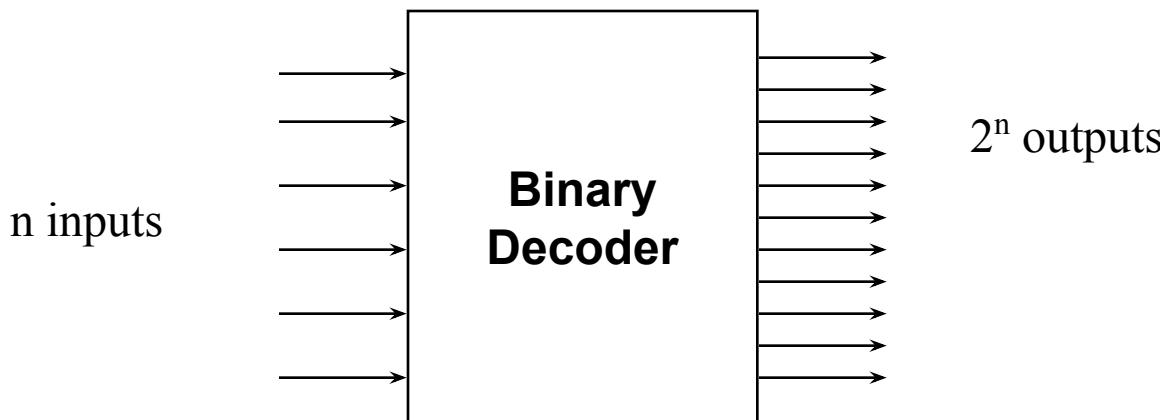
- Discrete quantities of information are represented in digital systems by binary codes.
- A binary code of  $n$  bits is capable of representing up to  $2^n$  distinct elements of coded information
- A decoder is a combinational circuit that converts binary information from  $n$  input lines to a maximum of  $2^n$  unique output lines. If the  $n$ -bit coded information has unused combinations, the decoder may have fewer than  $2^n$  outputs.
- The decoders presented here are called  $n$ -to- $m$ -line decoders, where  $m \dots 2^n$
- The name decoder is also used in conjunction with other code converters, such as a BCD-to-seven-segment decoder.

# Combinational logic

## Decoders

### Decoders

Black box with  $n$  input lines and  $2^n$  output lines



Only one output is a 1 for any given input

M. Morris Mano, Michael D. Ciletti, *Digital Design: With an Introduction to the Verilog HDL*, 5th ed., Prentice Hall, 2012, Section 4.9

# Combinational logic

## Decoders

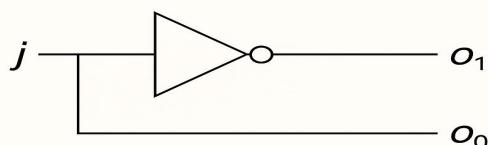
### 1: 2 Decoder:

A **1-to-2 decoder** is a simple combinational logic circuit that takes **1 input** and produces **2 outputs**. It is used to activate exactly **one of the two outputs** based on the binary value of the input.

Truth table :

Input (J)	Output( $O_1$ )	Output( $O_0$ )
0	1	0
1	0	1

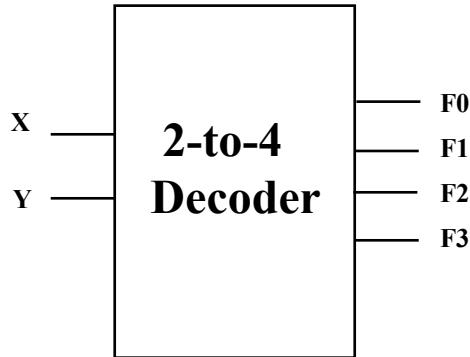
1:2 decoder logic circuit:



M. Morris Mano, Michael D. Ciletti, *Digital Design: With an Introduction to the Verilog HDL*, 5th ed., Prentice Hall, 2012, Section 4.9

# Combinational logic

## Decoders

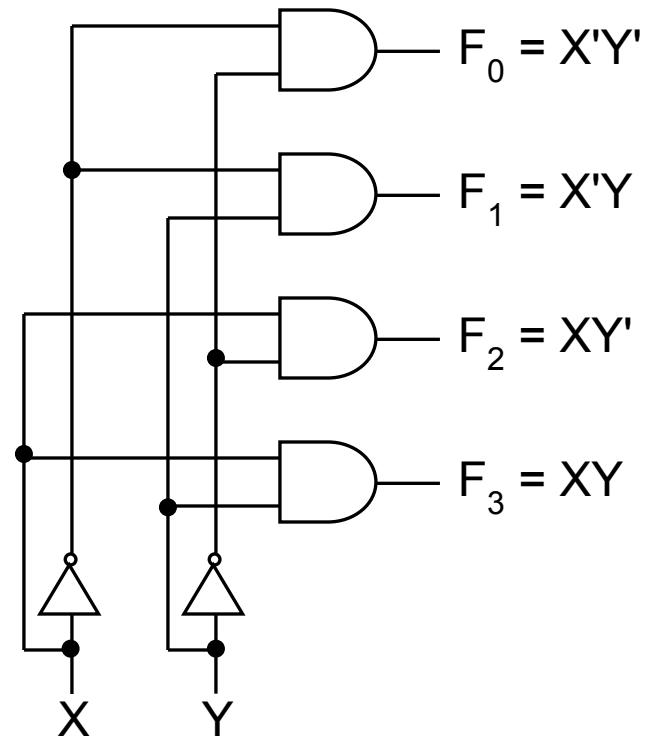


Truth Table:

X	Y	F <sub>0</sub>	F <sub>1</sub>	F <sub>2</sub>	F <sub>3</sub>
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

From truth table, circuit for 2x4 decoder is:

Note: Each output is a 2-variable minterm ( $X'Y'$ ,  $X'Y$ ,  $XY'$  or  $XY$ )



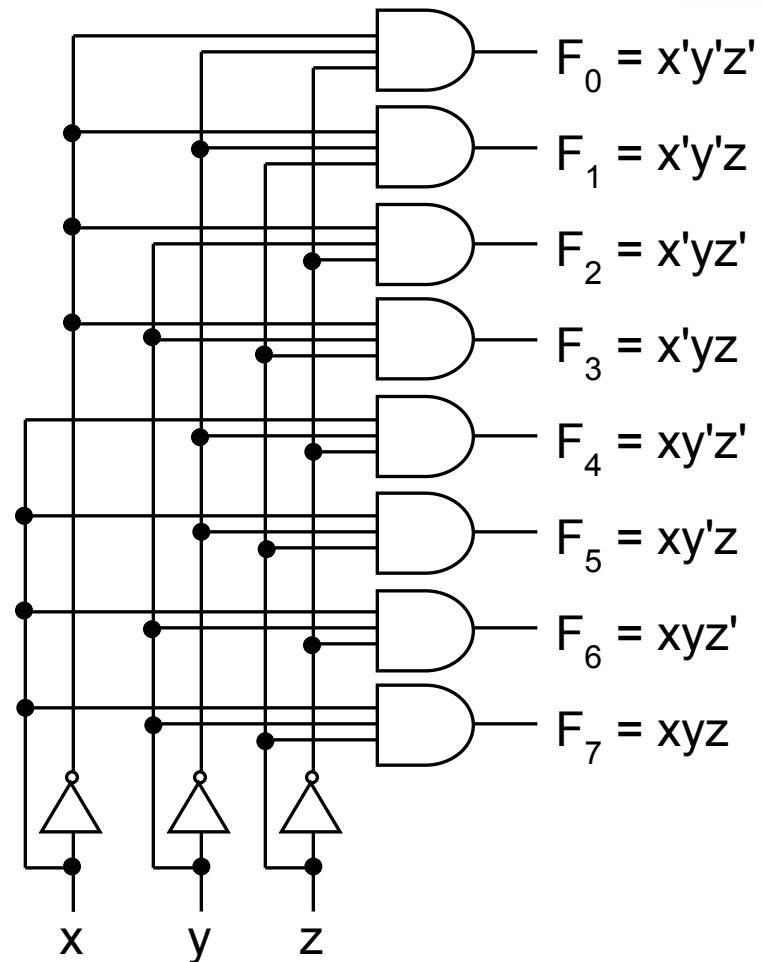
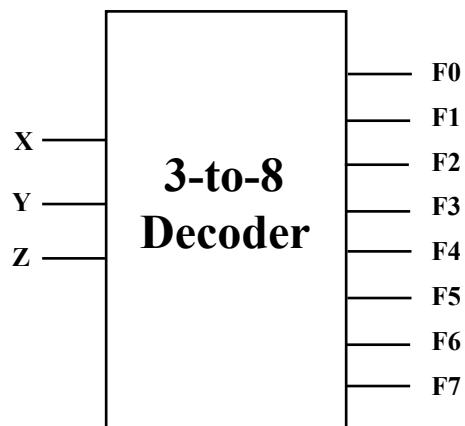
# Combinational logic

## Decoders (similar to code converter – binary to octal)

### 3-to-8 Binary Decoder

Truth Table:

x	y	z	F <sub>0</sub>	F <sub>1</sub>	F <sub>2</sub>	F <sub>3</sub>	F <sub>4</sub>	F <sub>5</sub>	F <sub>6</sub>	F <sub>7</sub>
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1



# Combinational logic

## Decoders with enable input

---



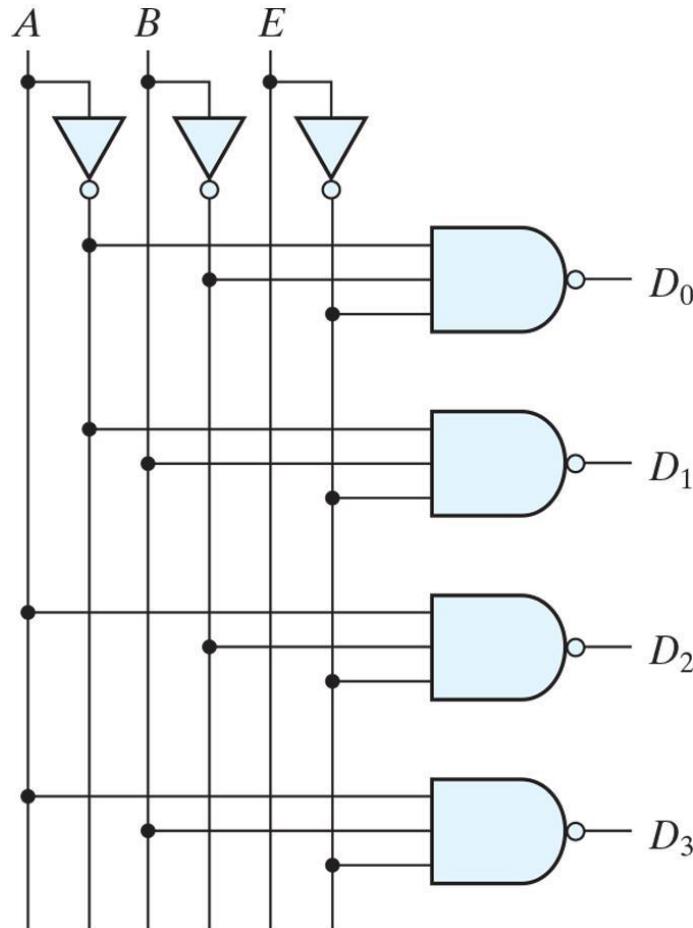
Enable input- allows the decoders outputs to be turned “ON” or “OFF” as required. Output is only generated when the Enable input has value 1; otherwise, all outputs are 0.

Some decoders are constructed with NAND gates. Since a NAND gate produces the AND operation with an inverted output, it becomes more economical to generate the decoder minterms in their complemented form

# Combinational logic

## Decoders with NAND gates

Two to four line Decoder with Enable Input



Start with a 2-bit decoder  
Add an enable signal (E)

Note: use of NANDs  
only one 0 active! (**active low**)  
if  $E = 0$

$AB=10$  then output is  $D_2$

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

# Combinational logic

## Decoders with enable input



A decoder with enable input can function as a demultiplexer— a circuit that receives information from a single line and directs it to one of  $2^n$  possible output lines. The selection of a specific output is controlled by the bit combination of n selection lines.

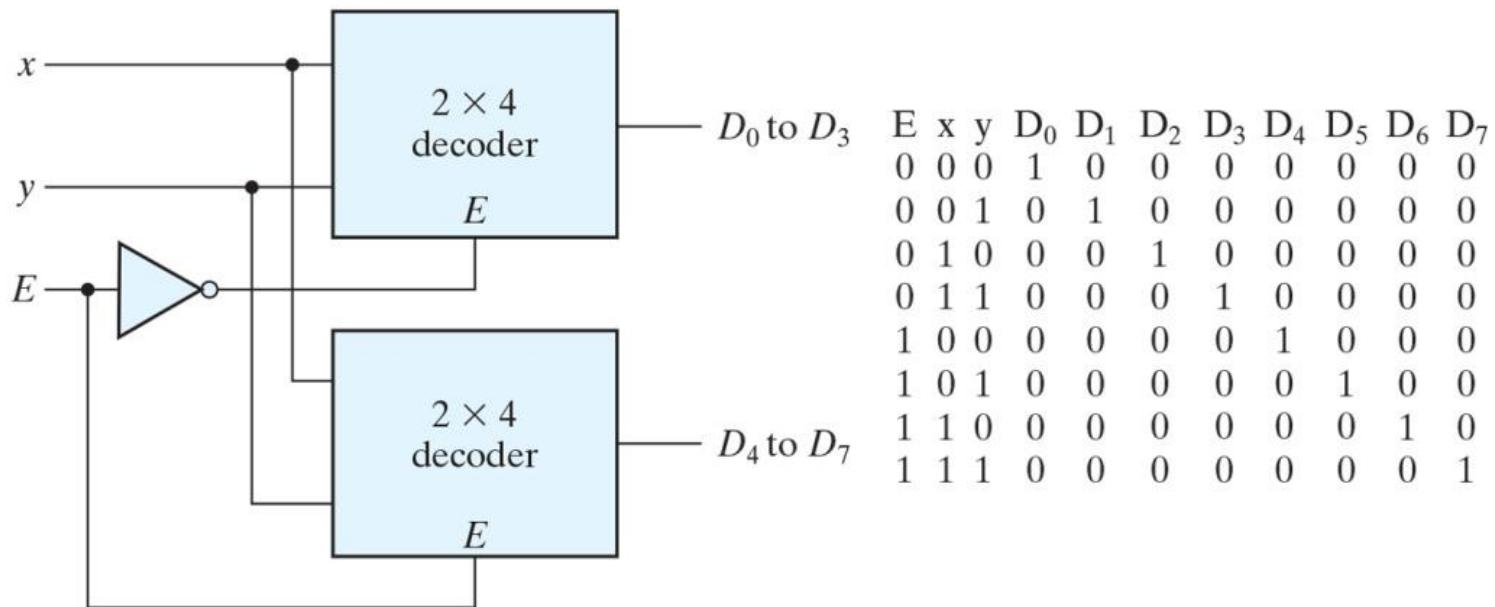
Because decoder and demultiplexer operations are obtained from the same circuit, a decoder with an enable input is referred to as a decoder – demultiplexer

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

# Combinational logic

## Decoders

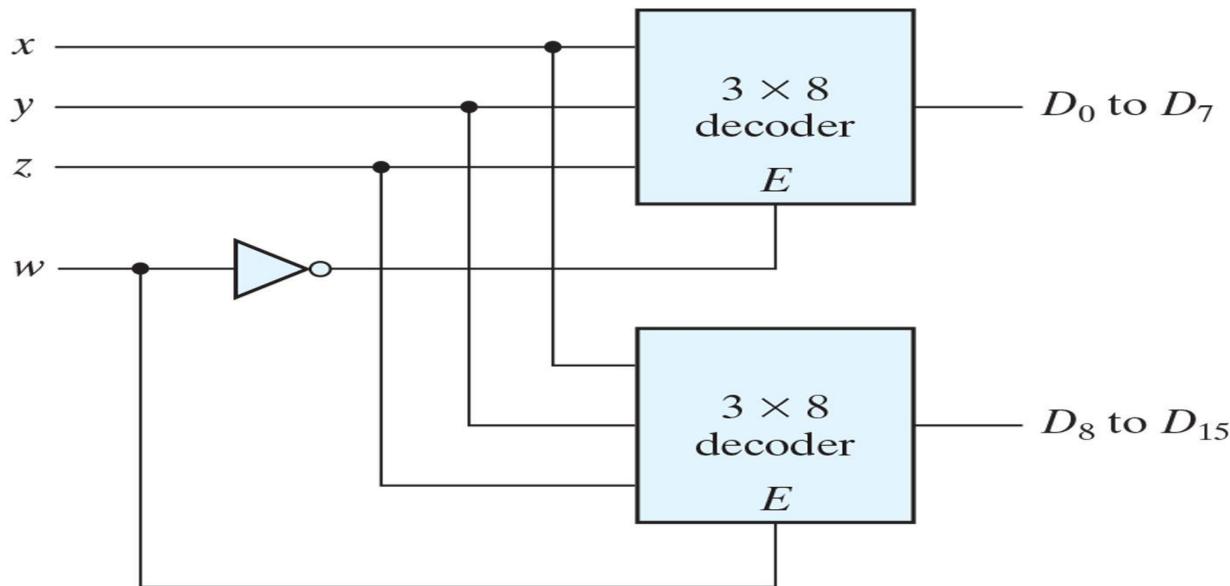
3 x 8 decoder constructed with two 2 x 4 decoders.



# Combinational logic

## Decoders

4 x 16 decoder constructed with two 3 x 8 decoders.



Enable can also be active high

In this example, only one decoder can be active at a time.

$x$ ,  $y$ ,  $z$  effectively select output line for  $w$

# Combinational logic

## Decoders

---

### Implementing Functions Using Decoders

Any  $n$ -variable logic function can be implemented using a single  $n$ -to- $2^n$  decoder to generate the minterms

- **OR gate forms the sum.**
- **The output lines of the decoder corresponding to the minterms of the function are used as inputs to the or gate.**

Any combinational circuit with  $n$  inputs and  $m$  outputs can be implemented with an  $n$ -to- $2^n$  decoder with  $m$  OR gates.

Suitable when a circuit has many outputs, and each output function is expressed with few minterms.

# Combinational logic

## Decoders-Combinational Logic implementation

A decoder provides the  $2^n$  minterms of n input variables

Each asserted output of the decoder is associated with a unique pattern of input bits.

Since any Boolean function can be expressed in sum-of-minterms form, a decoder that generates the minterms of the function, together with an external OR gate that forms their logical sum, provides a hardware implementation of the function.

In this way, any combinational circuit with n inputs and m outputs can be implemented with an n -to- $2^n$  -line decoder and m OR gates.

Questions:

1. Build XNOR Gate with 2 inputs using Decoders.
2. Implement full adder using decoders

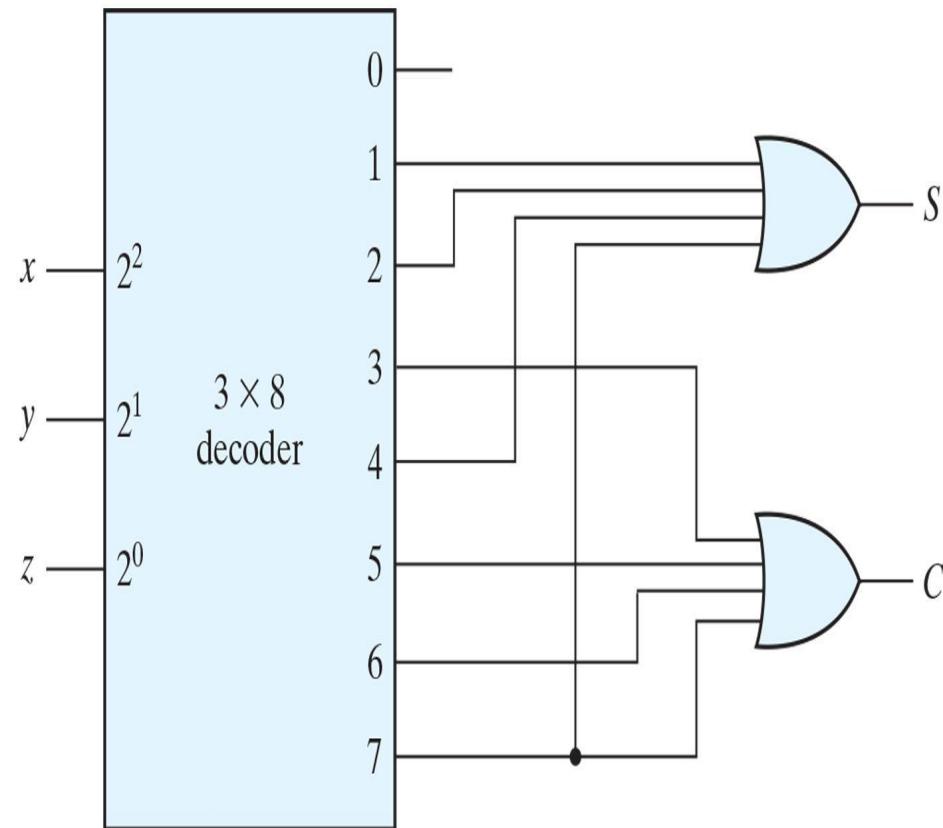
# Combinational logic

## Decoders-Combinational Logic implementation

A decoder provides the  $2^n$  minterms of n input variables

Implementation of a full adder with a decoder.

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



Full adder

$$S(x, y, z) = \sum (1, 2, 4, 7)$$

$$C(x, y, z) = \sum (3, 5, 6, 7)$$

# Combinational logic

## Decoders

A function with a long list of minterms requires an OR gate with a large number of inputs.

A function having a list of  $k$  minterms can be expressed in its complemented form  $F'$  with  $2^n - k$  minterms.

If the number of minterms in the function is greater than  $2^n/2$ , then  $F'$  can be expressed with fewer minterms.

(Example: For 3 variables ( $n=3$ ), total = 8 minterms.

If  $F = 6$  minterms, better to write

$$F = (\sum \text{6 minterms}) = (\sum \text{of all } 8 - 6) = \text{complement of 2 minterms}$$

$$F = (\sum \text{6 minterms}) = (\sum \text{of all } 8 - 6) = \text{complement of 2 minterms.}$$

In such a case, it is advantageous to use a NOR gate to sum the minterms of  $F$ .

# Combinational logic

## Decoders-Application

To manage traffic at a **4-way intersection** (North, South, East, West) **efficiently using fewer control lines**, by using a **2-to-4 line decoder**.

### Why Use a Decoder?

In a 4-way intersection:

You need to **control each direction's traffic light individually**.

Without a decoder, you'd need **4 separate control signals**, one for each direction.

A **2-to-4 decoder** allows you to **use only 2 input lines** to control 4 outputs, reducing hardware complexity

# Combinational logic

## Decoders-Application

Understanding the 2-to-4 Decoder

Inputs: 2 bits (say A and B)

Outputs: 4 lines: Y0, Y1, Y2, Y3

Operation: When inputs = 00, output Y0 is active (others inactive)

When inputs = 01, output Y1 is active

When inputs = 10, output Y2 is active

When inputs = 11, output Y3 is active

Decoder Output	Controls Traffic Light at
Y0	North direction
Y1	South direction
Y2	East direction
Y3	West direction

# Combinational logic

## Decoders-Think about it

---

1. Build XNOR gate using Decoder
2. A function with more than  $2^n/2$  minterms is better implemented using:
  - (A) A single AND gate
  - (B) Complemented function and a NOR gate
  - (C) Only XOR gates
  - (D) Full adders
3. In a 2-to-4 decoder with an active-low enable input, what happens when the enable input is 1?
  - (A) All outputs are enabled
  - (B) Decoder functions normally
  - (C) All outputs are 0
  - (D) All outputs are 1

# Combinational logic

## Decoders-Think about it

1. Build XNOR gate using Decoder

2. A function with more than  $2^n/2$  minterms is better implemented using:

- (A) A single AND gate
- (B) Complemented function and a NOR gate
- (C) Only XOR gates
- (D) Full adders

Answer: (B)

3. In a 2-to-4 decoder with an active-low enable input, what happens when the enable input is 1?

- (A) All outputs are enabled
- (B) Decoder functions normally
- (C) All outputs are 0
- (D) All outputs are 1

Answer: (D)



# THANK YOU

---



**Team DDCO**  
Department of Computer Science



**PES**

UNIVERSITY

CELEBRATING 50 YEARS

# DIGITAL DESIGN AND COMPUTER ORGANIZATION

## Encoders

---

**Team DDCO**

**Department of Computer Science and Engineering**

# **DIGITAL DESIGN AND COMPUTER ORGANIZATION**

---



## **Encoders**

Department of Computer Science and Engineering

# Combinational logic

## Encoders

### Encoders

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

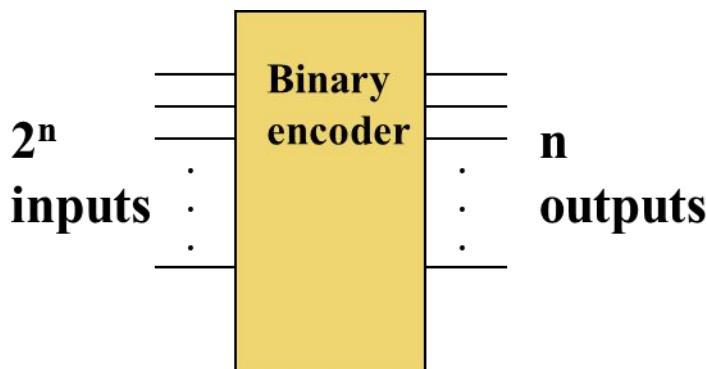
If the a decoder's output code has fewer bits than the input code, the device is usually called an encoder.

e.g.  $2^n$ -to- $n$

The simplest encoder is a  $2^n$ -to- $n$  binary encoder

One of  $2^n$  inputs = 1

Output is an  $n$ -bit binary number



M. Morris Mano, Michael D. Ciletti, *Digital Design: With an Introduction to the Verilog HDL*, 5th ed., Prentice Hall, 2012,  
Section 4.10

# Combinational logic

## Encoders

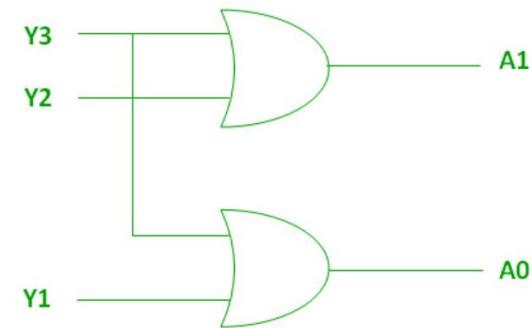
### 4:2 Encoder



$$A_1 = Y_3 + Y_2$$
$$A_0 = Y_3 + Y_1$$

The Truth table of 4 to 2 encoders is as follows.

INPUTS				OUTPUTS	
Y3	Y2	Y1	Y0	A1	A0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1



*Implementation using OR Gate*

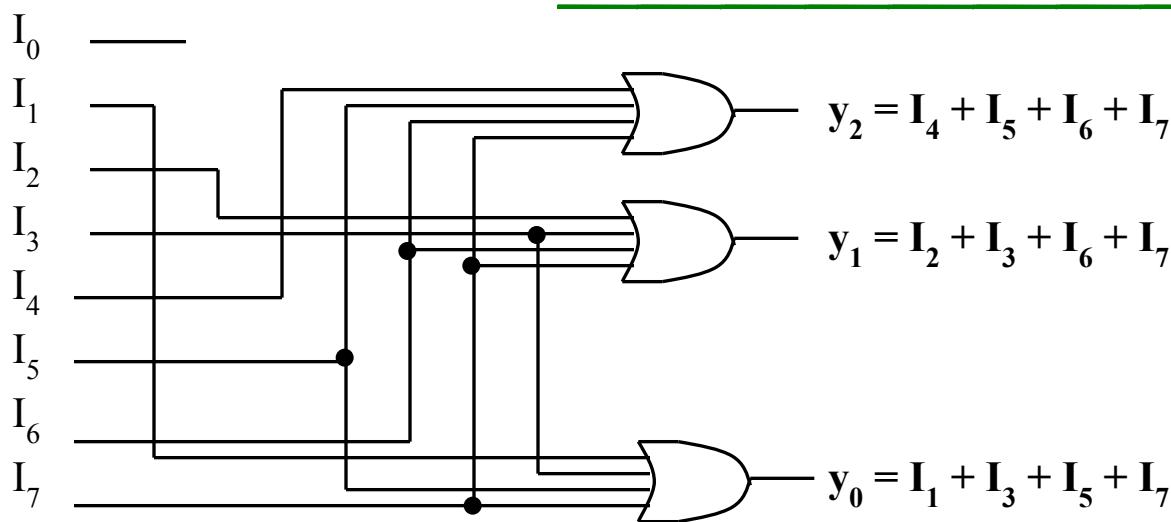
# Combinational logic

## Encoders (octal to binary)

8-to-3 Binary Encoder

**At any one time, only one input line has a value of 1.**

Inputs								Outputs		
I <sub>0</sub>	I <sub>1</sub>	I <sub>2</sub>	I <sub>3</sub>	I <sub>4</sub>	I <sub>5</sub>	I <sub>6</sub>	I <sub>7</sub>	y <sub>2</sub>	y <sub>1</sub>	y <sub>0</sub>
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1



# Combinational logic

## Encoders (octal to binary)

Truth Table for Octal-to-Binary Encoder

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

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

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

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

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

# Combinational logic

## Priority Encoder

---

A priority encoder is an encoder circuit that includes the priority function. The operation of the priority encoder is such that if two or more inputs are equal to 1 at the same time, the input having the highest priority will take precedence.

# Combinational logic

## Priority Encoder

Truth Table of a Priority Encoder

Inputs				Outputs		
$D_0$	$D_1$	$D_2$	$D_3$	$x$	$y$	$v$
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

# Combinational logic

## Priority Encoder

In addition to the two outputs  $x$  and  $y$ , the circuit has a third output designated by  $V$ ; this is a valid bit indicator that is set to 1 when one or more inputs are equal to 1. If all inputs are 0, there is no valid input and  $V$  is equal to 0. The other two outputs are not inspected when  $V$  equals 0 and are specified as don't-care conditions.

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

# Combinational logic

## Priority Encoders

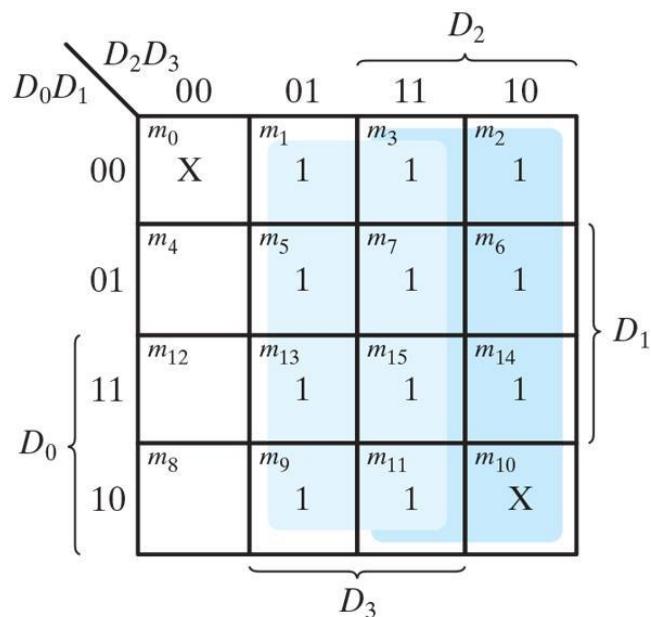


## Maps for a Priority Encoder

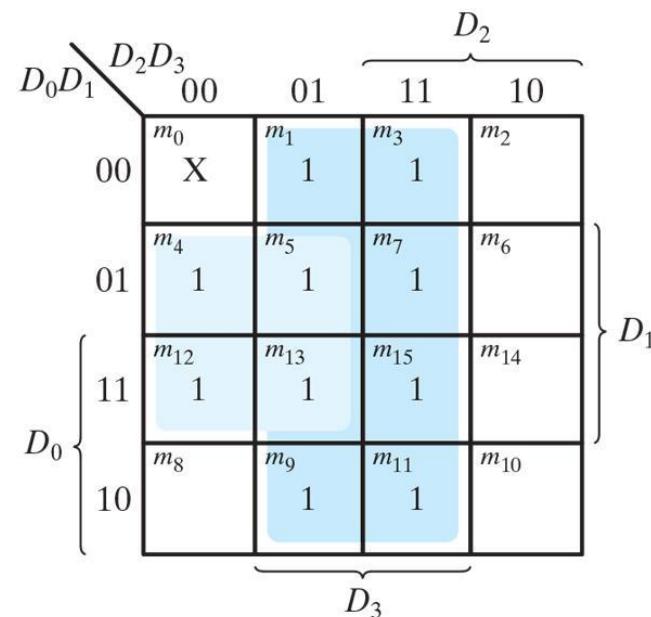
$$x = D_2 + D_3$$

$$y = D_3 + D_1 D_2^2$$

$$V = D_0 + D_1 + D_2 + D_3$$



$$x = D_2 + D_3$$

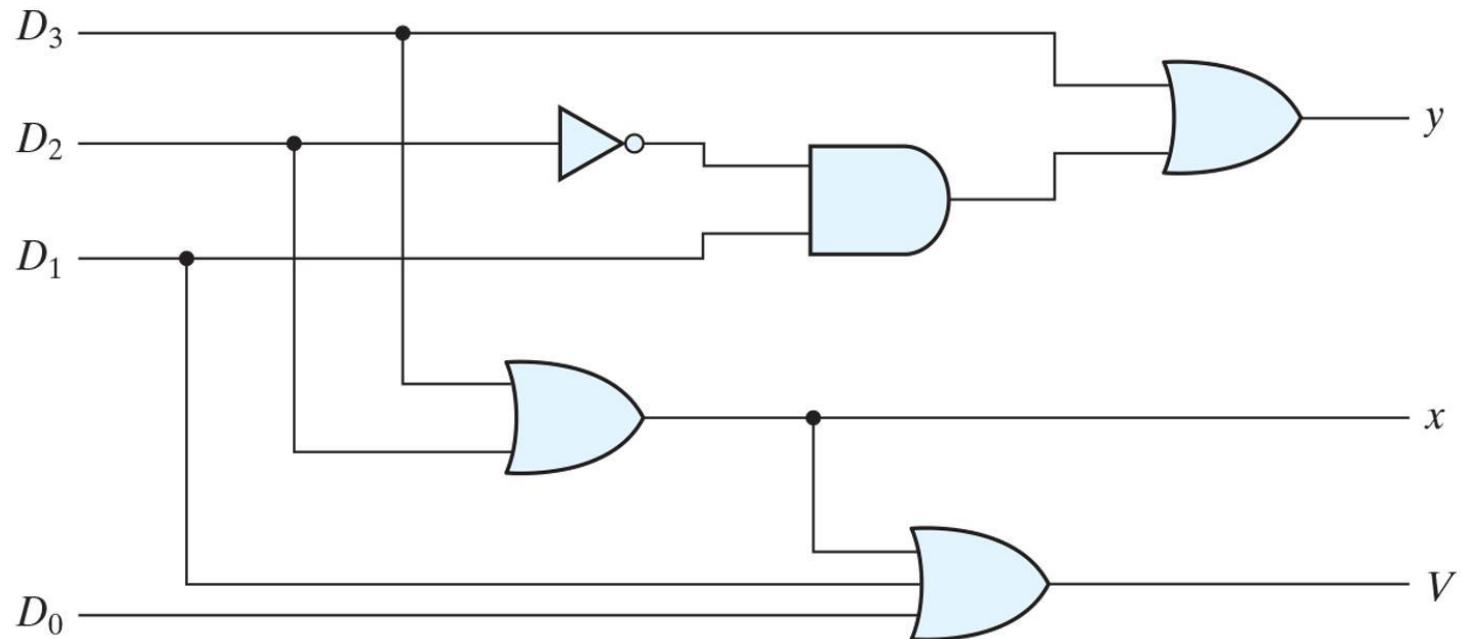


$$y = D_3 + D_1 D'{}_2$$

# Combinational logic

## Priority Encoders

Four input Priority Encoder



# Combinational logic

## Priority Encoders

- 8-to-3 Priority Encoder

- What if more than one input line has a value of 1?
- Ignore “lower priority” inputs.
- **Idle** indicates that no input is a 1.
- Note that polarity of **Idle** is opposite from Table 4-8 in Mano

Inputs								Outputs			Idle
I <sub>0</sub>	I <sub>1</sub>	I <sub>2</sub>	I <sub>3</sub>	I <sub>4</sub>	I <sub>5</sub>	I <sub>6</sub>	I <sub>7</sub>	y <sub>2</sub>	y <sub>1</sub>	y <sub>0</sub>	Idle
0	0	0	0	0	0	0	0	x	x	x	1
1	0	0	0	0	0	0	0	0	0	0	0
X	1	0	0	0	0	0	0	0	0	1	0
X	X	1	0	0	0	0	0	0	1	0	0
X	X	X	1	0	0	0	0	0	1	1	0
X	X	X	X	1	0	0	0	1	0	0	0
X	X	X	X	X	1	0	0	1	0	1	0
X	X	X	X	X	X	1	0	1	1	0	0
X	X	X	X	X	X	X	1	1	1	1	0

# Combinational logic

## Encoders

Priority Encoder (8 to 3 encoder)

Assign priorities to the inputs

When more than one input are asserted, the output generates the code of the input with the highest priority

Priority Encoder :

$$H7 = I7 \quad (\text{Highest Priority})$$

$$H6 = I6 \cdot I7'$$

$$H5 = I5 \cdot I6' \cdot I7'$$

$$H4 = I4 \cdot I5' \cdot I6' \cdot I7'$$

$$H3 = I3 \cdot I4' \cdot I5' \cdot I6' \cdot I7'$$

$$H2 = I2 \cdot I3' \cdot I4' \cdot I5' \cdot I6' \cdot I7'$$

$$H1 = I1 \cdot I2' \cdot I3' \cdot I4' \cdot I5' \cdot I6' \cdot I7'$$

$$H0 = I0 \cdot I1' \cdot I2' \cdot I3' \cdot I4' \cdot I5' \cdot I6' \cdot I7'$$

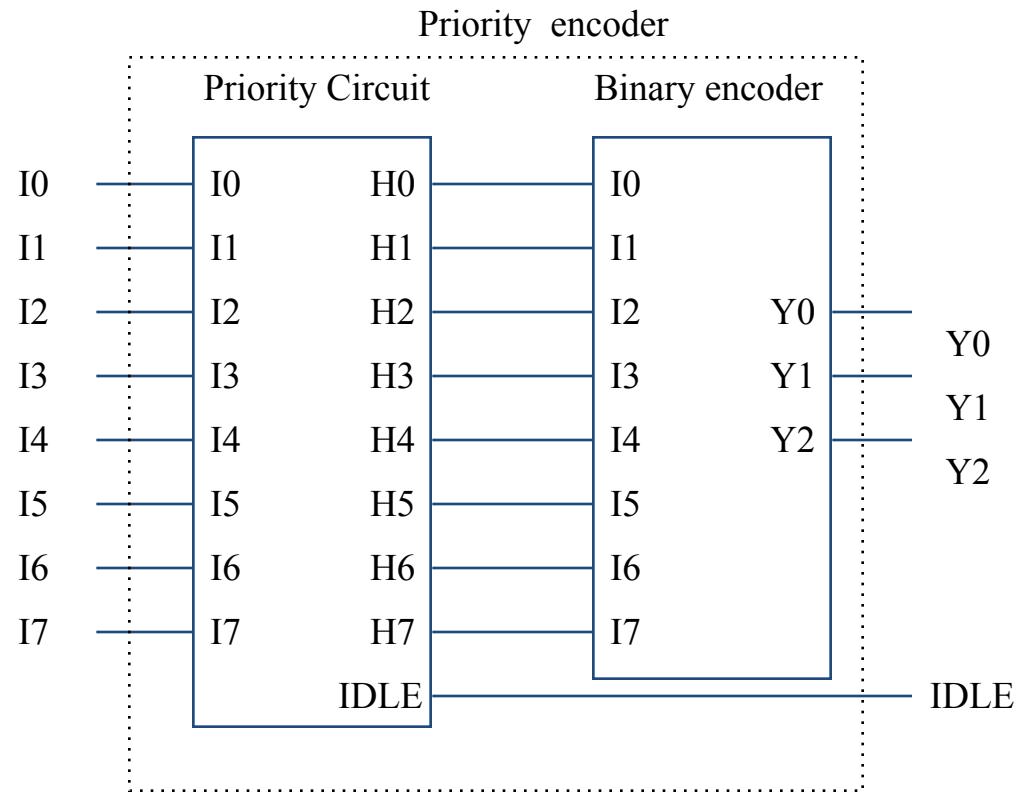
$$\text{IDLE} = I0' \cdot I1' \cdot I2' \cdot I3' \cdot I4' \cdot I5' \cdot I6' \cdot I7'$$

Encoder

$$Y_0 = I_1 + I_3 + I_5 + I_7$$

$$Y_1 = I_2 + I_3 + I_6 + I_7$$

$$Y_2 = I_4 + I_5 + I_6 + I_7$$

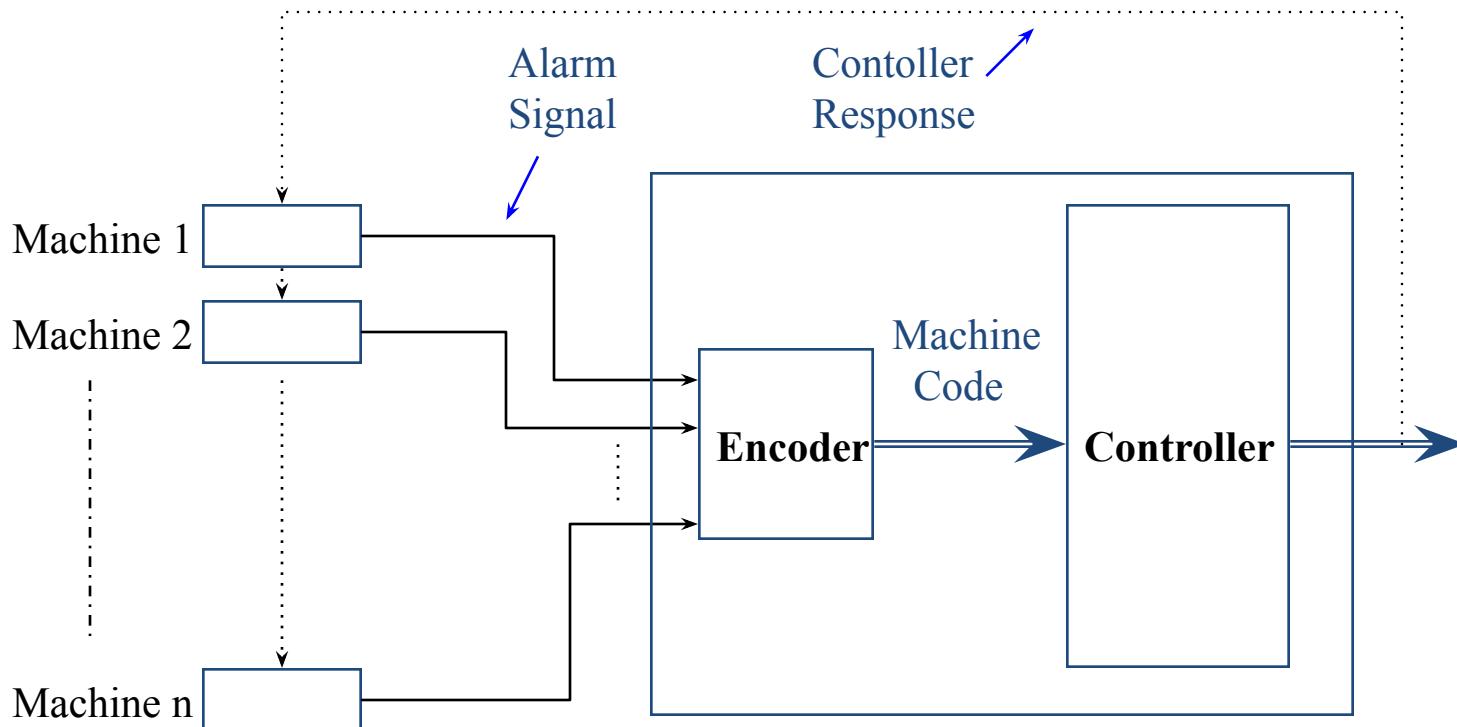


# Combinational logic

## Encoders

### Encoder Application (Monitoring Unit)

Encoder identifies the requester and encodes the value  
Controller accepts digital inputs.



# Combinational logic

## Encoders

### Machines (Machine 1, Machine 2, ... Machine n):

- These are the different machines connected to the encoder. Each machine sends data or a request to the encoder.

### Encoder:

- **The encoder's primary role is to identify which machine is sending the request** and then encode the value or information from that machine into a machine code.
- It also generates an alarm signal when needed, possibly indicating an issue or a specific condition that requires attention.

### controller:

- The controller receives the encoded machine code from the encoder.
- Based on the received machine code, the **controller performs specific actions or responses, such as sending signals back to the machines.**

### Alarm Signal:

- **The alarm signal seems to be an alert generated by the encoder if a certain condition is met.** This could indicate a **malfunction**, a process completion, or any other situation that needs immediate attention.

### Controller Response:

- After processing the machine code, **the controller might send a response back to the machine or trigger other actions as needed.**

# Combinational logic

## Encoders

### Applications:

Encoders- Application: In digital systems like computers, a keyboard encoder converts the pressing of keys into binary codes that the computer's processor can understand and act upon.

Priority encoders are commonly used in interrupt systems in microprocessors, where multiple interrupt signals may be received at the same time, and the processor needs to know which one to service first based on priority.

Decoders- In memory systems, a decoder takes the binary address provided by the CPU and activates the corresponding memory cell, allowing data to be read from or written to that specific location. For example, a 3-to-8 decoder can select one of eight memory locations based on a 3-bit address input.

# Combinational logic

## Encoders- think about it

---

Which of the following best describes the functionality of a **priority encoder**?

- A. It encodes the input having the lowest subscript among all active inputs.
- B. It encodes only the first input line, irrespective of other inputs.
- C. It encodes the input having the highest subscript (priority) among active inputs.
- D. It generates output only when a single input is active.

In a **4-to-2 priority encoder**, the output is “11” when:

- A. Only D0 is 1
- B. D1 and D2 are 1
- C. D3 is 1
- D. D3 is 0 and others are 1

# Combinational logic

## Encoders- think about it

---

Which of the following best describes the functionality of a **priority encoder**?

- A. It encodes the input having the lowest subscript among all active inputs.
- B. It encodes only the first input line, irrespective of other inputs.
- C. It encodes the input having the highest subscript (priority) among active inputs.
- D. It generates output only when a single input is active.

**Answer:** C

**Explanation:** In a priority encoder, if multiple inputs are active, the one with the **highest priority (usually highest subscript)** is considered.

In a **4-to-2 priority encoder**, the output is “11” when:

- A. Only D0 is 1
- B. D1 and D2 are 1
- C. D3 is 1
- D. D3 is 0 and others are 1

**Answer:** C

**Explanation:** D3 has highest priority.



# THANK YOU

---



**Team DDCO**  
Department of Computer Science



**PES**

UNIVERSITY

CELEBRATING 50 YEARS

# DIGITAL DESIGN AND COMPUTER ORGANIZATION

## Multiplexers

**Team DDCO**

**Department of Computer Science and Engineering**

# **DIGITAL DESIGN AND COMPUTER ORGANIZATION**

---



## **Multiplexers**

Department of Computer Science and Engineering

# Combinational logic

## Multiplexers

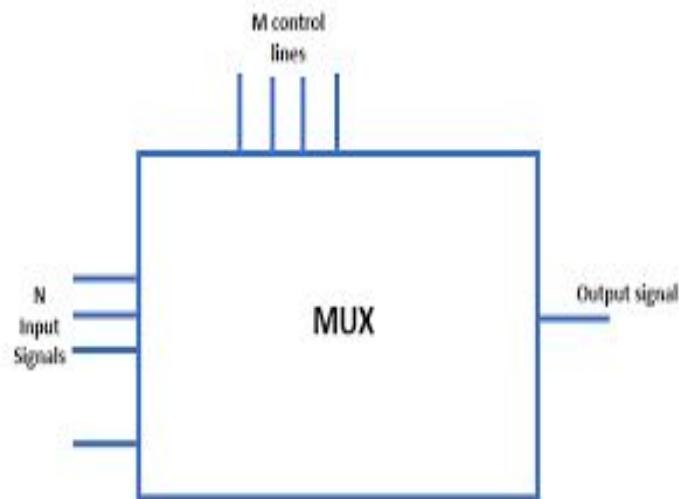
A multiplexer (also called a mux) *multiplexes* many inputs onto a single output

Data selector

Many to one

N:1 MUX

$2^N$  inputs n select lines , one output



[www.educba.com](http://www.educba.com)



# Combinational logic

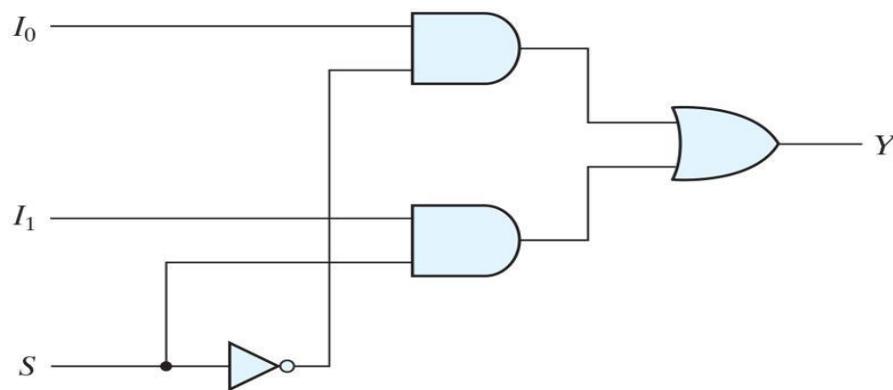
## Multiplexers

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

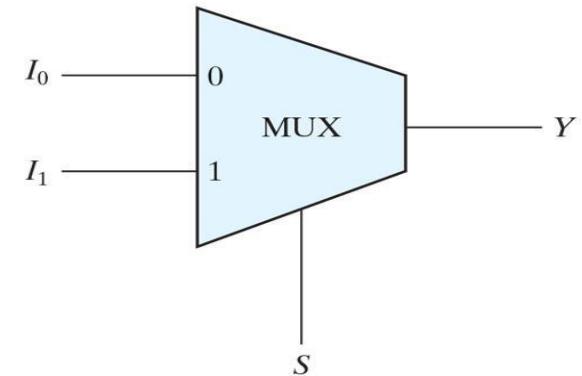
# Combinational logic

## Multiplexers

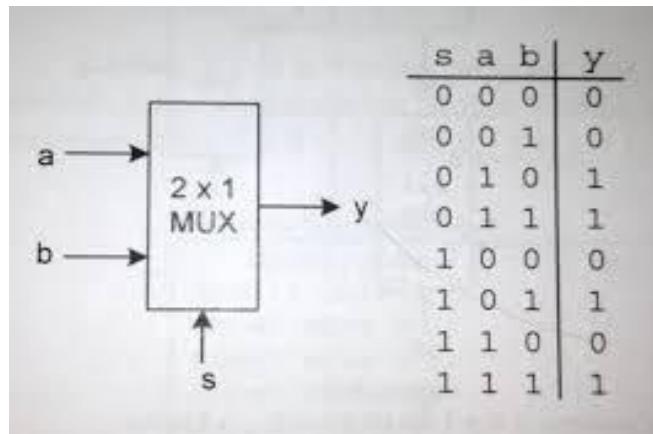
Two to One Line Multiplexer



(a) Logic diagram



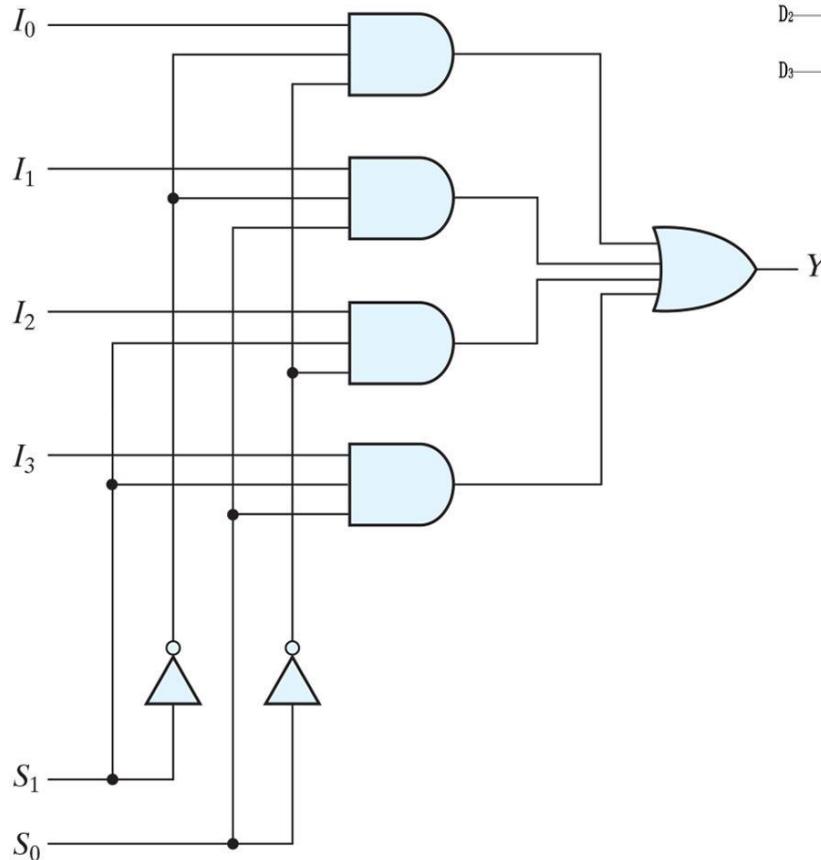
(b) Block diagram



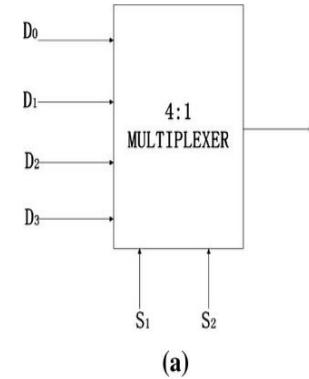
# Combinational logic

## Multiplexers

### Four to One Line Multiplexer



(a) Logic diagram



(a)

INPUT	OUTPUT	
$S_2$	$S_1$	$Y$
0	0	$D_0$
0	1	$D_1$
1	0	$D_2$
1	1	$D_3$

(b)

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

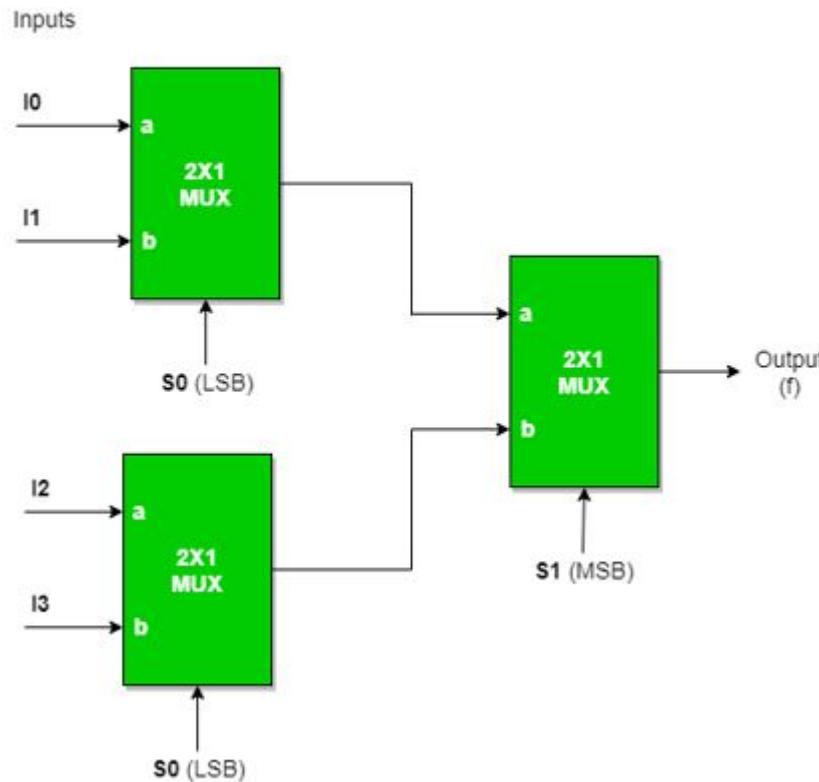
(b) Function table

# Combinational logic

## Multiplexers

Implementing 4:1 MUX using 2:1 MUX

Implementing 4:1 MUX using 2:1 MUX



Truth Table

$S_1$	$S_0$	$f$
0	0	$I_0$
0	1	$I_1$
1	0	$I_2$
1	1	$I_3$

# Combinational logic

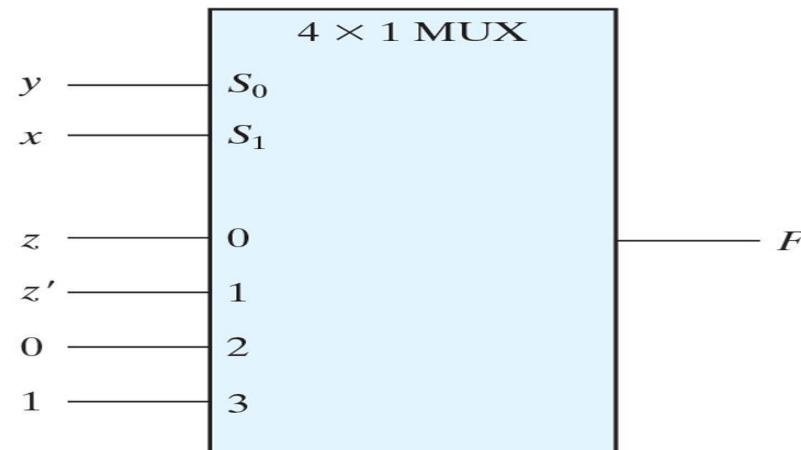
## Multiplexers-Boolean Function Implementation

Implementing a Boolean Function with a Multiplexer

$$F(x, y, z) = (1, 2, 6, 7)$$

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

(a) Truth table



(b) Multiplexer implementation

Connect input variables to select inputs of multiplexer ( $n-1$  for  $n$  variables)

Set data inputs to multiplexer equal to values of function for corresponding assignment of select variables

Using a variable at data inputs reduces size of the multiplexer

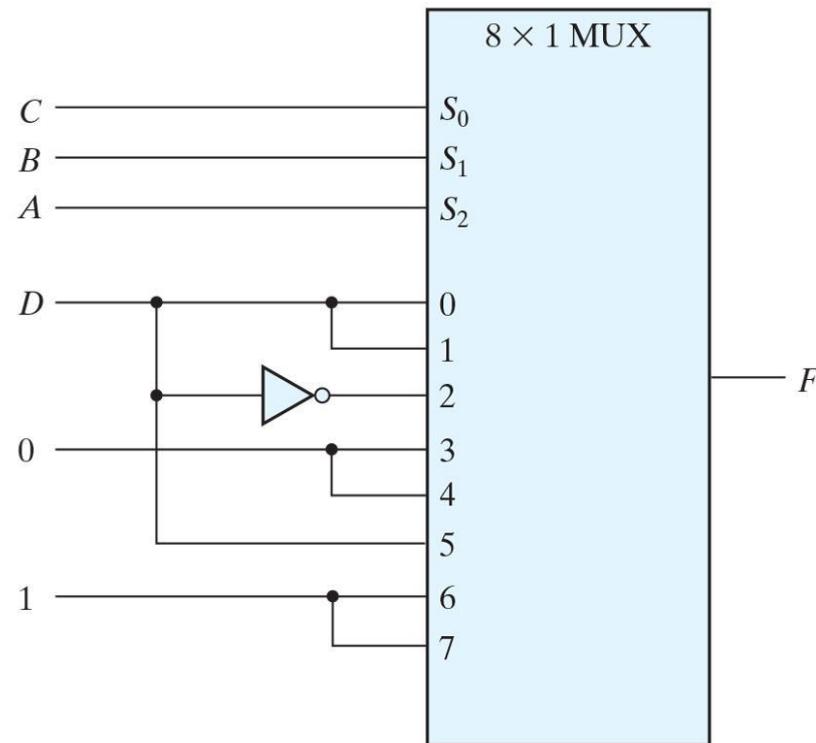
# Combinational logic

## Multiplexers

Implementing a four input Function with a 8X1 Multiplexer

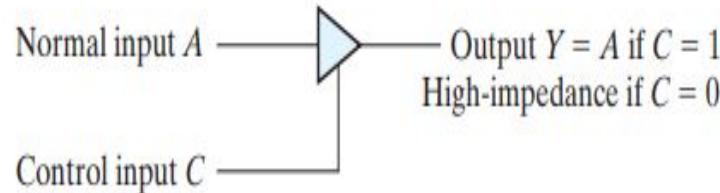
$$F(A, B, C, D) = (1, 3, 4, 11, 12, 13, 14, 15)$$

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



# Combinational logic

## Multiplexers- Three State Gates



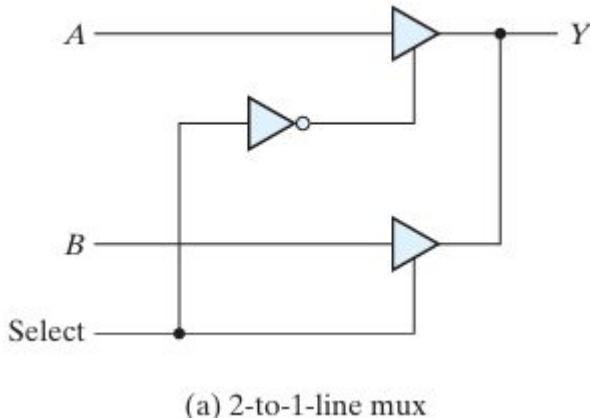
**FIGURE 4.29**

Graphic symbol for a three-state buffer

Two of the states are signals equivalent to logic 1 and logic 0 as in a conventional gate. The third state is a high-impedance state in which (1) the logic behaves like an **open circuit**, which means that the output appears to be disconnected, (2) the circuit has **no logic significance**, and (3) the circuit connected to the **output** of the **three-state gate is not affected by the inputs to the gate**.

# Combinational logic

## Multiplexers- Three State Gates



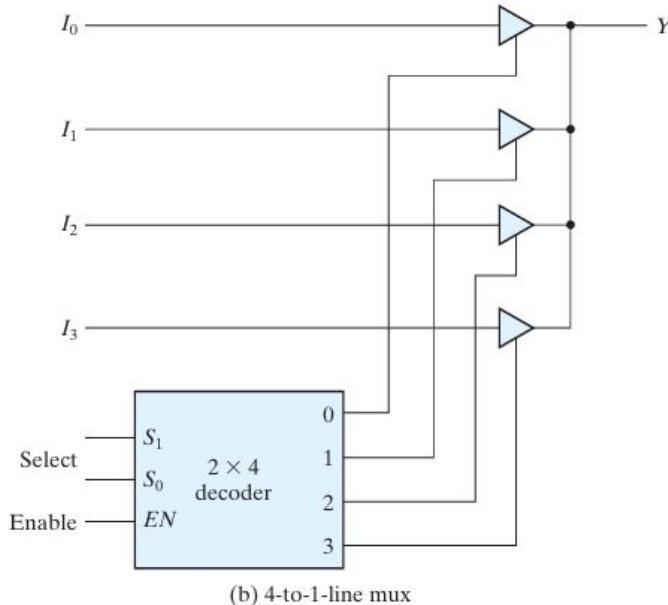
**FIGURE 4.30**  
Multiplexers with three-state gates

When the select input is 0, the upper buffer is enabled by its control input and the lower buffer is disabled. Output Y is then equal to input A. When the select input is 1, the lower buffer is enabled and Y is equal to B.

The construction of multiplexers with three-state buffers is demonstrated in Fig. 4.30 . Figure 4.30(a) shows the construction of a two-to-one-line multiplexer with 2 three-state buffers and an inverter. The two outputs are connected together to form a single output line.

# Combinational logic

## Multiplexers- Three State Gates



No more than one buffer may be in the active state at any given time.

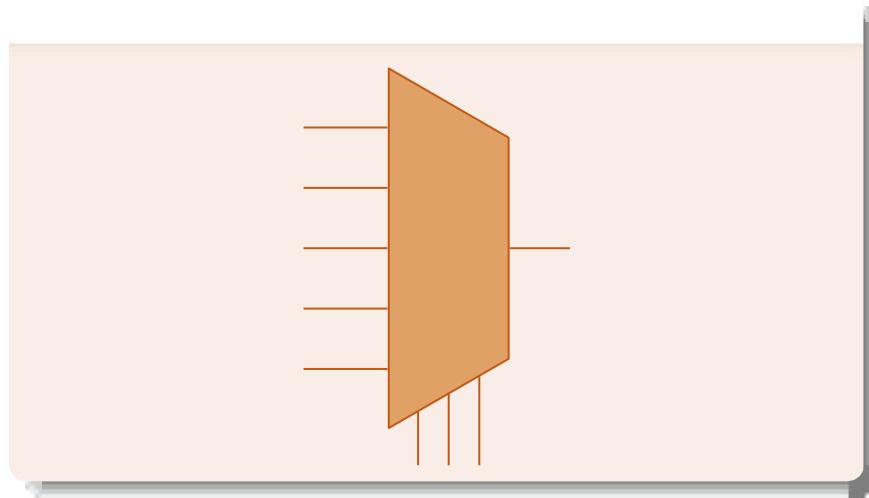
One way to ensure that no more than one control input is active at any given time is to use a decoder, as shown in the diagram

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

# Combinational logic

## Multiplexers- Think about it

5:1 Mux



A combinational logic circuit having  $n$  data inputs,  $\lceil \log_2 n \rceil$  control inputs and one output, that connects the data input indicated by the control inputs to the output

- What is the Boolean formula for a 3:1 mux? Construct a 3:1 mux using 2:1 muxes AND, OR and NOT gates

# Combinational logic

## Multiplexers- Think about it

---

Using 2 X 1 MUX design  $y=AB$

Using 4 X 1 MUX design  $y=AB' + B'C' + A'BC$

# Combinational logic

## Multiplexers-

---

A 4-to-1 multiplexer is used to implement the Boolean function

$$F(A,B,C) = \sum m(1,3,5,7)$$

Which variables should be connected to the select lines of the MUX?

- A) A and B
- B) B and C
- C) A and C
- D) Any two variables

How many select lines are required for a **5:1 multiplexer**?

- A) 2
- B) 3
- C) 4
- D) 5

# Combinational logic

## Multiplexers-

---

A 4-to-1 multiplexer is used to implement the Boolean function

$$F(A,B,C) = \sum m(1,3,5,7)$$

Which variables should be connected to the select lines of the MUX?

- A) A and B
- B) B and C
- C) A and C
- D) Any two variables

Answer: D

. How many select lines are required for a **5:1 multiplexer**?

- A) 2
- B) 3
- C) 4
- D) 5

**Answer: B) 3**

# Combinational logic

## Multiplexers-

---

The Boolean function  $F(x,y,z)=\sum m(1,2,6,7)$  is implemented using an 8:1 multiplexer. What should be the values of data inputs  $D_0$  to  $D_7$ ?

- A)  $D = \{1,1,1,0,0,0,1,1\}$
- B)  $D = \{0,1,0,1,0,1,0,1\}$
- C)  $D = \{1,0,0,1,1,1,0,0\}$
- D)  $D = \{0,1,1,0,0,0,1,1\}$

# Combinational logic

## Multiplexers-

---

The Boolean function  $F(x,y,z)=\sum m(1,2,6,7)$  is implemented using an 8:1 multiplexer. What should be the values of data inputs  $D_0$  to  $D_7$ ?

- A)  $D = \{1,1,1,0,0,0,1,1\}$
- B)  $D = \{0,1,0,1,0,1,0,1\}$
- C)  $D = \{1,0,0,1,1,1,0,0\}$
- D)  $D = \{0,1,1,0,0,0,1,1\}$

Answer: D)  $D = \{0,1,1,0,0,0,1,1\}$

Explanation:

The minterms correspond to the positions where  $F = 1$ .

So,  $D_1 = D_2 = D_6 = D_7 = 1$ , rest are 0.



# THANK YOU

---



**Team DDCO**  
Department of Computer Science



# DIGITAL DESIGN AND COMPUTER ORGANIZATION

## Synchronous Sequential Logic

---

**Team DDCO**

**Department of Computer Science and Engineering**

# Introduction- T1- section 5.1

---

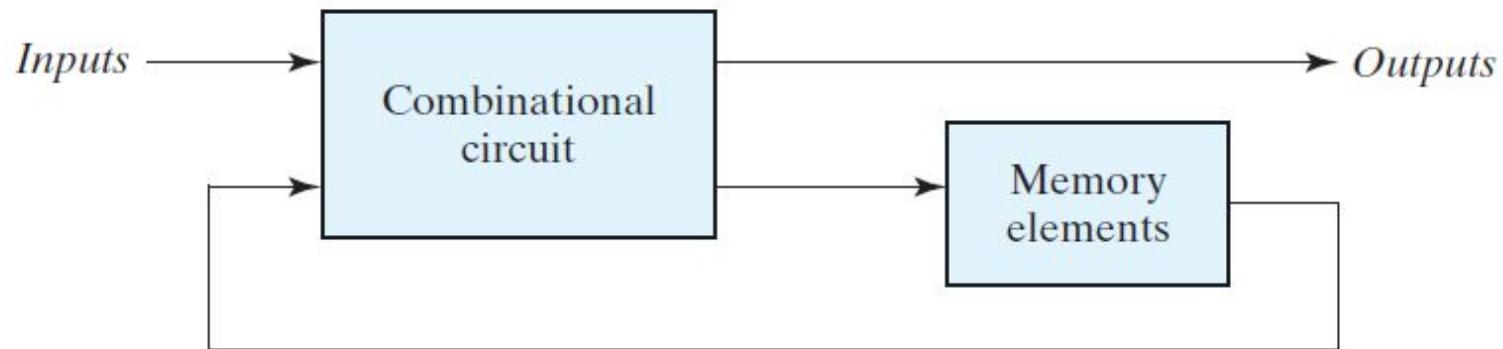
- Hand-held devices, cell phones, navigation receivers, personal computers, digital cameras, personal media players, and virtually all electronic consumer products have the ability to send, receive, store, retrieve, and process information represented in a binary format.
- The technology enabling and supporting these devices is critically dependent on electronic components that can store information, i.e., have memory.
- This chapter examines the operation and control of these devices and their use in circuits.

# Introduction

---

- The digital circuits considered thus far have been **combinational**.
- **Their output depends only and immediately on their inputs—they have no memory, i.e., dependence on past values of their inputs.**
- Sequential circuits, however, act as **storage elements and have memory**.
- They can store, retain, and then retrieve information when needed at a later time.

# Sequential Circuits (T1- section 5.2)



# Sequential Circuits

---

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

# Sequential Circuits: Key Points

---

- Depends on current and prior input
- Memory element
- Feedback/cycles
- Remember previous input
- Next state= external input + present state
- State of system-binary information stored in system
- State variables

Thus, **a sequential circuit is specified by a time sequence of inputs, outputs, and internal states.**

# Types of Sequential Circuits

---

There are two main types of sequential circuits, and their classification is a function of the timing of their signals.

1. A **synchronous sequential circuit** is a system whose behavior can be defined from the knowledge of its signals at discrete instants of time.
2. The behavior of an **asynchronous sequential circuit** depends upon the **input signals** at any instant of time and the order in which the inputs change.

# Synchronous Sequential Circuit

---

- A synchronous sequential circuit employs signals that affect the storage elements at only discrete instants of time.
- Synchronization is achieved by a timing device called a **clock generator**, which provides a clock signal having the form of a periodic train of clock pulses.
- The clock signal is commonly denoted by the identifiers ***clock*** and ***clk***.
- The **clock pulses are distributed throughout the system in such a way that storage elements are affected only with the arrival of each pulse.**

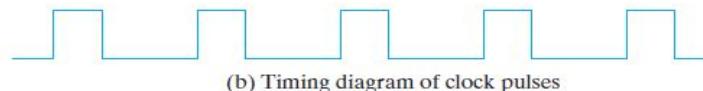
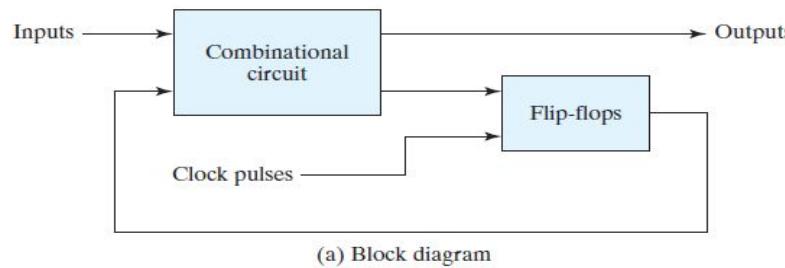
# Synchronous Sequential Circuit

---

- In practice, the clock pulses determine when computational activity will occur within the circuit, and other signals (external inputs and otherwise) determine what changes will take place affecting the storage elements and the outputs.
- For example, a **circuit that is to add and store two binary numbers would compute their sum from the values of the numbers and store the sum at the occurrence of a clock pulse.**
- Synchronous sequential circuits that use clock pulses to control storage elements are called **clocked sequential circuits**

# Synchronous Sequential Circuit

- The storage elements (memory) used in clocked sequential circuits are **called flip flops**. A flip-flop is a binary storage device capable of storing one bit of information. In a stable state, the output of a flip-flop is either **0 or 1**



- The new value is stored (i.e., the flip-flop is updated) when a pulse of the clock signal occurs

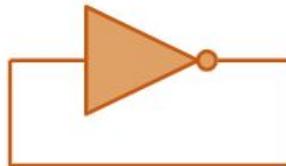
# Synchronous Sequential Circuit

---

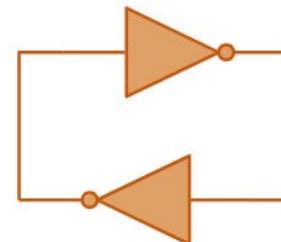
- Propagation delays play an important role in determining the minimum interval between clock pulses that will allow the circuit to operate correctly. **A change in state of the flip-flops is initiated only by a clock pulse transition—for example, when the value of the clock signals changes from 0 to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

# How to Implement Memory?

- The inverter is essentially the simplest logic gate
- What happens when we connect its output to input, forming a loop?
- Inverter loop is thus not in a **stable state**



- How about a loop of two inverters?
- A two-inverter loop can be in one of **two stable states**.
- A bit can have one of **two different values**. Thus, a two-inverter loop can **store** (or “remember”) a single bit.
- But how can we **change the bit stored** or the **stable state**?



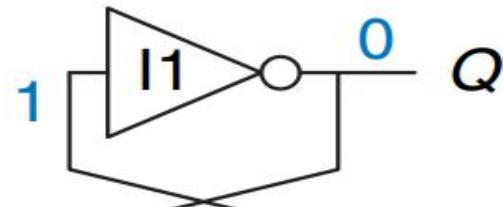
# Storage Using Cross-Coupled Inverters

---

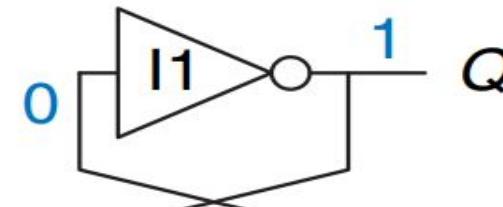
- **Cross-coupled inverters** form a basic memory element with **two stable states**.
- A system with **N stable states** can store up to  **$\log_2(N)$  bits of information**.
- Therefore, a **bistable element** (with 2 stable states) can store **1 bit** of information.

# Storage Using Cross-Coupled Inverters

## Bistable Operation of Cross-Coupled Inverters



(a)



(b)

# Storage Using Cross-Coupled Inverters

---

- When power is first applied to sequential circuit, initial state is unknown.
- It changes each time clock is on, it is unpredictable which is not desirable.
- **Cross coupled inverters are not practical because user has no input to control the state**
- **Hence we have flipflops as memory elements**

# Asynchronous Sequential Circuit

---

- The storage elements commonly used in asynchronous sequential circuits are **time-delay devices**.
- The storage capability of a time-delay device varies with the time it takes for the signal to propagate through the device.
- An asynchronous sequential circuit is **one where the outputs (and next states) can change immediately when inputs change, without waiting for a clock pulse**.
- In gate-type asynchronous systems, **the storage elements consist of logic gates whose propagation delay provides the required storage**.
- Thus, an **asynchronous sequential circuit may be regarded as a combinational circuit with feedback**.

# Flip-Flops-key points

---

- The storage elements (memory) used in clocked sequential circuits are called flip-flops.
- A flip-flop is a binary storage device capable of storing one bit of information.
- In a stable state, the output of a flip-flop is either 0 or 1
- The new value is stored (i.e., the flip-flop is updated) when a pulse of the clock signal occurs.
- A change in state of the flip-flops is initiated only by a clock pulse transition, eg: when the value of the clock signals changes from 0 to 1
- Thus, the transition from one state to the next occurs only at predetermined intervals dictated by the clock pulses.

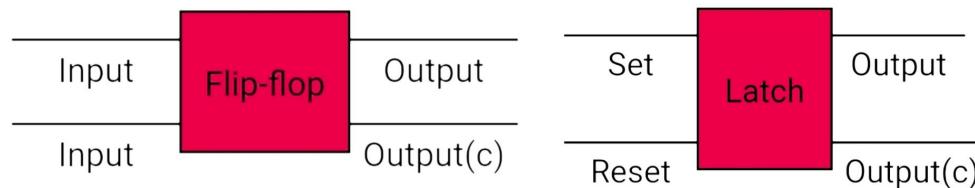
# Latches

---

- A storage element in a digital circuit can maintain a binary state indefinitely (as long as power is delivered to the circuit), until directed by an input signal to switch states.
- The major differences among various types of storage elements are in the number of inputs they possess and in the manner in which the inputs affect the binary state.
- Storage elements that operate with **signal levels** (rather than signal transitions) are referred to as latches ; those controlled by a **clock transition are flip-flops** .
- Latches are said to be **level sensitive devices**; flip-flops are edge-sensitive devices.

# Latches

- The two types of storage elements are related because latches are the basic circuits from which all flip-flops are constructed.
- Although latches are useful for storing binary information and for the design of asynchronous sequential circuits, they are not practical for use as storage elements in synchronous sequential circuits.
- They are the building blocks of flip-flops.



Difference between flip flop and latch

# Difference b/w Latches & Flip-Flops

Latches	Flip-Flops
Latches are building blocks of sequential circuits, these can be built from <b>logic gates</b>	Flip flops are also building blocks of sequential circuits. But, these can be built from the <b>latches</b> .
Latch continuously <b>checks its inputs and changes its output</b> correspondingly.	Flip flop continuously checks its inputs and changes its output correspondingly only at times determined by <b>clocking signal</b>
The latch is <b>sensitive to the duration of the pulse</b> and can send or receive the data when the switch is on.	Flipflop is sensitive to a <b>signal change</b> . They can transfer data only at the single instant and data cannot be changed until next signal change. Flip flops are used as a register.
It is based on the enable function input.	It works on the basis of clock pulses.
It is a level triggered, it means that the output of the present state and input of the next state depends on the level that is binary input 1 or 0. ( <b>asynchronous circuits</b> )	It is an edge triggered, it means that the output and the next state input changes when there is a change in clock pulse whether it may a+ve or-ve clock pulse. ( <b>synchronous circuits</b> )

# Difference b/w Latches & Flip-Flops

---

## Example: Traffic Light Controller

### Scenario:

A city traffic light controller uses digital storage elements to decide when to change from Green → Yellow → Red.

### Using a Latch (Level-triggered)

Suppose the system uses a **latch** to store the current state of the traffic light.

Since a latch is **level-sensitive**, whenever the clock/enable signal is **high**, the latch is “transparent” — it immediately passes input changes to the output.

**Problem:** If there is noise or a small glitch during the enable-high period, the latch will capture it. This could cause the traffic light to change state unexpectedly (e.g., Green jumps to Red without Yellow).

Application-level interpretation: **Not safe** for traffic control, because continuous input changes during the active level may corrupt the output.

# Difference b/w Latches & Flip-Flops

---

## Example: Traffic Light Controller

### Scenario:

A city traffic light controller uses digital storage elements to decide when to change from Green → Yellow → Red.

### Using a Flip-Flop (Edge-triggered)

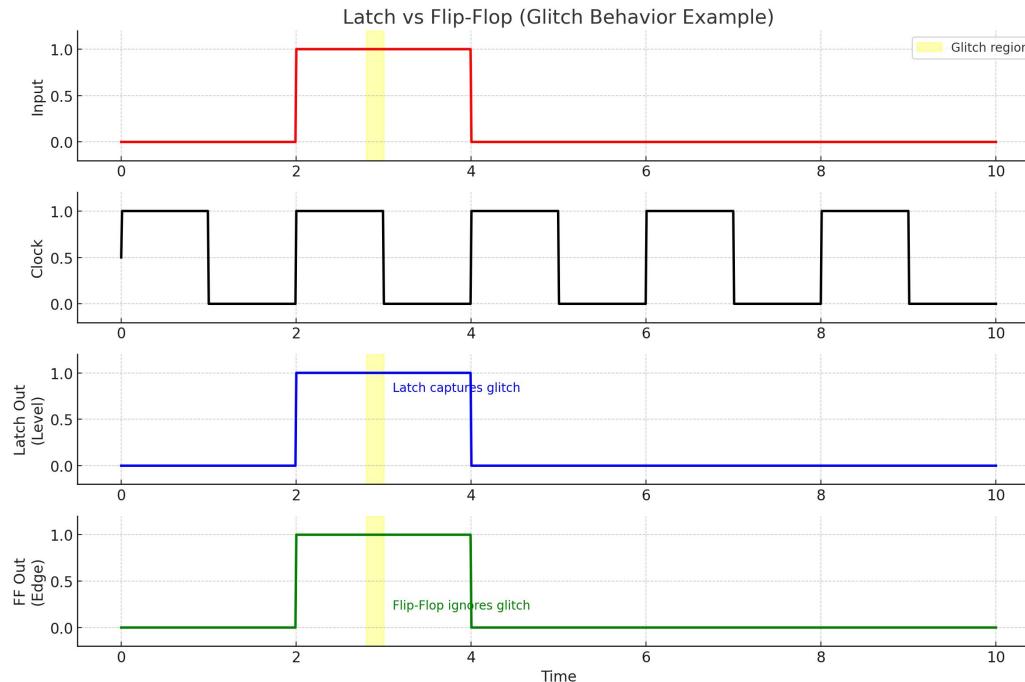
If the system uses a **Flip-Flop**, the light state is updated **only at the clock's rising (or falling) edge**.

Input changes happening in between edges are ignored until the next edge.

**Advantage:** The traffic light changes state in a controlled manner (e.g., every 30 seconds on the rising clock edge). Even if input glitches occur while the clock is low, they won't affect the state.

Application-level interpretation: **Much safer and reliable** for real-world sequential control (traffic lights, elevators, etc.).

# Difference b/w Latches & Flip-Flops



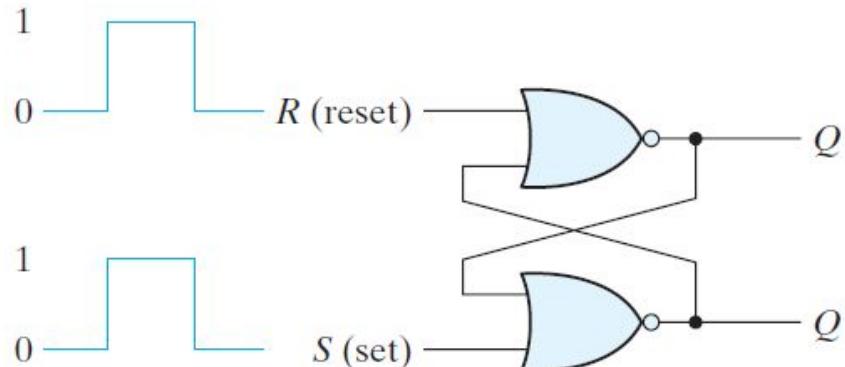
Yellow region → glitch in the input.

Latch Output (blue) → immediately captures the glitch while clock is HIGH.

Flip-Flop Output (green) → ignores the glitch, updates only at the clock edge.

# SR Latch (Using NOR Gates) T1- section 5.3

- The SR latch is a circuit with two cross-coupled NOR gates, and two inputs labeled S for set and R for reset. The latch has two useful states.
- When output  $Q = 1$  and  $Q' = 0$ , the latch is said to be in the *set state*.
- When  $Q = 0$  and  $Q' = 1$ , it is in the *reset state*.
- Outputs Q and Q' are normally the complement of each other.



(a) Logic diagram

S	R	Q	$Q'$
1	0	1	0
0	0	1	0
0	1	0	1
0	0	0	1
1	1	0	0

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

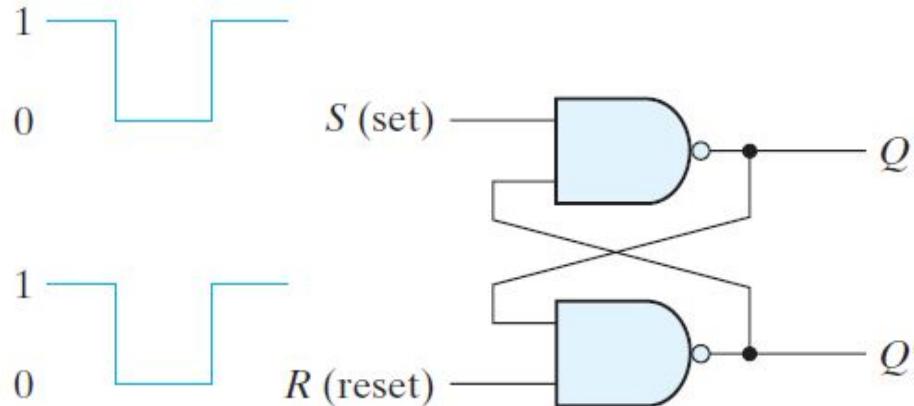
(b) Function table

Truth table of NOR Gate

Input A	Input B	$0 = (A + B)'$
0	0	1
0	1	0
1	0	0
1	1	0

# SR Latch (Using NAND Gates)

- The SR latch is a circuit with two cross-coupled NAND gates and two inputs labeled S for set and R for reset.
- It operates with both inputs **normally at 1**, unless the state of the latch has to be changed.



(a) Logic diagram

Truth table of NAND Gate

A	B	Output
0	0	1
1	0	1
0	1	1
1	1	0

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

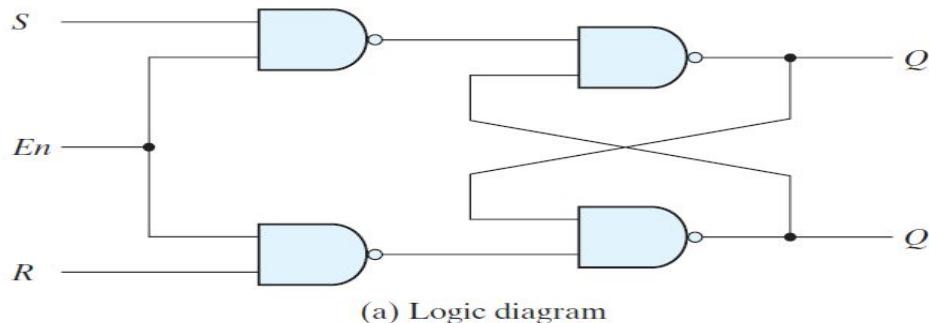
# SR Latch (Using NAND Gates)

---

- The application of **0 to the S input** causes output Q to go to 1, putting the latch in the **set state**.
- When the S input goes back to 1, the circuit remains in the set state.
- After both inputs go back to 1, we are allowed to change the state of the latch by placing a **0 in the R input**.
- This action causes the circuit to go to the **reset state** and stay there even after both inputs return to 1.
- The condition that is forbidden for the NAND latch is **both inputs being equal to 0 at the same time**, an input combination that should be **avoided**.

# SR Latch with Control Input

- In comparing the NAND with the NOR latch, note that the input signals for the NAND require the complement of those values used for the NOR latch. Because the NAND latch requires a 0 signal to change its state, **it is sometimes referred to as an S'R' latch**.
- The operation of the basic SR latch can be modified by providing an **additional input signal** that determines (controls) when the state of the latch can be changed by determining whether S and R (or S' and R') can affect the circuit.
- It consists of the basic SR latch and two additional NAND gates.



<i>En</i>	<i>S</i>	<i>R</i>	Next state of <i>Q</i>
0	X	X	No change
1	0	0	No change
1	0	1	$Q = 0$ ; reset state
1	1	0	$Q = 1$ ; set state
1	1	1	Indeterminate

(b) Function table

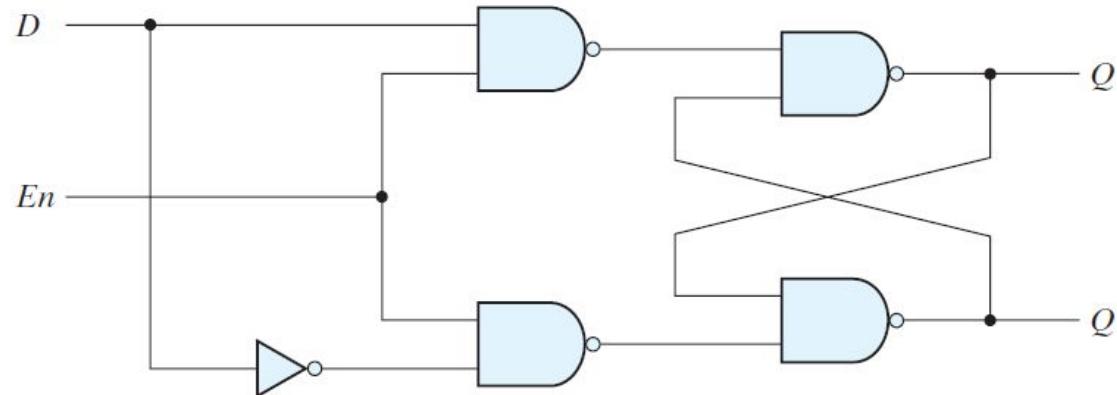
# SR Latch with Control Input

---

- The control input **En** acts as an enable signal for the other two inputs.
- The outputs of the NAND gates stay at the logic-1 level as long as the enable signal remains at 0.
- This is the quiescent condition for the SR latch.
- When the **enable input goes to 1, information from the S or R input is allowed to affect the latch.**
- The set state is reached with  $S = 1$ ,  $R = 0$ , and  $En = 1$ . (active-high enabled).
- To change to the reset state, the inputs must be  $S = 0$ ,  $R = 1$ , and  $En = 1$ .
- In either case, **when En returns to 0, the circuit remains in its current state.**
- The control input disables the circuit by applying 0 to En, so that the state of the output does not change regardless of the values of S and R .
- An indeterminate condition occurs when all three inputs are equal to 1.

# D Latch(Transparent Latch)

- One way to eliminate the undesirable condition of the indeterminate state in the SR latch is to ensure that inputs S and R are never equal to 1 at the same time. This is done in the D latch.
- This latch has only two inputs: D (data) and En (enable).



(a) Logic diagram

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

(b) Function table

# D Latch

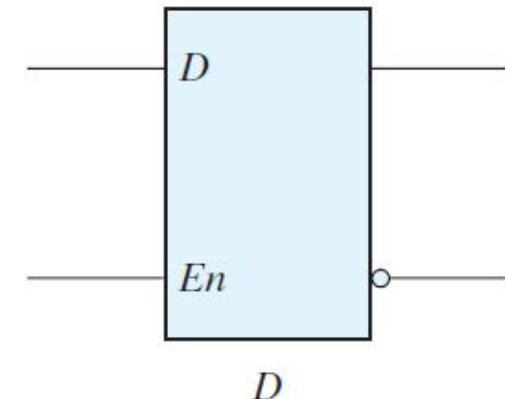
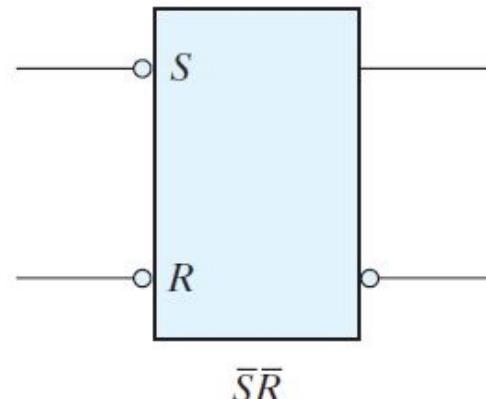
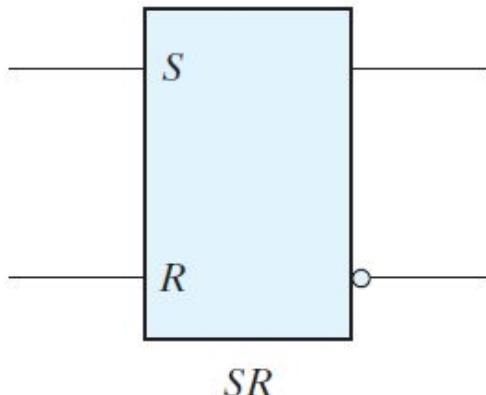
---

- As long as the **enable input is at 0**, the cross-coupled SR latch has both inputs at the 1 level and the **circuit cannot change state regardless of the value of D**.
- The binary information present at the data input of the D latch is transferred to the Q output when the enable input is asserted.
- **The output follows changes in the data input as long as the enable input is asserted.**
- This situation provides a path from input D to the output, and for this reason, the circuit is often called a **transparent latch**.
- When the **enable input signal is de-asserted**, the binary information that was present at the data input at the time the transition occurred is **retained** (i.e., stored) at the Q output until the enable input is asserted again.

# Graphic Symbols for Latches

---

- A latch is designated by a rectangular block with inputs on the left and outputs on the right.
- One output designates the normal output, and the other (with the bubble designation) designates the complement output.



# Flip-Flops T1- section 5.4

---

- When latches are used for the storage elements, a serious difficulty arises. The state transitions of the latches start as soon as the clock pulse changes to the logic-1 level. **The new state of a latch appears at the output while the pulse is still active.**
- **If the inputs applied to the latches change while the clock pulse is still at the logic-1 level,** the latches will respond to new values and a new output state may occur. The result is an unpredictable situation.
- **A flipflop triggers only during a signal transition (from 0 to 1 or from 1 to 0) of the synchronizing signal (clock) and is disabled during the rest of the clock pulse.**

# Clock Response in Latch and Flip-Flop



(a) Response to positive level

The problem with the latch is that it responds to a change in the level of a clock pulse.



(b) Positive-edge response

The key to the proper operation of a flip-flop is to trigger it only during a signal transition



(c) Negative-edge response

A clock pulse goes through two transitions: from 0 to 1 and the return from 1 to 0.

# Flip-Flop

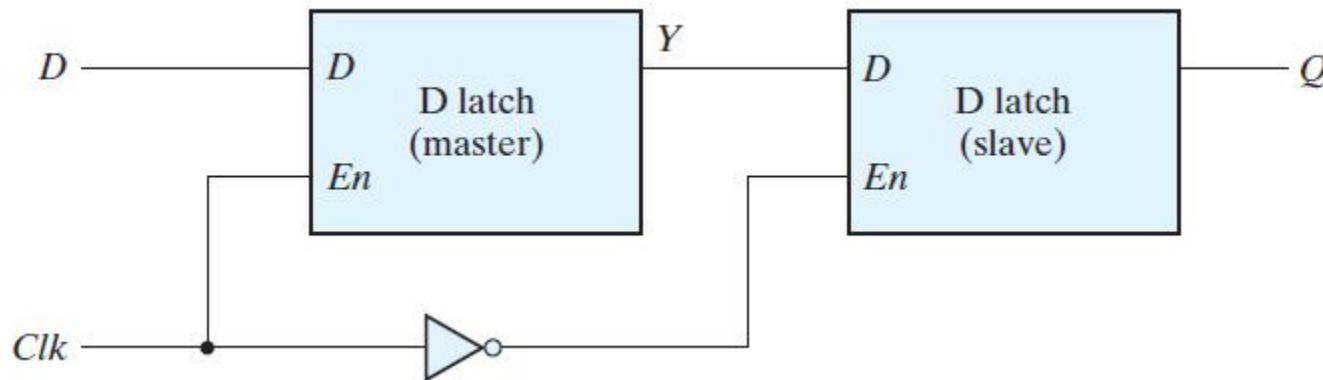
---

There are two ways that a latch can be modified to form a flip-flop

- One way is to employ two latches in a special configuration that isolates the output of the flip-flop and prevents it from being affected while the input to the flip-flop is changing.
- Another way is to produce a flip-flop that triggers only during a signal transition (from 0 to 1 or from 1 to 0) of the synchronizing signal (clock) and is disabled during the rest of the clock pulse.

# Edge-Triggered D Flip-Flop

- The construction of a D flip-flop with two D latches and an inverter:



- Also called **Master Slave D Flip-Flop**.
- The first latch is called the master and the second the slave.
- The circuit samples the D input and changes its output Q only at the negative edge of the synchronizing or controlling clock (designated as Clk ).

# Edge-Triggered D Flip-Flop

---

- When  $\text{clk}=1$  master is enabled and  $\text{clk}=0$  slave is enabled.
- Any change in the input changes the master output at  $Y$ , but cannot affect the slave output.
- Thus, a change in the output of the flip-flop can be triggered only by and during the transition of the clock from 1 to 0.
- The behavior of the master–slave flip-flop just described dictates that
  - the output may change only once,
  - a change in the output is triggered by the negative edge of the clock,
  - the change may occur only during the clock’s negative level.
- The value that is produced at the output of the flip-flop is the value that was stored in the master stage immediately before the negative edge occurred .

# Edge-Triggered D Flip-Flop

---

- It is also possible to design the circuit so that the flip-flop output changes on the positive edge of the clock.
- In sum, 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.
- 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
- minimum time called the **hold time** during which the  $D$  input must not change after the application of the positive transition of the clock.
- 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.

# Edge-Triggered D Flip-Flop

Parameter	Definition	Example Value	Illustration
<b>Setup Time</b>	Minimum time $D$ must be stable <b>before</b> the clock edge	5 ns	If clock $\uparrow$ at 20 ns $\rightarrow D$ must be stable from 15 ns onwards
<b>Hold Time</b>	Minimum time $D$ must remain stable <b>after</b> the clock edge	2 ns	If clock $\uparrow$ at 20 ns $\rightarrow D$ must remain unchanged until 22 ns
<b>Propagation Delay</b>	Time taken for output $Q$ to update after the clock edge	8 ns	If clock $\uparrow$ at 20 ns $\rightarrow$ output $Q$ updates at $\sim 28$ ns

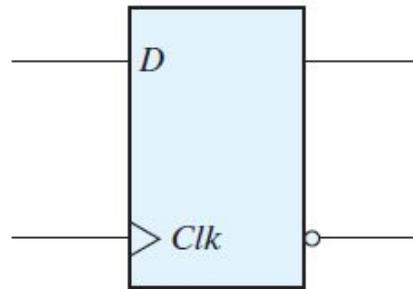
## Real-Life Analogy

Think of it like catching a photo with a camera:

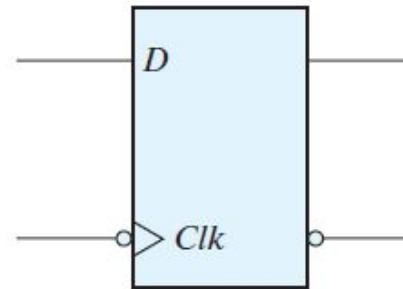
- **Setup time**  $\rightarrow$  Subject must pose *before* the shutter clicks.
- **Hold time**  $\rightarrow$  Subject must stay still *just after* the shutter clicks.
- **Propagation delay**  $\rightarrow$  Time it takes for the photo to actually appear on screen after clicking.

# Edge-Triggered D Flip-Flop

- Graphic symbol for edge-triggered D flip-flop:



(a) Positive-edge



(a) Negative-edge

- The dynamic indicator ( $>$ ) denotes the fact that the flip-flop responds to the edge transition of the clock.
- A bubble outside the block adjacent to the dynamic indicator designates a negative edge for triggering the circuit.
- The absence of a bubble designates a positive-edge response.

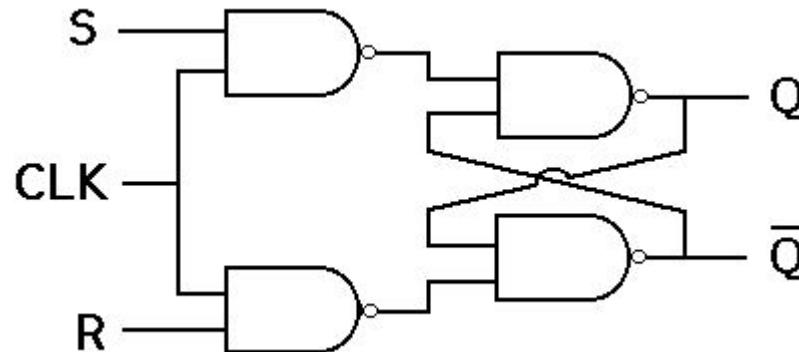
# Flip-Flops

---

Flip-Flops have following representation:

- Function table- output Q and  $Q'$
- Characteristic table- They define next state
- Characteristic equation-solve using K map- The logical properties of a flip-flop, as described in the characteristic table, can be expressed algebraically with a characteristic equation.
- Excitation table- predict the i/p based on current state and next state to build state diagram

# SR Flip-Flops Characteristic table and Characteristic equation



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

(b) Function table

Characteristic table

$S$	$R$	$Q_n$	$Q_{n+1}$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	-
1	1	1	-

Characteristic equation by simplifying using K maps  
$$Q_{n+1} = S + R'Q_n$$

# Excitation Table

---

An excitation table shows the minimum inputs that are necessary to generate a particular next state when the current state is known.

The excitation tables are used to determine the inputs of the flip-flop when the present state and the next state to which the flip-flop goes after the occurrence of the clock pulse are known.

# Excitation Table and State Diagram

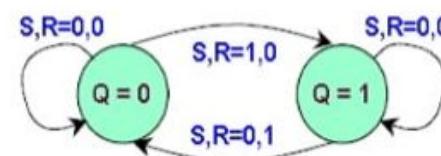
S	R	Present state $Q_n$	Next state $Q_{n+1}$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	X
1	1	1	X

Truth table of SR flip flop

} Invalid states

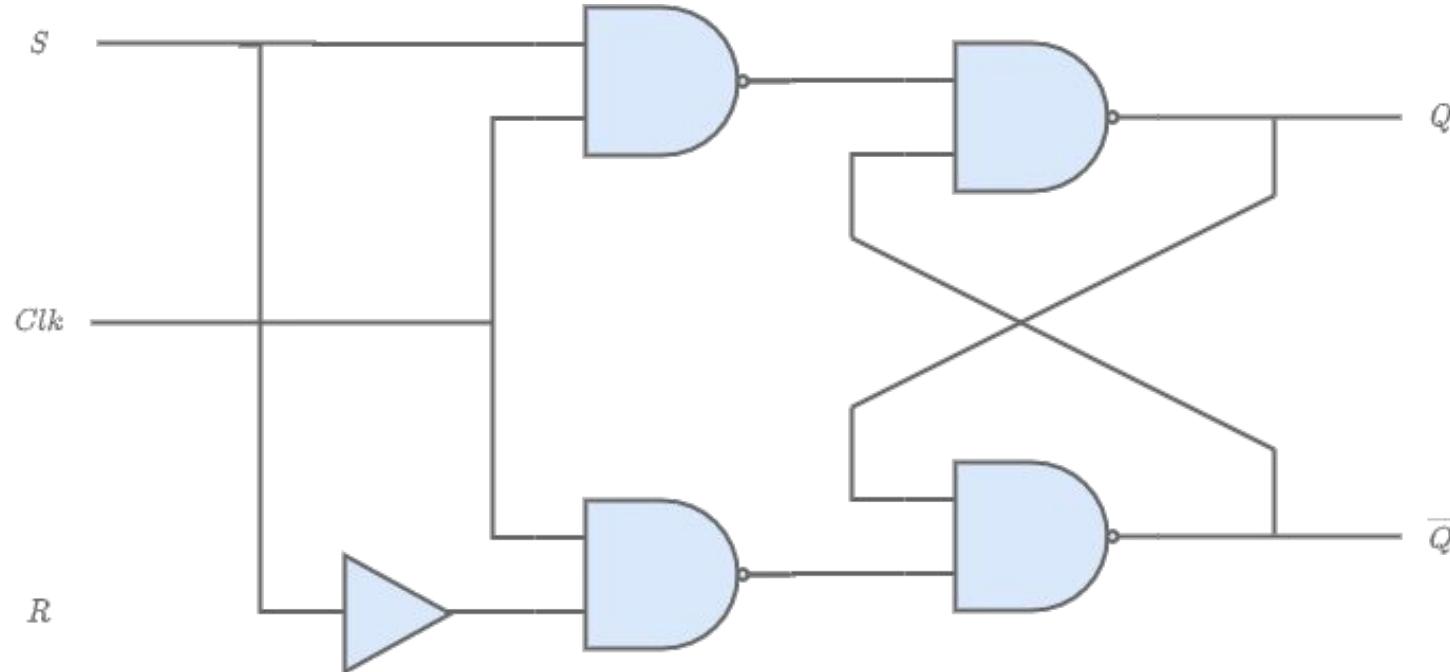
$Q_n$	$Q_{n+1}$	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

Excitation table of SR flip flop

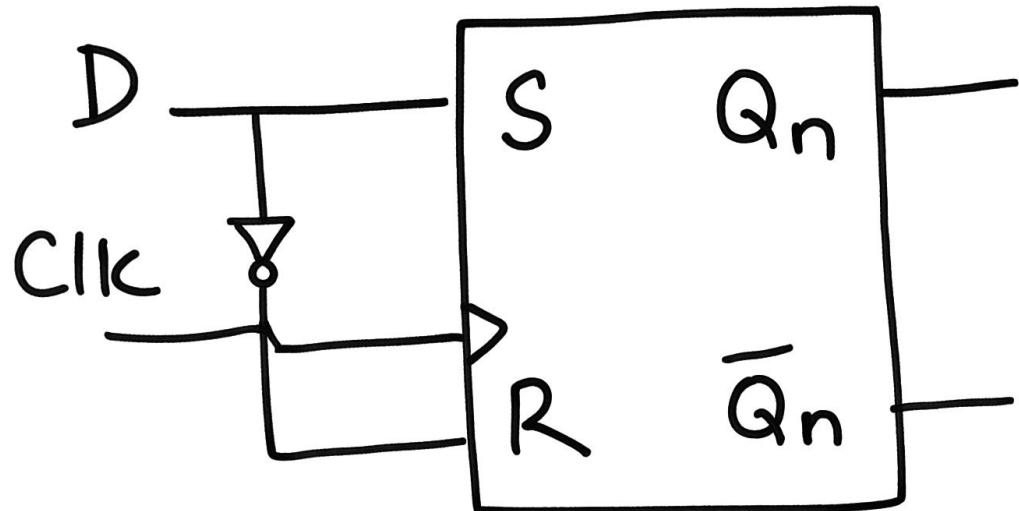


State Diagram

# D Flip-Flop



# D Flip-Flop



# D Flip-Flop- Characteristic Table and Characterstic Equation

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

(b) Function table

$D$	$Q_n$	$Q_{n+1}$
0	0	0
0	1	0
1	0	1
1	1	1

## **D Flip-Flop**

<b>D</b>	<b><math>Q(t + 1)</math></b>
0	0
1	1

Reset                      Set

Simplifying using K maps- characteristic equation is

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

# D Flip-Flop- excitation table

---

Q <sub>n</sub>	Q(n+1)	D
0	0	0
0	1	1
1	0	0
1	1	1

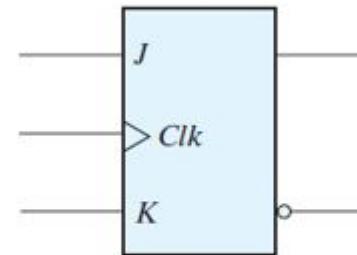
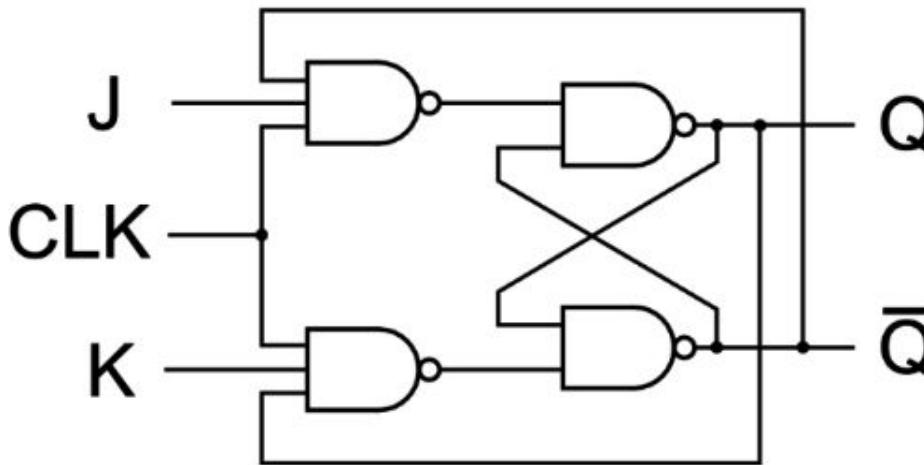
State Diagram

Excitation Table

# JK Flip-Flop

A **JK flip-flop** is required mainly to overcome the limitations of the **SR flip-flop** and to provide a more versatile storage element in sequential circuits.

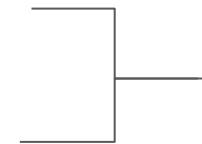
In an **SR flip-flop**, when **S=1** and **R=1** simultaneously, the output becomes indeterminate (invalid).



(b) Graphic symbol

# JK Flip-Flop-function table

Clk	J	K	Q
0	x	x	$Q_n$
1	0	0	$Q_n$
1	0	1	0
1	1	0	1
1	1	1	$\overline{Q_n}$



Memory

Toggle      race around condition

# JK Flip-Flop

---

**Characteristics table**

<b>Q(n)</b>	<b>J</b>	<b>K</b>	<b>Q(n+1)</b>
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

Solve using K map to  
get characteristic  
equation:

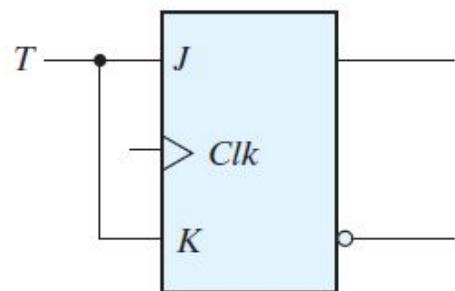
$$Q_{n+1} = Q_n K' + Q_n' J$$

# JK Flip-Flop

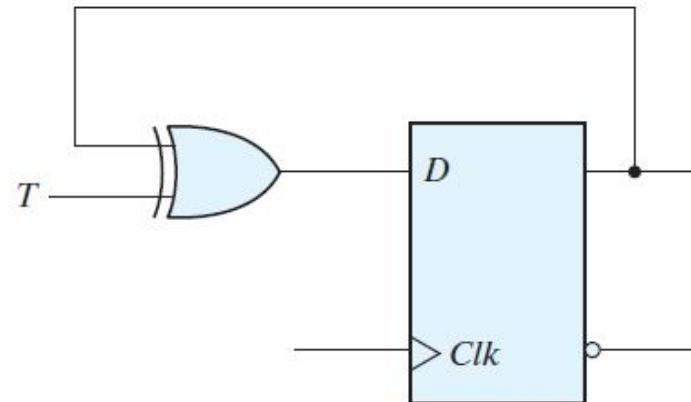
Excitation table

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

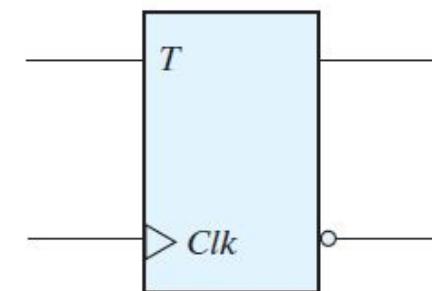
# T Flip-Flop (Toggle Flip-Flop)



(a) From JK flip-flop



(b) From D flip-flop



(c) Graphic symbol

# T Flip-Flop (Toggle Flip-Flop)

Truth table- if  $\text{clk}=0$  then previous state

CLK	T	$Q_{n+1}$
↑	0	$Q_n$
↑	1	$Q_n'$

# T Flip-Flop (Toggle Flip-Flop)

Characteristic table and equations:

T	Qn	Qn+1
0	0	0
0	1	1
1	0	1
1	1	0

Excitation table

Qn	Qn+1	T
0	0	0
0	1	1
1	0	1
1	1	0

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

# Flip-Flops

---

- JK flip-flop can function as:
- **SR flip-flop** ( $J = S$ ,  $K = R$ )
- **T flip-flop** ( $J=K=1 \rightarrow$  toggles)
- **D flip-flop** ( $J=D$ ,  $K=\neg D$ )
- Hence, it's a **universal flip-flop** that can emulate other types.

# Characteristics Table -summary

## Flip-Flop Characteristic Tables

### JK Flip-Flop

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

### D Flip-Flop

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

### T Flip-Flop

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

# Characteristics Equations-summary

The logical properties of a flip-flop, as described in the characteristic table, can be expressed algebraically with a characteristic equation. For the *D* flip-flop, we have the characteristic equation

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

which states that the next state of the output will be equal to the value of input *D* in the present state. The characteristic equation for the *JK* flip-flop can be derived from the characteristic table or from the circuit of Fig. 5.12. We obtain

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

where *Q* is the value of the flip-flop output prior to the application of a clock edge. The characteristic equation for the *T* flip-flop is obtained from the circuit of Fig. 5.13:

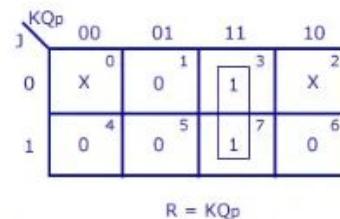
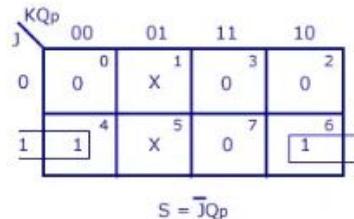
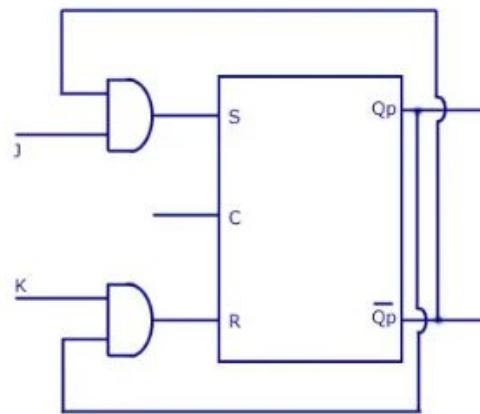
$$Q(t + 1) = T \oplus Q = TQ' + T'Q$$

# SR Flip-Flop to JK Flip-Flop

Conversion Table

J-K Inputs		Outputs		S-R Inputs	
J	K	$Q_p$	$Q_{p+1}$	S	R
0	0	0	0	0	X
0	0	1	1	X	0
0	1	0	0	0	X
0	1	1	0	0	1
1	0	0	1	1	0
1	0	1	1	X	0
1	1	0	1	1	0
1	1	1	0	0	1

Logic Diagram



<https://www.geeksforgeeks.org/digital-logic/flip-flop-types-their-conversion-and-applications/>

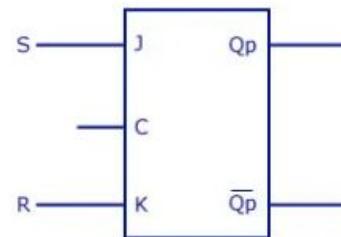
# JK Flip-Flop to SR Flip-Flop

J-K Flip Flop to S-R Flip Flop

Conversion Table

S-R Inputs		Outputs		J-K Inputs	
S	R	Q <sub>p</sub>	Q <sub>p+1</sub>	J	K
0	0	0	0	0	X
0	0	1	1	X	0
0	1	0	0	0	X
0	1	1	0	X	1
1	0	0	1	1	X
1	0	1	1	X	0
1	1	Invalid		Dont care	
1	1	Invalid		Dont care	

Logic Diagram



S	RQ <sub>p</sub>			
	00	01	11	10
0	0	X	X	0
1	4	5	7	6

J=S

S	RQ <sub>p</sub>			
	00	01	11	10
0	0	1	3	2
1	1	X	X	X

K-maps

K=R

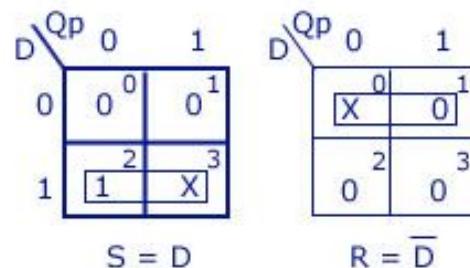
# SR Flip-Flop to D Flip-Flop

## S-R Flip Flop to D Flip Flop

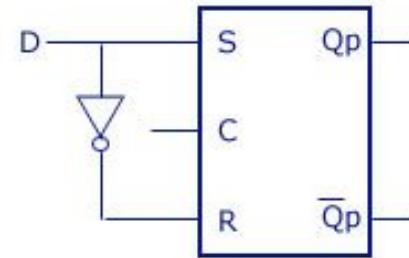
Conversion Table

D Input	Outputs		S-R Inputs	
	Q <sub>p</sub>	Q <sub>p+1</sub>	S	R
0	0	0	0	X
0	1	0	0	1
1	0	1	1	0
1	1	1	X	0

K-maps



Logic Diagram

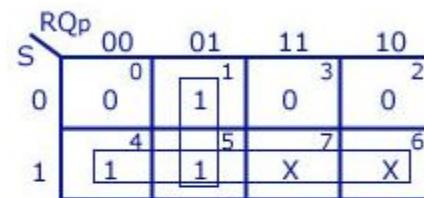


# D Flip-Flop to SR Flip-Flop

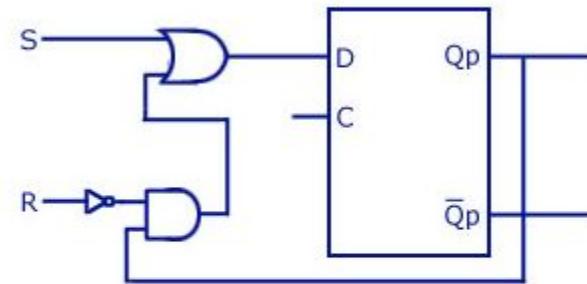
Conversion Table

S-R Inputs		Outputs		D Input
S	R	Q <sub>p</sub>	Q <sub>p+1</sub>	
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	0	0
1	0	0	1	1
1	0	1	1	1
1	1	Invalid	Dont care	
1	1	Invalid	Dont care	

K-map



Logic Diagram



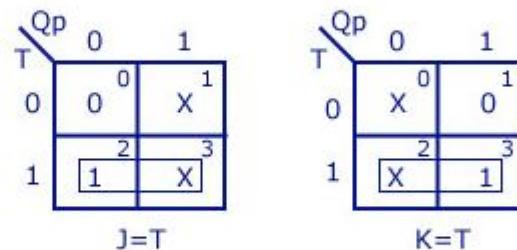
# JK Flip-Flop to T Flip-Flop

## J-K Flip Flop to T Flip Flop

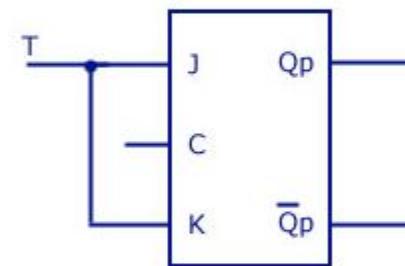
Conversion Table

T Input	Outputs		J-K Inputs	
	$Q_p$	$Q_{p+1}$	J	K
0	0	0	0	X
0	1	1	X	0
1	0	1	1	X
1	1	0	X	1

K-maps



Logic Diagram



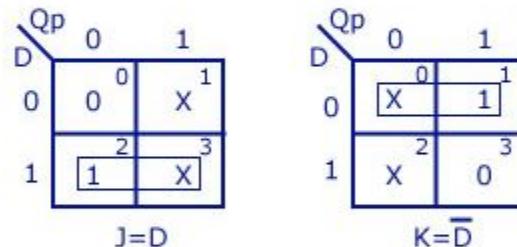
# JK Flip-Flop to D Flip-Flop

## J-K Flip Flop to D Flip Flop

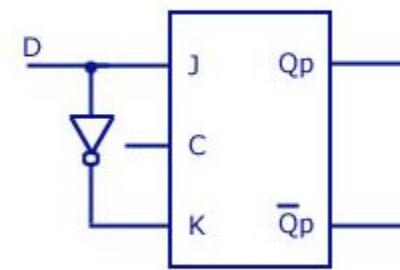
Conversion Table

D Input	Outputs		J-K Inputs	
	Q <sub>p</sub>	Q <sub>p+1</sub>	J	K
0	0	0	0	X
0	1	0	X	1
1	0	1	1	X
1	1	0	X	0

K-maps



Logic Diagram



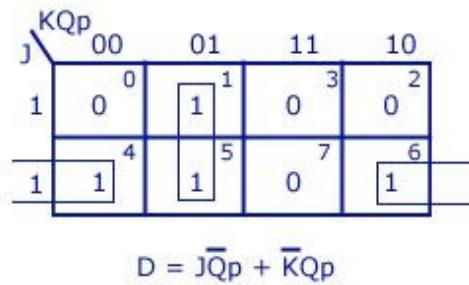
# D Flip-Flop to JK Flip-Flop

## D Flip Flop to J-K Flip Flop

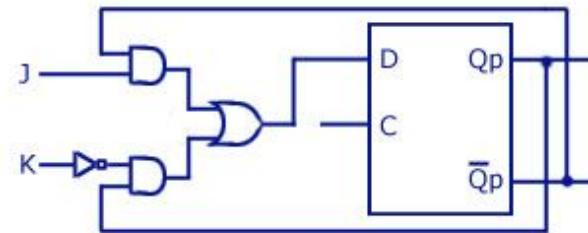
Conversion Table

J-K Input	K	Outputs		D Input
		Q <sub>p</sub>	Q <sub>p+1</sub>	
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	0	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	0	0

K-map



Logic Diagram



# Applications of Flipflops and Latches

---

- **Electronic Voting Machine**
- **Scenario:** Registering and storing votes securely.
- **How flip-flops help:** Flip-flops store the vote data bit-by-bit, allowing fast, reliable vote counting.

# Applications of Flipflops and Latches

Flip-Flop Type	Real-Time Example	How It Works in This Example	Why This Flip-Flop Is Suitable
JK Flip-Flop	Digital Watch Seconds Counter	Counts seconds by toggling state every clock pulse, progressing through time steps in binary.	JK flip-flop toggles easily on $J=K=1$ , ideal for counters.
D Flip-Flop	Computer Register (CPU Storage)	Holds the current data/instruction being processed, updated only on clock edges for accuracy.	D flip-flop stores data bits reliably, synchronized to clock.
SR Flip-Flop	Elevator Door Control System	Sets door to open (set) or closed (reset) state based on user inputs or safety sensors.	Simple set/reset functionality matches door open/close signals.
T Flip-Flop	LED Blinking Circuit	Toggles LED ON and OFF on each clock pulse to create blinking effect.	T flip-flop toggles output state each pulse, perfect for blinking LEDs.

- 1. Which of the following statements about a latch is TRUE?**
  - A) A latch is edge-triggered.
  - B) A latch is level-sensitive.
  - C) A latch can only store analog values.
  - D) A latch cannot hold its state without a clock signal.
  
- 2. In an SR latch constructed with NOR gates, what is the output when both S and R inputs are 0?**
  - A) Output is set ( $Q=1$ )
  - B) Output is reset ( $Q=0$ )
  - C) Output retains previous state
  - D) Output is indeterminate/invalid

**1. Which of the following statements about a latch is TRUE?**

- A) A latch is edge-triggered.
- B) A latch is level-sensitive.
- C) A latch can only store analog values.
- D) A latch cannot hold its state without a clock signal.

**Answer:** B

**2. In an SR latch constructed with NOR gates, what is the output when both S and R inputs are 0?**

- A) Output is set ( $Q=1$ )
- B) Output is reset ( $Q=0$ )
- C) Output retains previous state
- D) Output is indeterminate/invalid

**Answer:** C

3. In a JK flip-flop, when both J and K inputs are 1 and a clock pulse occurs, the output:

- A) Is set to 1
- B) Is reset to 0
- C) Toggles from its previous state
- D) Remains unchanged

4. A D flip-flop can be constructed using:

- A) Two SR latches connected in master-slave configuration.
- B) Only combinational logic
- C) Only NOR gates without feedback.
- D) A JK flip-flop with both J and K tied to the D input.

3. In a JK flip-flop, when both J and K inputs are 1 and a clock pulse occurs, the output:

- A) Is set to 1
- B) Is reset to 0
- C) Toggles from its previous state
- D) Remains unchanged

**Answer:** C

4. A D flip-flop can be constructed using:

- A) Two SR latches connected in master-slave configuration.
- B) Only combinational logic
- C) Only NOR gates without feedback.
- D) A JK flip-flop with both J and K tied to the D input.

**Answer:** A



**PES**  
UNIVERSITY

CELEBRATING 50 YEARS

**THANK YOU**

---

**Team DDCO  
Department of Computer Science and Engineering**



# DIGITAL DESIGN AND COMPUTER ORGANIZATION

## Analysis of Clocked Seq. Circuits

---

**Team DDCO**

**Department of Computer Science and Engineering**

# Introduction(section 5.5 T1)

---

- Analysis describes what a given circuit will do under certain operating conditions.
- The **behavior** of a clocked sequential circuit is determined from **the inputs, the outputs, and the state of its flip-flops**.
- The outputs and the next state are both a function of the inputs and the present state.
- The **analysis of a sequential circuit** consists of obtaining a **table or a diagram** for the **time sequence of inputs, outputs, and internal states**.
- It is also possible to write Boolean expressions that describe the behavior of the sequential circuit.

# Introduction

---

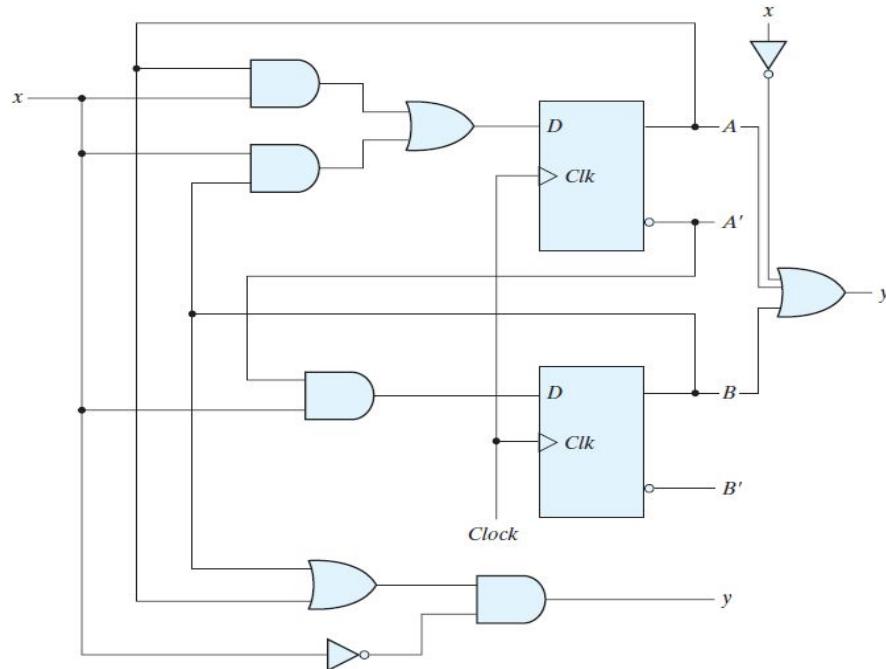
- These expressions must include the necessary time sequence, either directly or indirectly.
- A logic diagram is recognized as a clocked sequential circuit if it includes flip-flops with clock inputs.
- The flip-flops may be of any type, and the logic diagram may or may not include combinational logic gates.
- **Algebraic representation for specifying the next-state condition in terms of the present state and inputs**

# State Equations

---

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

# State Equations



Circuit acts as a 0-detector by asserting its output when a 0 is detected in a stream of 1s

Behavior of circuit:

Imagine you're monitoring a signal line that should be continuously 1 (high).

- As long as the signal is 1, the detector stays quiet.
- When a glitch (a single 0) happens, the detector alerts with  $y = 1$ .
- But if the line stays 0, it doesn't keep alerting — only the first error is flagged.

# State Equations

---

- It consists of two D flip-flops A and B, an input x and an output y .
- Since the D input of a flip-flop determines the value of the next state (i.e., the state reached after the clock transition), it is possible to write a set of state equations for the circuit:
  - $A(t + 1) = A(t)x(t) + B(t)x(t)$
  - $B(t + 1) = A'(t)x(t)$

# State Equations

---

- The Boolean expressions for the state equations can be derived directly from the gates that form the combinational circuit part of the sequential circuit, since the D values of the combinational circuit determine the next state.
- Similarly, the present-state value of the output can be expressed algebraically as
  - $y(t) = [A(t) + B(t)]x'(t)$
- By removing the symbol  $(t)$  for the present state, we obtain the output Boolean equation:
  - $y = (A + B)x'$

# State Table

---

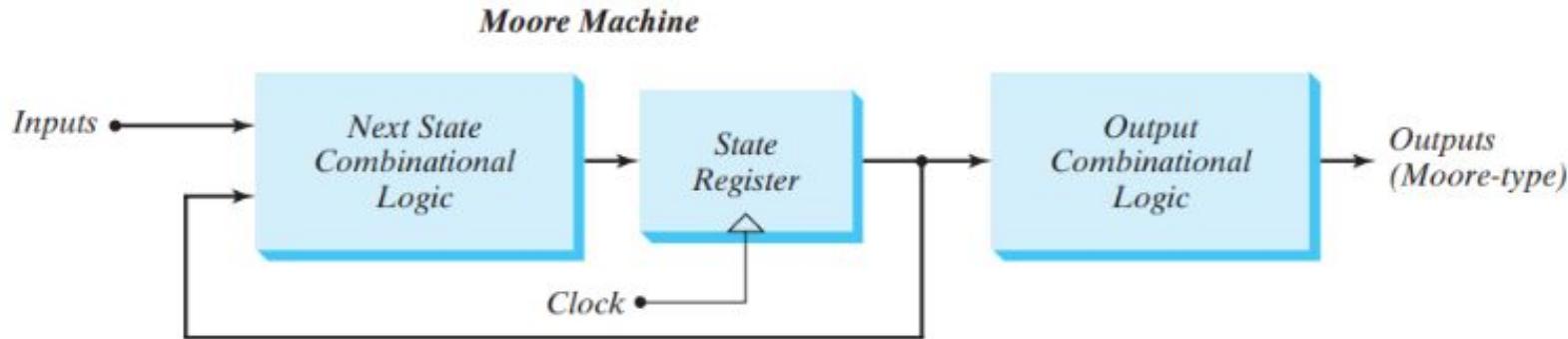
- The time sequence of **inputs, outputs, and flip-flop states** can be enumerated in a ***state table*** (sometimes called a ***transition table***).
- The state table for the previous circuit is shown in the table.
- The table consists of 4 sections: **present state, input, next state, and output** .
- The present-state section shows the states of flip-flops A and B at any given time t .
- The input section gives a value of x for each possible present state.
- The next-state section shows the states of the flip-flops one clock cycle later, at time t + 1.
- The output section gives the value of y at time t for each present state and input condition.

# State Table- Type 1

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

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

# Moore Models of Finite State Machines



- In the Moore model, the output is a function of only the present state
- Next state is function of present state and input

# State Table

---

- The state table of a sequential circuit with D -type flip-flops is obtained by the same procedure outlined in the previous example.
- In general, a sequential circuit with m flip-flops and n inputs needs  $2^{m+n}$  rows in the state table.
- The binary numbers from 0 through  $2^{m+n} - 1$  are listed under the present-state and input columns.
- The next-state section has m columns, one for each flip-flop.
- The binary values for the next state are derived directly from the state equations.
- The output section has as many columns as there are output variables.
- Its binary value is derived from the circuit or from the Boolean function in the same manner as in a truth table.

# State Table

---

- It is sometimes convenient to express the state table in a slightly different form having only three sections: present state, next state, and output.
- The input conditions are enumerated under the next-state and output sections.
- The new state table is as shown in this second form.
- For each present state, there are two possible next states and outputs, depending on the value of the input.

# State Table – Type 2

*Second Form of the State Table*

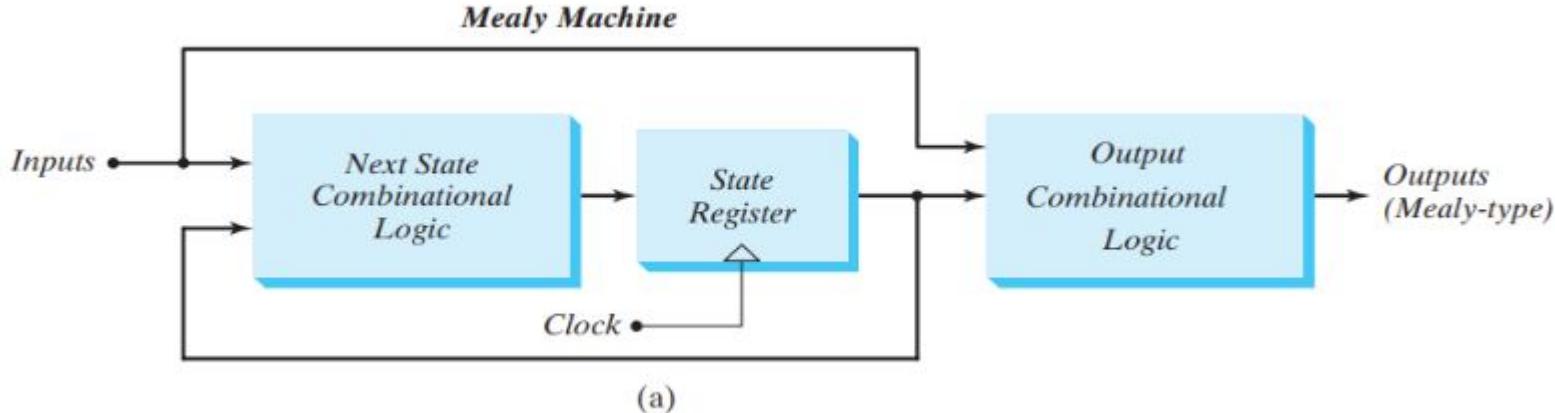
Present State		Next State				Output	
		$x = 0$		$x = 1$		$x = 0$	
$A$	$B$	$A$	$B$	$A$	$B$	$y$	$y$
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0

$$A(t+1) = Ax + Bx$$

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

$$y = Ax' + Bx'$$

# Mealy Models of Finite State Machines

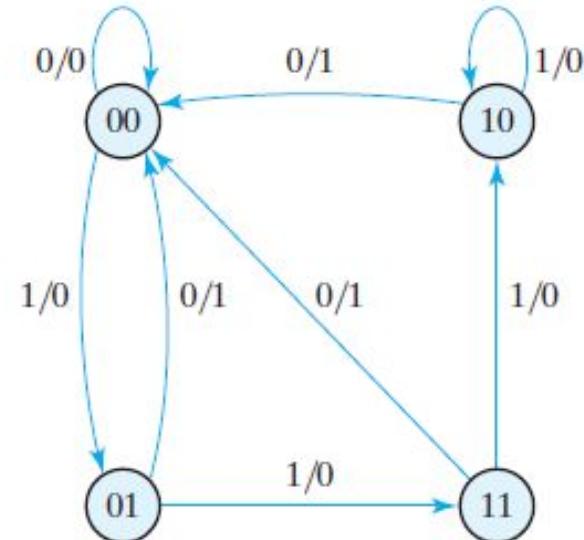


In the Mealy model, the output is a function of both the present state and the input.  
Next state depends on present state and inputs

# State Diagram

## Second Form of the State Table

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



Circuit diagram → Equations – State table → State diagram

# State Diagram

---

- The state diagram provides the same information as the state table and is obtained directly from table. **The information available in a state table can be represented graphically in the form of a state diagram**
- The binary number inside each circle identifies the state of the flip-flops.
- The directed lines are labelled with two binary numbers separated by a slash.
- The input value during the present state is labelled first, and the number after the slash gives the output during the present state with the given input.
- The steps presented in this example are summarized below:

Circuit diagram → Equations – State table → State diagram

**The state diagram gives a pictorial view of state transitions and is the form more suitable for human interpretation of the circuit's operation**

# Flip-Flop Input Equations

---

- The part of the combinational circuit that generates external outputs is described algebraically by a set of Boolean functions **called output equations**.
- The part of the circuit that generates the inputs to flip-flops is described algebraically by a set of Boolean functions called **flip-flop input equations** (or, sometimes, excitation equations).

$$D_A = Ax + Bx$$

$$D_B = A'x$$

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

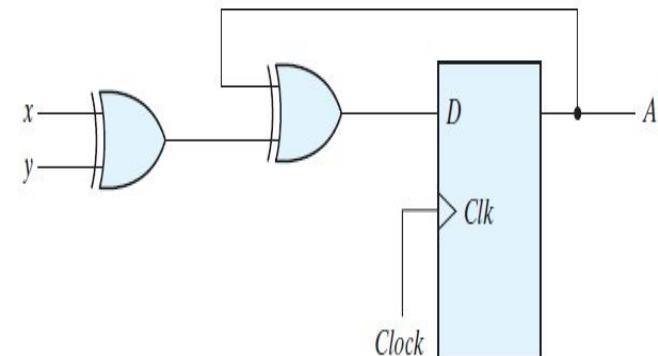
Note that the expression for the input equation for a D flip-flop is identical to the expression for the corresponding state equation. This is because of the characteristic equation that equates the next state to the value of the D input:  $Q(t + 1) = D_Q$ .

# Analysis with D Flip-Flops

- The procedure for analyzing a clocked sequential circuit with D flip-flops by means of a simple example. The circuit we want to analyze is described by the input equation:

$$D_A = A \oplus x \oplus y$$

- The  $D_A$  symbol implies a D flip-flop with output A.
- The x and y variables are the inputs to the circuit.
- No output equations are given, which implies that the output comes from the output of the flip-flop.



(a) Circuit diagram

# Analysis with D Flip-Flops

---

- The state table has one column for the present state of flip-flop  $A$ , two columns for the two inputs, and one column for the next state of  $A$ .
- The binary numbers under  $Axy$  are listed from 000 through 111 as shown in figure (b).
- The next-state values are obtained from the state equation

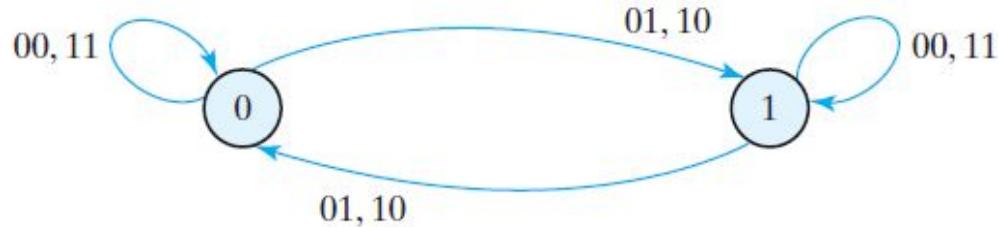
$$A(t + 1) = A \oplus x \oplus y$$

- The expression specifies an odd function and is equal to 1 when only one variable is 1 or when all three variables are 1.
- This is indicated in the column for the next state of  $A$  .

# Analysis with D Flip-Flops

Present state	Inputs		Next state
$A$	$x$	$y$	$A$
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

(b) State table



(c) State diagram

# Analysis with D Flip-Flops

---

- The circuit has one flip-flop and two states.
- The state diagram consists of two circles, one for each state as shown in figure (c).
- The present state and the output can be either 0 or 1, as indicated by the number inside the circles.(moore model)
- A slash on the directed lines is not needed, because there is no output from a combinational circuit.
- The two inputs can have four possible combinations for each state.
- Two input combinations during each state transition are separated by a comma to simplify the notation.

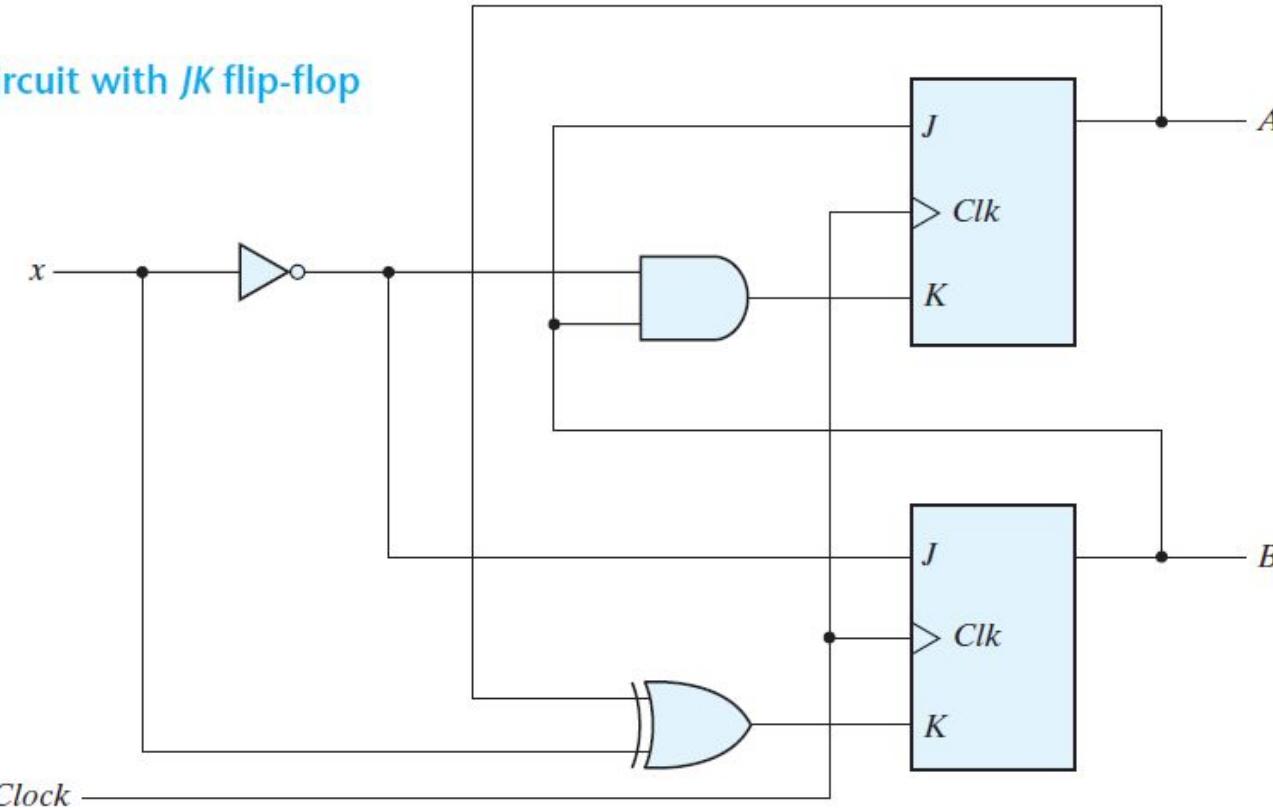
# Analysis of JK Flip-Flop

---

- When a flip-flop other than the  $D$  type is used, such as  $JK$  or  $T$ , it is necessary to refer to the corresponding characteristic table or characteristic equation to obtain the next-state values.
- The procedure is illustrated first by using the characteristic table and again by using the characteristic equation.
- The next-state values of a sequential circuit that uses  $JK$  – or  $T$  –type flip-flops can be derived as follows:
  - a. Determine the flip-flop input equations in terms of the present state and input variables.
  - b. List the binary values of each input equation.
  - c. Use the corresponding flip-flop characteristic table to determine the next-state values in the state table.

# Analysis of JK Flip-Flop

Sequential circuit with JK flip-flop



# Analysis of JK Flip-Flop

---

- As an example, consider the sequential circuit with two *JK* flip-flops *A* and *B* and one input *x*.
- The circuit has no outputs; therefore, the state table does not need an output column.
- The circuit can be specified by the flip-flop input equations

$$J_A = B \quad K_A = Bx'$$

$$J_B = x' \quad K_B = A'x + Ax' = A \oplus x$$

# Analysis of JK Flip-Flop

*State Table for Sequential Circuit with JK Flip-Flops*

Present State		Input <i>x</i>	Next State		Flip-Flop Inputs			
<i>A</i>	<i>B</i>		<i>A</i>	<i>B</i>	<i>J<sub>A</sub></i>	<i>K<sub>A</sub></i>	<i>J<sub>B</sub></i>	<i>K<sub>B</sub></i>
0	0	0	0	1	0	0	1	0
0	0	1	0	0	0	0	0	1
0	1	0	1	1	1	1	1	0
0	1	1	1	0	1	0	0	1
1	0	0	1	1	0	0	1	1
1	0	1	1	0	0	0	0	0
1	1	0	0	0	1	1	1	1
1	1	1	1	1	1	0	0	0

# Analysis of JK Flip-Flop

---

The next-state values can also be obtained by evaluating the state equations from the characteristic equation. This is done by using the following procedure:

1. Determine the flip-flop input equations in terms of the present state and input variables.
2. Substitute the input equations into the flip-flop characteristic equation to obtain the state equations.
3. Use the corresponding state equations to determine the next-state values in the state table.

# Analysis of JK Flip-Flop

---

The input equations for the two *JK* flip-flops of Fig. 5.18 were listed a couple of paragraphs ago. The characteristic equations for the flip-flops are obtained by substituting *A* or *B* for the name of the flip-flop, instead of *Q*:

$$A(t + 1) = JA' + K'A$$

$$B(t + 1) = JB' + K'B$$

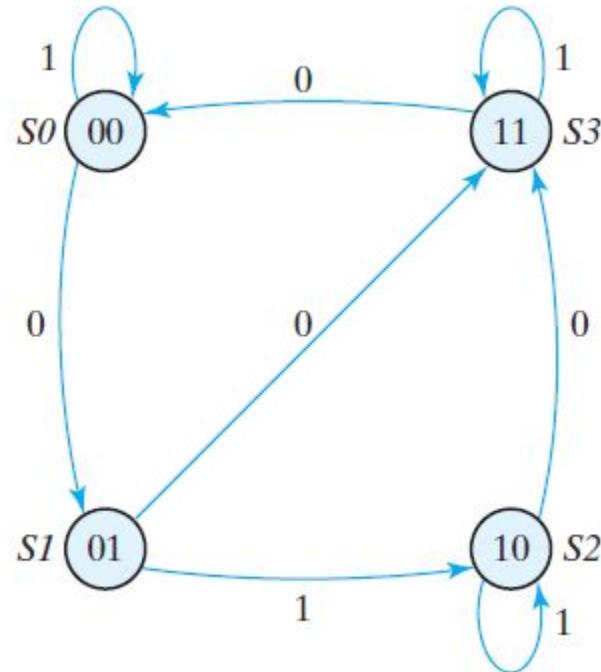
Substituting the values of  $J_A$  and  $K_A$  from the input equations, we obtain the state equation for *A*:

$$A(t + 1) = BA' + (Bx')' A = A'B + AB' + Ax$$

The state equation provides the bit values for the column headed “Next State” for *A* in the state table. Similarly, the state equation for flip-flop *B* can be derived from the characteristic equation by substituting the values of  $J_B$  and  $K_B$ :

$$B(t + 1) = x'B' + (A \oplus x)'B = B'x' + ABx + A'Bx'$$

# Analysis of JK Flip-Flop



State diagram of the circuit

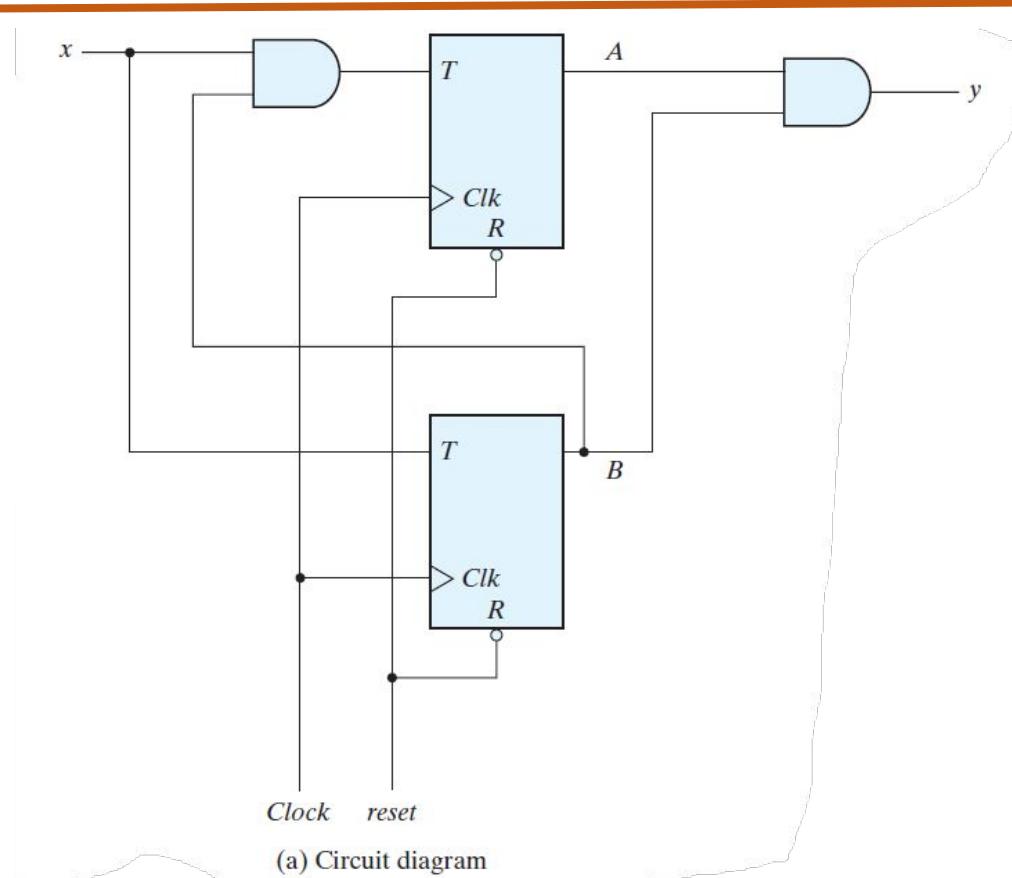
# Analysis with T Flip-Flops

---

- The analysis of a sequential circuit with T flip-flops follows the same procedure.
- The next-state values in the state table can be obtained by using either using the characteristic table or characteristic equation.

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

# Analysis with T Flip-Flops



# Analysis with T Flip-Flops

---

It has two flip-flops A and B, one input x, and one output y and can be described algebraically by two input equations and an output equation:

$$T_A = Bx$$

$$T_B = x$$

$$y = AB$$

# Analysis with T Flip-Flops

*State Table for Sequential Circuit with T Flip-Flops*

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

# Analysis with T Flip-Flops

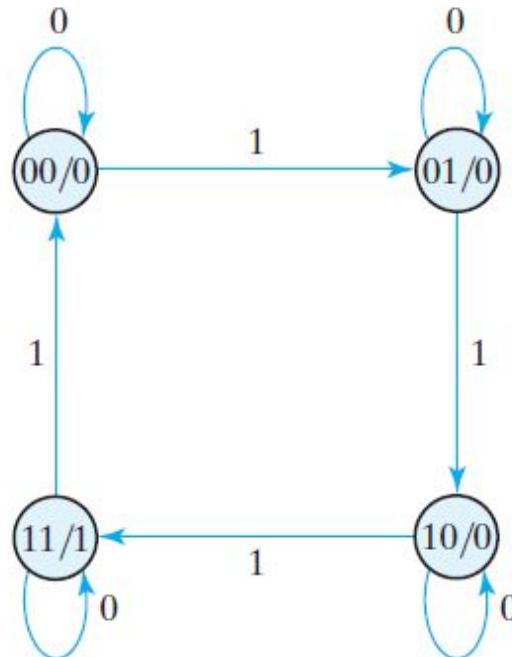
---

The values for y are obtained from the output equation. The values for the next state can be derived from the state equations by substituting TA and TB in the characteristic equations, yielding

$$A(t+1) = (Bx)'A + (Bx)A' = AB' + Ax' + A'Bx$$

$$B(t+1) = x \oplus B$$

# Analysis with T Flip-Flops



As long as input  $x$  is equal to 1, the circuit behaves as a binary counter with a sequence of states 00, 01, 10, 11, and back to 00.

When  $x = 0$ , the circuit remains in the same state. Output  $y$  is equal to 1 when the present state is 11.

Here, the output depends on the present state only and is independent of the input.

The two values inside each circle and separated by a slash are for the present state and output.

(b) State diagram

# Applications

## Elevator Controller (Multi-floor Building)

### •Description:

The elevator logic determines movement between floors, door operation, and priority handling.

### •How it relates to your concept:

- States: Floor positions + Door open/close status.
- Inputs: Floor request buttons, sensors.
- Flip-flops: Store current floor/door state;
- **state equations** decide next movement.

## Traffic Light Controller

### •Description:

A four-way traffic signal is controlled by a **finite state machine (FSM)** implemented using flip-flops.

### •How it relates to your concept:

- **States:** Red → Green → Yellow → Red.
- **Flip-flops:** Store the current light state.
- **State table & equations:** Determine which light turns on next based on current state and timer pulse.

# Applications

## Digital Lock System

### Description:

Unlocks only when a correct sequence of digits/buttons is entered.

### How it relates to your concept:

**States:** Each entered digit represents a state transition.

**Inputs:** Keypad inputs.

**Flip-flops:** Store entered sequence progress.

**State equations:** Verify input sequence and transition to “Unlock” state if correct.

1. The present state of a D flip-flop is 1. If the input equation is  $D=A \oplus x \oplus y$  where A is the current flip-flop output,  $x=1$  and  $y=0$ , what will be the next state after the clock edge?  
**A) 0**  
**B) 1**  
**C) Cannot be determined**  
**D) Remains same**
  
2. A sequential circuit uses two D flip-flops A and B, and one input x. The next state equations are:  
 $A(t+1)=A \cdot x + B \cdot x$   
 $B(t+1)=A' \cdot x$   
If present state is  $A=1, B=0, x=1$ , what is the next state?  
**A) (1,0)**  
**B) (1,1)**  
**C) (0,1)**  
**D) (0,0)**

1. The present state of a D flip-flop is 1. If the input equation is  $D=A \oplus x \oplus yD$  where A is the current flip-flop output,  $x=1$ , and  $y=0$ , what will be the next state after the clock edge?

- A) 0
- B) 1
- C) Cannot be determined
- D) Remains same

**Answer:** A —  $D=1 \oplus 1 \oplus 0$ , so next state is 0.

2. A sequential circuit uses two D flip-flops A and B, and one input x. The next state equations are:

$$\begin{aligned}A(t+1) &= A \cdot x + B \cdot x \\B(t+1) &= A' \cdot x\end{aligned}$$

If present state is  $A=1, B=0, x=1$ , what is the next state?

- A) (1,0)
- B) (1,1)
- C) (0,1)
- D) (0,0)

**Answer:** A — Direct substitution into the state equations.



**PES**  
UNIVERSITY

CELEBRATING 50 YEARS

**THANK YOU**

---

**Team DDCO  
Department of Computer Science and Engineering**



# DIGITAL DESIGN AND COMPUTER ORGANIZATION

## Mealy and Moore Models of FSMs

---

**Team DDCO**

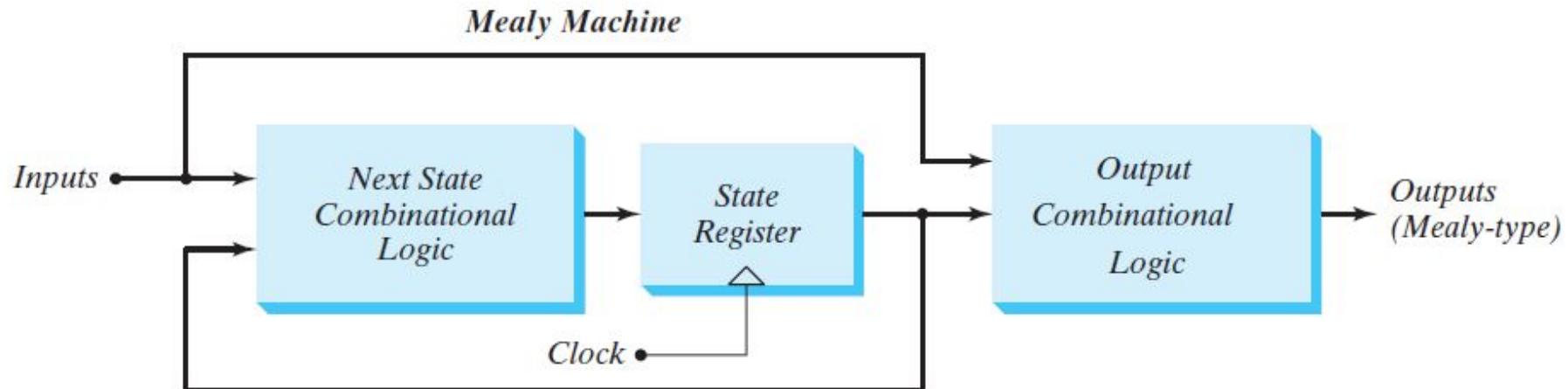
**Department of Computer Science and Engineering**

# Introduction(T1-section 5.5)

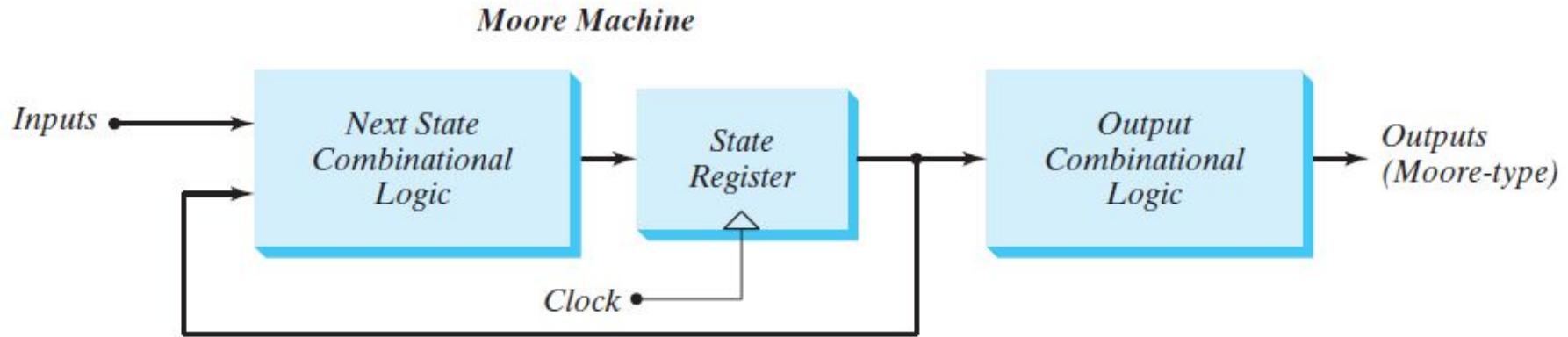
---

- 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.

# Block Diagrams



# Block Diagrams



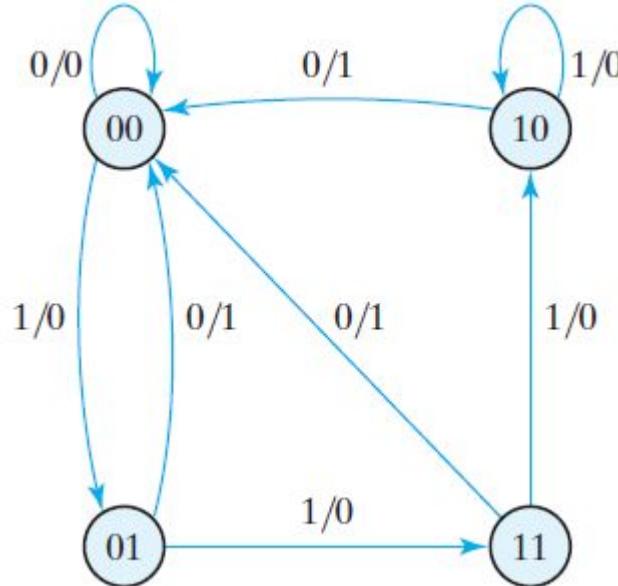
# Introduction

---

- In the Mealy model, the output is a function of both the present state and the input.
- 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.
- The Mealy model of a sequential circuit is referred to as a Mealy FSM or Mealy machine.
- The Moore model is referred to as a Moore FSM or Moore machine.

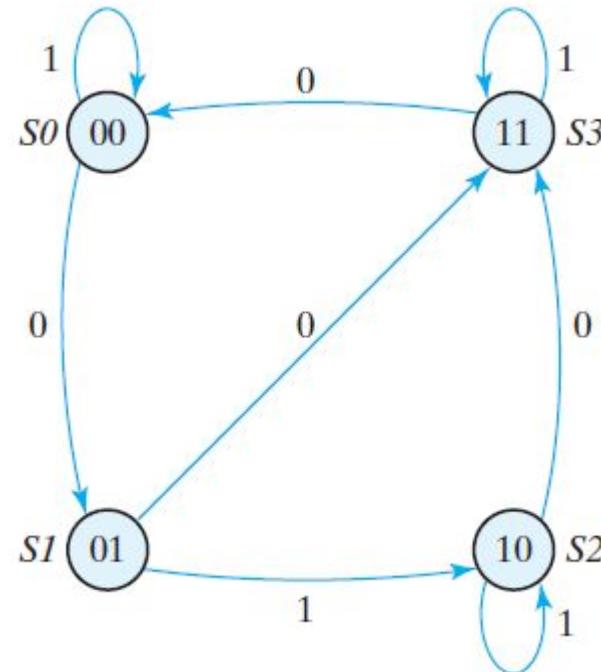
# Mealy Machine

- Output  $y$  is a function of both input  $x$  and the present state of A and B .
- The corresponding state diagram shows both the input and output values, separated by a slash along the directed lines between the states.



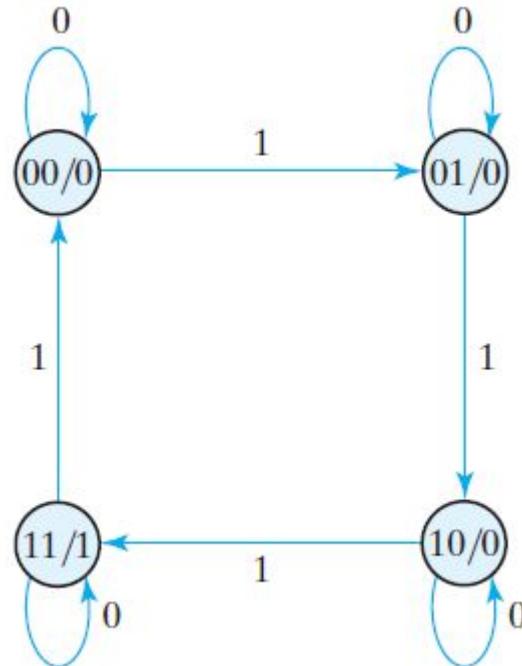
# Moore Machine

- Here, the output is a function of the present state only.
- The corresponding state diagram has only inputs marked along the directed lines.
- The outputs are the flip-flop states marked inside the circles.



# Moore Machine

- The output depends only on flip-flop values, and that makes it a function of the present state only.
- The input value in the state diagram is labeled along the directed line, but the output value is indicated inside the circle together with the present state.



# Points to Remember

---

- In a Moore model, the outputs of the sequential circuit are synchronized with the clock, because they depend only on flip-flop outputs that are synchronized with the clock. In a Mealy model, the outputs may change if the inputs change during the clock cycle.
- In order to synchronize a Mealy-type circuit, the inputs of the sequential circuit must be synchronized with the clock and the outputs must be sampled immediately before the clock edge.
- The inputs are changed at the inactive edge of the clock to ensure that the inputs to the flip-flops stabilize before the active edge of the clock occurs.
- Thus, the output of the Mealy machine is the value that is present immediately before the active edge of the clock.

# Applications

---

## Mealy Model

### Vending Machine with Coin Detection

- **How it works:** As soon as the correct coin is inserted (input) while the machine is in the “waiting” state, it immediately unlocks the product selection buttons (output).
- **Why Mealy:** Output depends on both **current state** (“waiting”) and **instant input** (coin detected) → faster interaction.

# Applications

---

## Moore Model

- **Example: Washing Machine Cycle**
- **How it works:** Once the washer enters the “spin” state, it spins for a fixed duration regardless of what buttons are pressed during that time.
- **Why Moore:** Output (motor spinning) depends only on **current state**, ensuring stability and avoiding mid-cycle changes

### Moore:

- Needs separate states for “output=0” and “output=1.”
- More states.

### ● Mealy:

- Can stay in one state and just make the output depend on  $x$ .
- Fewer states.

Consider a Mealy machine and a Moore machine implemented for the same problem. Which of the following is true?

- A) Moore machine always has fewer states.
- B) Mealy machine may have fewer states.
- C) Both have the same number of states always.
- D) Moore machine always produces output one clock earlier than Mealy.

Consider a Mealy machine and a Moore machine implemented for the same problem. Which of the following is true?

- A) Moore machine always has fewer states.
- B) Mealy machine may have fewer states.
- C) Both have the same number of states always.
- D) Moore machine always produces output one clock earlier than Mealy.

**Answer:** B — Mealy can have fewer states due to immediate output change.

In a Mealy machine, the output depends on:

- A) Present state only
- B) Present state and next state
- C) Present state and input
- D) Input only

The main difference between a Moore and a Mealy machine is that:

- A) Moore's output depends only on input.
- B) Mealy's output depends only on state.
- C) Moore's output depends only on state, while Mealy's output depends on state and input.
- D) Mealy's output is always faster than Moore's output, regardless of design.

In a Mealy machine, the output depends on:

- A) Present state only
- B) Present state and next state
- C) Present state and input
- D) Input only

**Answer: C** — Present state and input.

The main difference between a Moore and a Mealy machine is that:

- A) Moore's output depends only on input.
- B) Mealy's output depends only on state.
- C) Moore's output depends only on state, while Mealy's output depends on state and input.
- D) Mealy's output is always faster than Moore's output, regardless of design.

**Answer: C** — Moore → State only; Mealy → State + Input



**PES**  
UNIVERSITY

CELEBRATING 50 YEARS

**THANK YOU**

---

**Team DDCO  
Department of Computer Science and Engineering**



# DIGITAL DESIGN AND COMPUTER ORGANIZATION

**State Reduction and Assignment  
Design**

---

**Team DDCO**

**Department of Computer Science and Engineering**

# State Reduction(T1- section 5.7)

---

- The **analysis** of sequential circuits starts from a circuit diagram and culminates in a state table or diagram.
- The **design** (synthesis) of a sequential circuit starts from a set of specifications and culminates in a logic diagram. (section 5.8)
- Two sequential circuits may exhibit the same input–output behavior, but have a different number of internal states in their state diagram.

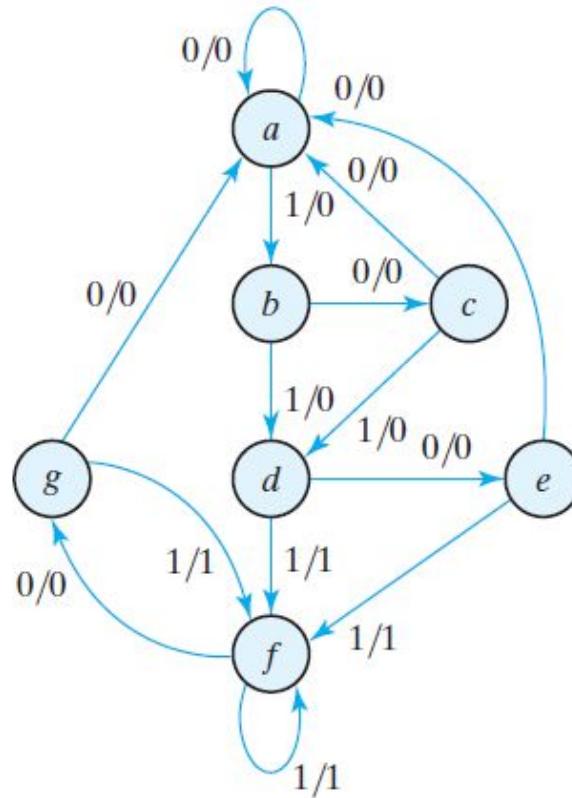
# State Reduction(T1- section 5.7)

---

- Properties of sequential circuits that may **simplify a design by reducing the number of gates and flip-flops it uses.** In general, reducing the number of flip flops reduces the **cost** of a circuit.
- **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.
- Each state can be encoded with binary numbers for 2 flipflops
  - A=00
  - B=01
  - C=10
  - D=11

# State Reduction

State diagram



As an example, consider the input sequence 01010110100 starting from the initial state a .

# State Reduction

---

- 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.

state	$a$	$a$	$b$	$c$	$d$	$e$	$f$	$f$	$g$	$f$	$g$	$a$
input	0	1	0	1	0	1	1	0	1	0	0	
output	0	0	0	0	0	1	1	0	1	0	0	

# State Reduction

---

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 Reduction

**Table 5.6**  
*State Table*

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

# State Reduction

- 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.
- states f and d are equivalent, and state f can be removed and replaced by d .

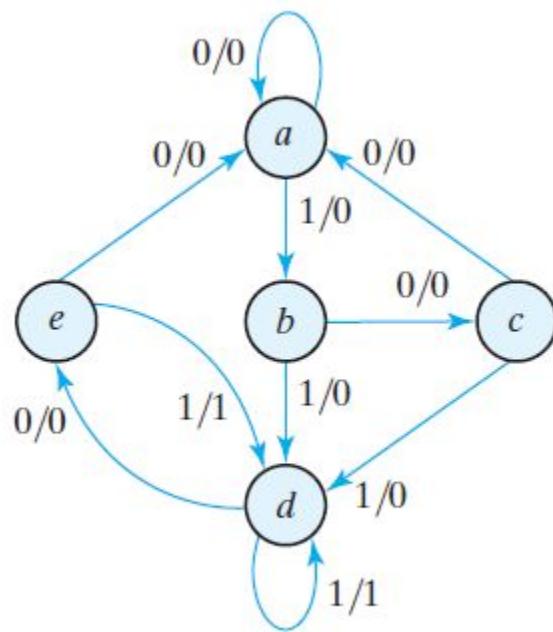
**Table 5.7**  
*Reducing the State Table*

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	e	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

# State Reduction



Reduced state diagram

# State Reduction

---

- The sequential circuit of this example was reduced from seven to five states.
- In general, reducing the number of states in a state table may result in a circuit with **less equipment**.
- 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**.
- In actual practice designers may skip this step because target devices are rich in resources.

# State Assignment

---

- 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.
- Example: 7 states are there
  - 000 to 111
  - one state can be unused.
  - In truth table mentioned as don't care condition

# State Assignment

## *Three Possible Binary State Assignments*

<b>State</b>	<b>Assignment 1, Binary</b>	<b>Assignment 2, Gray Code</b>	<b>Assignment 3, One-Hot</b>
<i>a</i>	000	000	00001
<i>b</i>	001	001	00010
<i>c</i>	010	011	00100
<i>d</i>	011	010	01000
<i>e</i>	100	110	10000

- Binary=  $2^n=m$ -The simplest way to code five states is to use the first five integers in binary counting order
- Gray code = adjacent bit differ by one bit
- On hot encoding-t. At any given time, only one bit is equal to 1 while all others are kept at 0. This type of assignment uses **one flipflop per state**.

# State Assignment

---

- One-hot encoding usually leads to simpler decoding logic for the next state and output
- One-hot machines can be **faster** than machines with sequential binary encoding, and the silicon area required by the **extra flip-flops** can be offset by the area saved by using simpler decoding logic.
- This **trade-off** is not guaranteed, so it must be evaluated for a given design.

# State Assignment

*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

*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

Also referred as transition table

A different assignment will result in a state table with different binary values for the states.

# Design Procedure(T1 –section 5.8)

---

- Design procedures or methodologies **specify hardware** that will implement a desired behavior
- The sequential building block used by synthesis tools is the D flip-flop. Together with additional logic, it can implement the behavior of JK and T flip-flops.
- In fact, designers **generally do not concern themselves with the type of flip-flop; rather, their focus is on correctly describing the sequential functionality that is to be implemented by the synthesis tool**
- In contrast to a combinational circuit, which is fully specified by a truth table, a sequential circuit requires a state table for its specification.

# Design Procedure(T1 –section 5.8)

---

- A synchronous sequential circuit is made up of flip-flops and combinational gates.
- The number of flip-flops is determined from the number of states needed in the circuit and the choice of state assignment codes.
- The combinational circuit is derived from the state table

# Design Procedure

---

The procedure for designing synchronous sequential circuits can be summarized by a list of recommended steps:

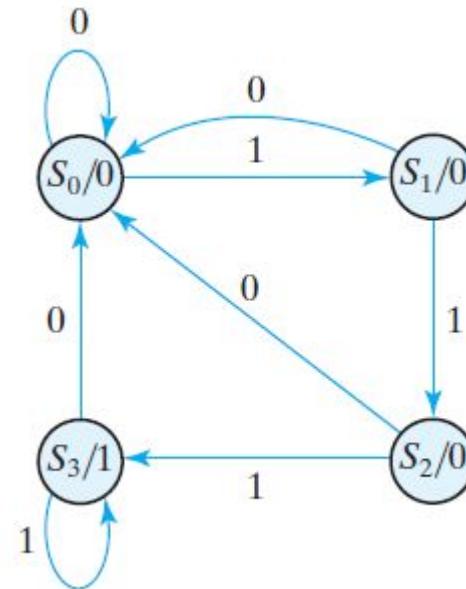
1. From the word description and specifications of the desired operation, derive a state diagram for the circuit.
2. Reduce the number of states if necessary.
3. Assign binary values to the states.
4. Obtain the binary-coded state table.
5. Choose the type of flip-flops to be used.
6. Derive the simplified flip-flop input equations and output equations.
7. Draw the logic diagram.

# Design Procedure

Suppose we wish to design a circuit that detects a sequence of **three or more consecutive 1's** in a string of bits coming through an input line (i.e., the input is a serial bit stream).

**S<sub>0</sub>= reset state**

**Below shown is moore model**



# Design Procedure

---

If the input is 0, the circuit stays in S0, but if the input is 1, it goes to state S1 to indicate that a 1 was detected. If the next input is 1, the change is to state S2 to indicate the arrival of two consecutive 1's, but if the input is 0, the state goes back to S0. The third consecutive 1 sends the circuit to state S3. If more 1's are detected, the circuit stays in S3. Any 0 input sends the circuit back to S0. In this way, the circuit stays in S3 as long as there are three or more consecutive 1's received.

**This is a Moore model sequential circuit, since the output is 1 when the circuit is in state S3 and is 0 otherwise.**

# Synthesis Using D Flip-Flops

---

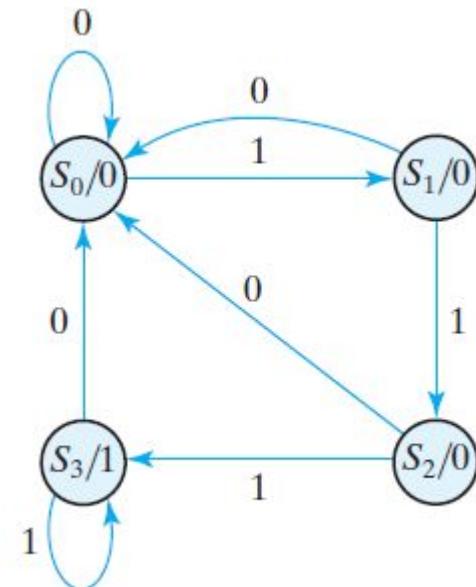
- We choose two D flip-flops to represent the four states, and we label their outputs A and B. There is one input x and one output y.
- The characteristic equation of the D flip-flop is  $Q(t+1)=DQ$ , which means that the next-state values in the state table specify the D input condition for the flip-flop.

**State Table for Sequence Detector. – BINARY ENCODED STATE TABLE**  
**S0=00, S1=01, S2=10 s3=11**

# Synthesis Using D Flip-Flops

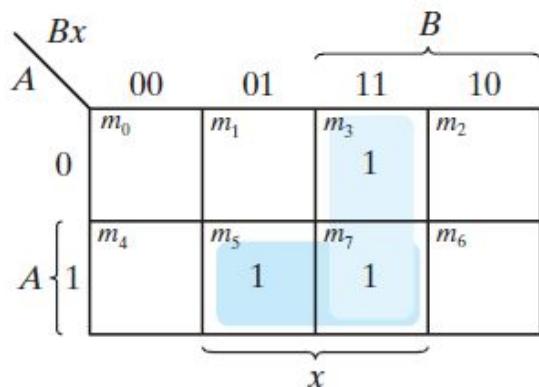
*State Table for Sequence Detector*

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

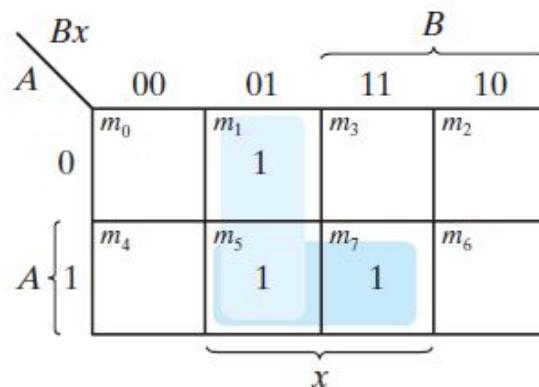


# Synthesis Using D Flip-Flops

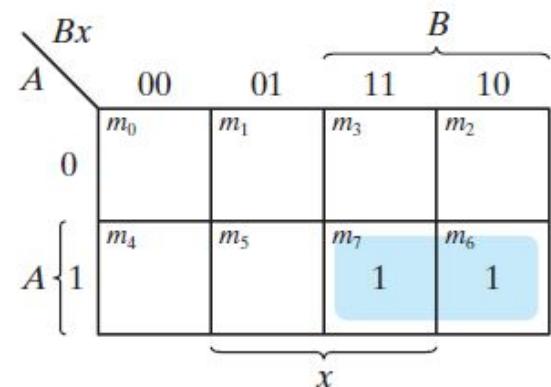
## K-Maps for sequence detector



$$D_A = Ax + Bx$$



$$D_B = Ax + B'x$$



$$y = AB$$

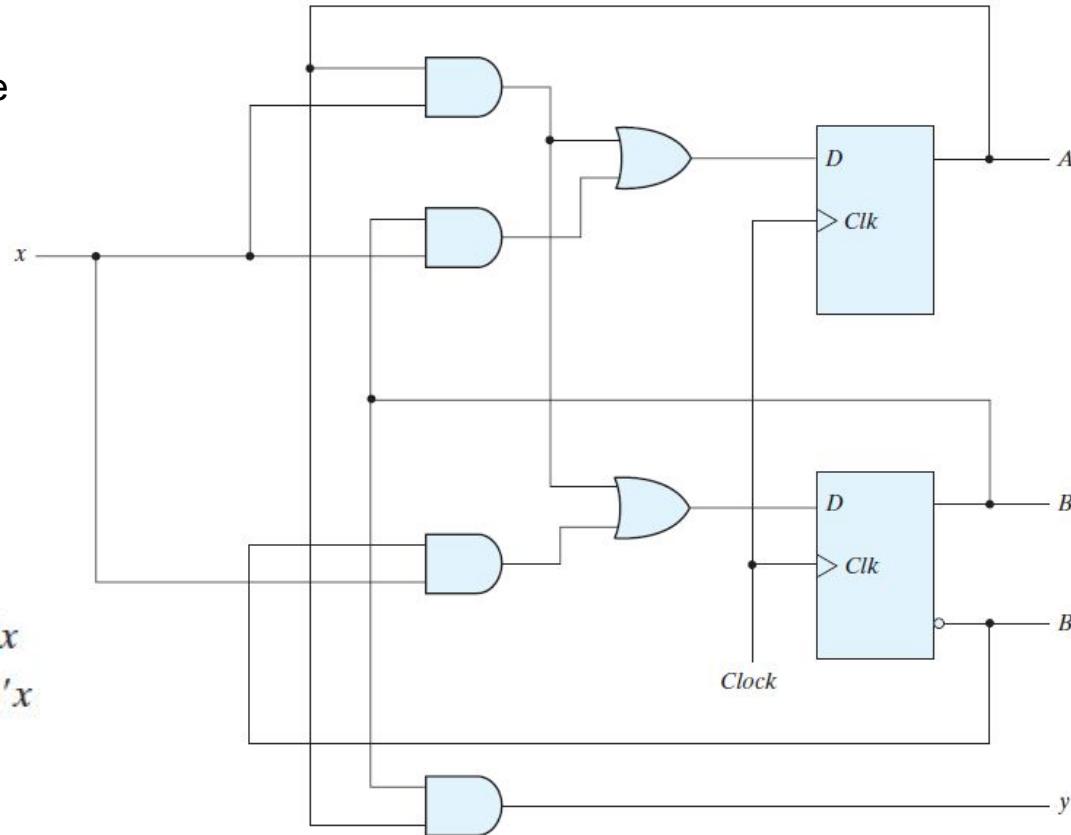
$$A(t+1) = D_A(A, B, x) = \Sigma(3, 5, 7)$$

$$B(t+1) = D_B(A, B, x) = \Sigma(1, 5, 7)$$

$$y(A, B, x) = \Sigma(6, 7)$$

# Synthesis Using D Flip-Flops

Logic diagram of a  
Moore-type sequence  
detector



# Excitation Table

- When D -type flip-flops are employed, the input equations are obtained directly from the next state. This is not the case for the JK and T types of flip-flops.
- In order to determine the input equations for these flip-flops, it is necessary to derive a functional relationship between the state table and the input equations.

*Flip-Flop 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

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

(b) T Flip-Flop

# Synthesis Using JK Flip-Flops

## *State Table and JK Flip-Flop Inputs*

Present State		Input <i>x</i>	Next State		Flip-Flop Inputs			
<i>A</i>	<i>B</i>		<i>A</i>	<i>B</i>	<i>J<sub>A</sub></i>	<i>K<sub>A</sub></i>	<i>J<sub>B</sub></i>	<i>K<sub>B</sub></i>
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

<i>Q(t)</i>	<i>Q(t = 1)</i>	<i>J</i>	<i>K</i>
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

(a) JK Flip-Flop

Excitation Table

# Synthesis Using JK Flip-Flops

$A \backslash Bx$	00	01	11	10
0	$m_0$	$m_1$	$m_3$	$m_2$
$A \backslash 1$	$m_4$	$m_5$	$m_7$	$m_6$
	$\underbrace{\hspace{4em}}_X$			
	$\overbrace{\hspace{4em}}^B$			

$J_A = Bx'$

$A \backslash Bx$	00	01	11	10
0	$m_0$	$m_1$	$m_3$	$m_2$
$A \backslash 1$	$m_4$	$m_5$	$m_7$	$m_6$
	$\underbrace{\hspace{4em}}_X$			
	$\overbrace{\hspace{4em}}^B$			

$K_A = Bx$

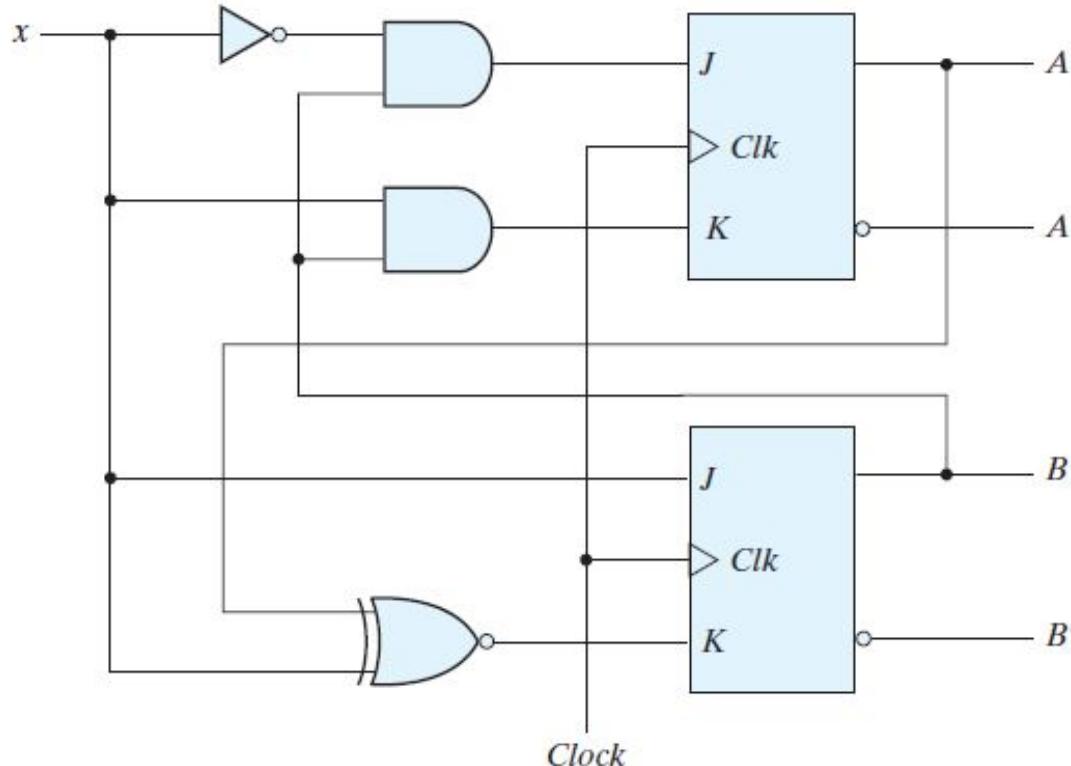
$A \backslash Bx$	00	01	11	10
0	$m_0$	$m_1$	$m_3$	$m_2$
$A \backslash 1$	$m_4$	$m_5$	$m_2$	$m_6$
	$\underbrace{\hspace{4em}}_X$			
	$\overbrace{\hspace{4em}}^B$			

$J_B = x$

$A \backslash Bx$	00	01	11	10
0	$m_0$	$m_1$	$m_3$	$m_2$
$A \backslash 1$	$m_4$	$m_5$	$m_7$	$m_6$
	$\underbrace{\hspace{4em}}_X$			
	$\overbrace{\hspace{4em}}^B$			

$K_B = (A \oplus x)'$

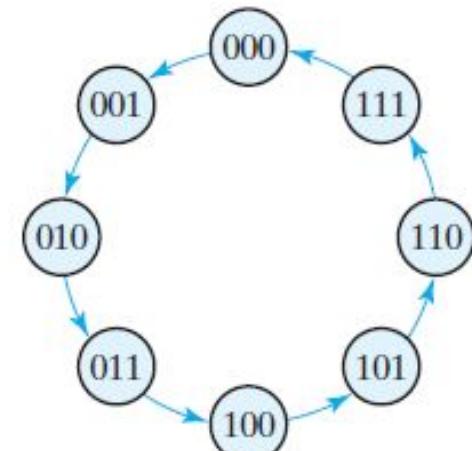
# Synthesis Using JK Flip-Flops



Logic diagram for sequential circuit with JK flip-flops

# Synthesis Using T Flip-Flops

- The procedure for synthesizing circuits using T flip-flops will be demonstrated by designing a binary counter.
- An n-bit binary counter consists of n flip-flops that can count in binary from 0 to  $2^n - 1$*
- the state diagram of a counter does not have to show input and output values along the directed lines.
- The only input to the circuit is the clock, and the outputs are specified by the present state of the flip-flops.
- The next state of a counter depends entirely on its present state, and the state transition occurs every time the clock goes through a transition.



State diagram of three-bit binary counter.

# Synthesis Using T Flip-Flops

*State Table for Three-Bit Counter*

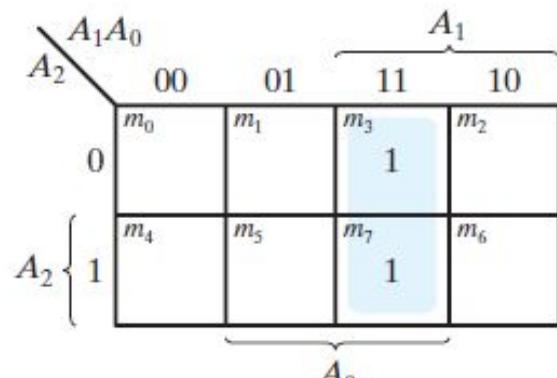
Present State			Next State			Flip-Flop Inputs		
$A_2$	$A_1$	$A_0$	$A_2$	$A_1$	$A_0$	$T_{A2}$	$T_{A1}$	$T_{A0}$
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	0	1
1	1	1	0	0	0	1	1	1

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

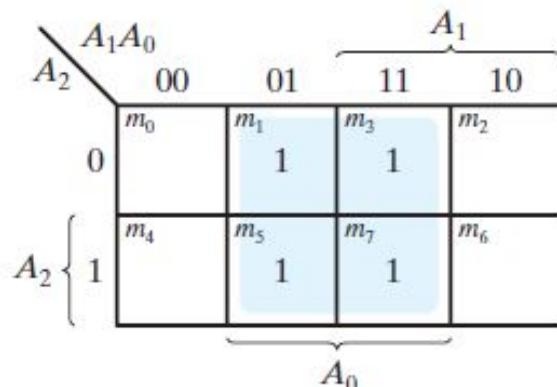
(b)  $T$  Flip-Flop

Excitation Table

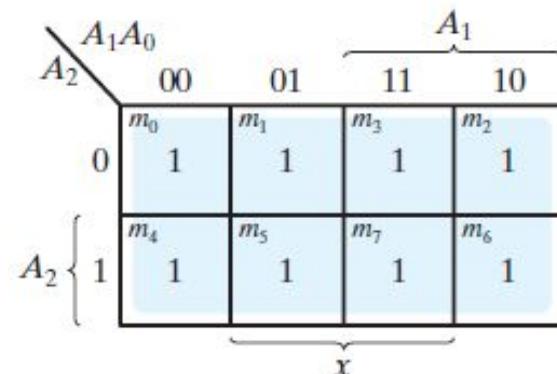
# Synthesis Using T Flip-Flops



$$T_{A2} = A_1A_0$$



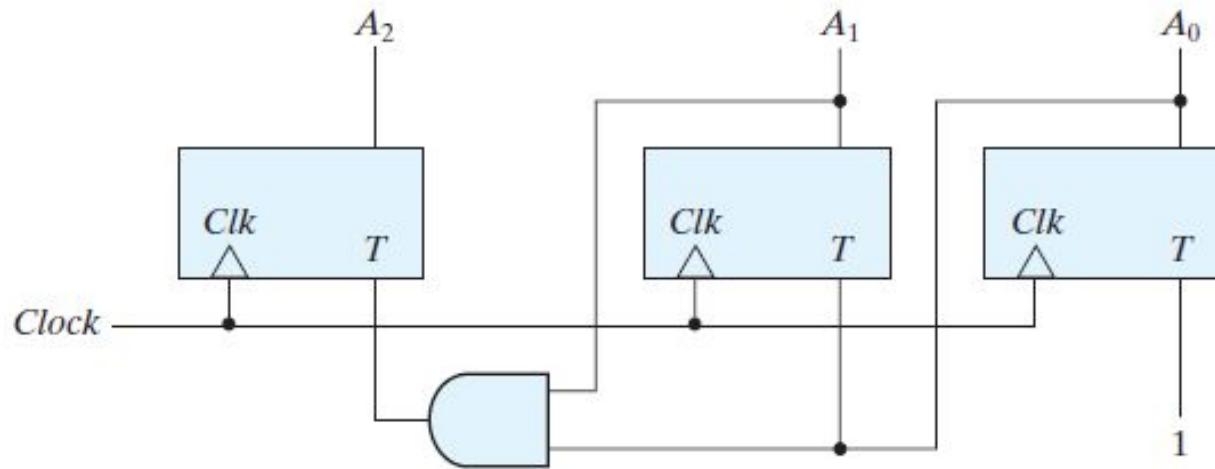
$$T_{A1} = A_0$$



$$T_{A0} = 1$$

Maps for three-bit binary counter

# Synthesis Using T Flip-Flops

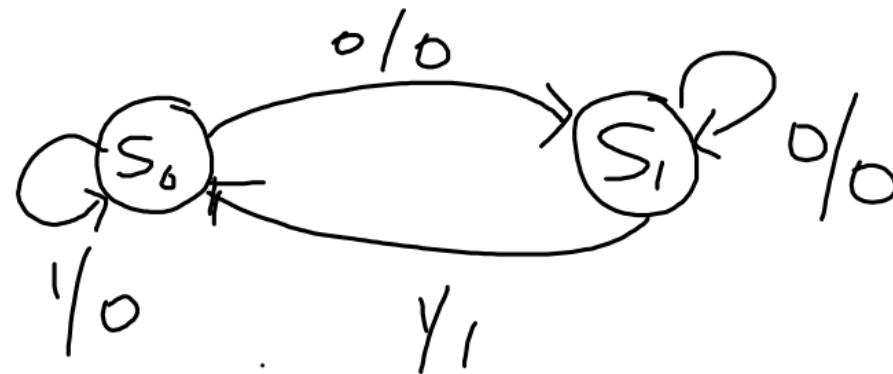


Logic diagram of three-bit binary counter

# Additional questions

---

Design the circuit diagram for mealy state diagram using D flipflop with binary encoding



Steps: write state table, use binary encodings, build transition table , derive equations for next state and output

Think about it: Design the circuit diagram for mealy state diagram using D flipflop with one hot encoding. One-hot encoding for 8 states → needs 8 flip-flops. Binary encoding for 8 states → needs only 3 flip-flops.

# Additional questions

---

Present state	Input X	Next state	output
S0	0	S1	0
S0	1	S0	0
S1	0	S1	0
S1	1	S0	1

AFTER USING BINARY ENCODING-S<sub>0</sub>=0 AND S<sub>1</sub>=1

Present state	Input X	Next state	output
0	0	1	0
0	1	0	0
1	0	1	0
1	1	0	1

# Additional questions

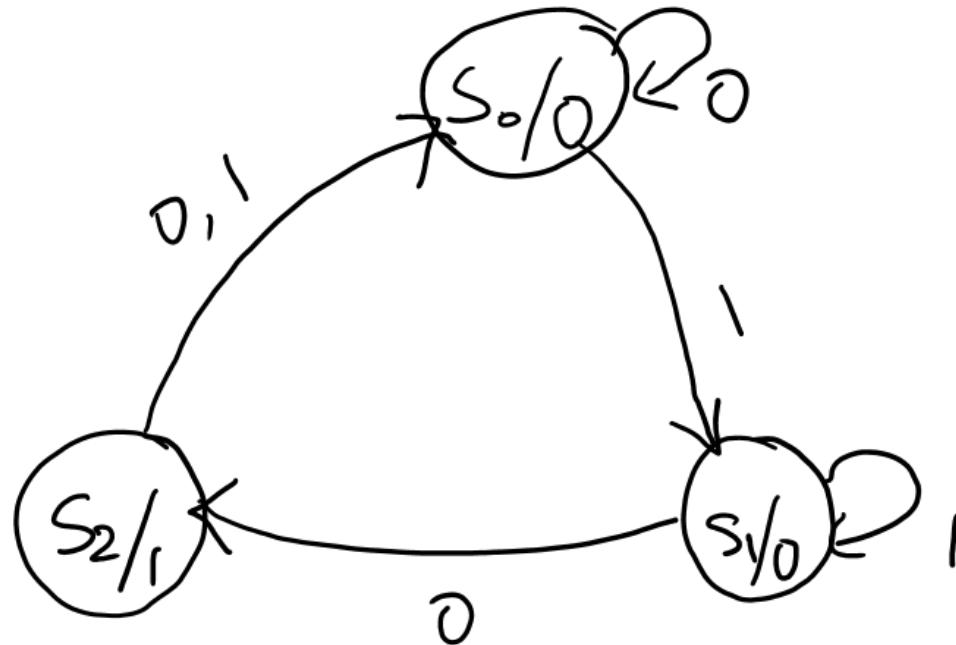
AFTER USING BINARY ENCODING-S<sub>0</sub>-0 AND S<sub>1</sub>=1

Present state A	Input X	Next state A	Output Y
0	0	1	0
0	1	0	0
1	0	1	0
1	1	0	1

Solving using K map next state equation D<sub>A</sub>=x` and y=Ax

# Additional questions

Design the circuit diagram for Moore state diagram using D flipflop with Gray code encoding and one hot encoding



# Summary

---

## Analysis of circuit:

Circuit diagram → Equations – State table → State diagram

The state table can of 2 format: format 1: output depends on present state(Moore model)

format 2: Output depends on present state and input (Mealey Model)

Both cases Next state always depends on present state and input

For JK and T flipflop to build next state in table use characteristic equation

## Design of circuit : From state diagram build Circuit

Steps:(Moore and Mealey model):

State diagram– check for equivalent states– use encoding(binary/gray code/one hot encoding) and build table---select type of flip flop--- get equations for next state and output- build circuit (number of flipflops to be decided on type of encoding)

For JK and T flipflop

Use excitation tables inputs

# Summary

	Goal	Real-Time Benefit	Example
State Reduction	Minimize number of equivalent states	Lower logic complexity → faster response	Sequence detector in network packet filtering
One-Hot Assignment	One bit per state	Faster decoding logic, suitable for FPGA designs	High-speed motor control
Gray Code	Adjacent states differ by 1 bit	Ensures reliable state transitions (only 1-bit change).	Rotary encoder position tracking-As the shaft rotates, only one bit changes at a time
Binary Assignment	Compact encoding	Since fewer bits are used, fewer <b>flip-flops</b> are needed to store the state, reduces hardware cost	Embedded communication controller

A finite state machine (FSM) designed using **one-hot encoding** for 5 states will require:

- A) 3 flip-flops
- B) 4 flip-flops
- C) 5 flip-flops
- D) 6 flip-flops

A sequential circuit is reduced from 8 states to 5 states after **state reduction**. If implemented with binary state assignment, the change in minimum flip-flops required is:

- A) From 4 to 3
- B) From 3 to 2
- C) From 4 to 2
- D) No change

A finite state machine (FSM) designed using **one-hot encoding** for 5 states will require:

- A) 3 flip-flops
- B) 4 flip-flops
- C) 5 flip-flops
- D) 6 flip-flops

**Answer:** C

**Explanation:** In one-hot encoding, **one flip-flop per state** → 5 states → 5 flip-flops.

A sequential circuit is reduced from 8 states to 5 states after **state reduction**. If implemented with binary state assignment, the change in minimum flip-flops required is:

- A) From 4 to 3
- B) From 3 to 2
- C) From 4 to 2
- D) No change

**Answer:** D

**Explanation:**

$$N = 8 \lceil \log_2(8) \rceil = \lceil 3 \rceil = 3 \text{ flip-flops}$$

$$\lceil \log_2(5) \rceil = \lceil 2.321 \rceil = 3 \text{ flip-flops}$$



**PES**  
UNIVERSITY

CELEBRATING 50 YEARS

**THANK YOU**

---

**Team DDCO  
Department of Computer Science and Engineering**



**PES**

UNIVERSITY

CELEBRATING 50 YEARS

# DIGITAL DESIGN AND COMPUTER ORGANIZATION

## Registers

**Team DDCO**

---

**Department of Computer Science and Engineering**

# Registers(T1- section 6.1 and 6.2)

---

## Introduction:

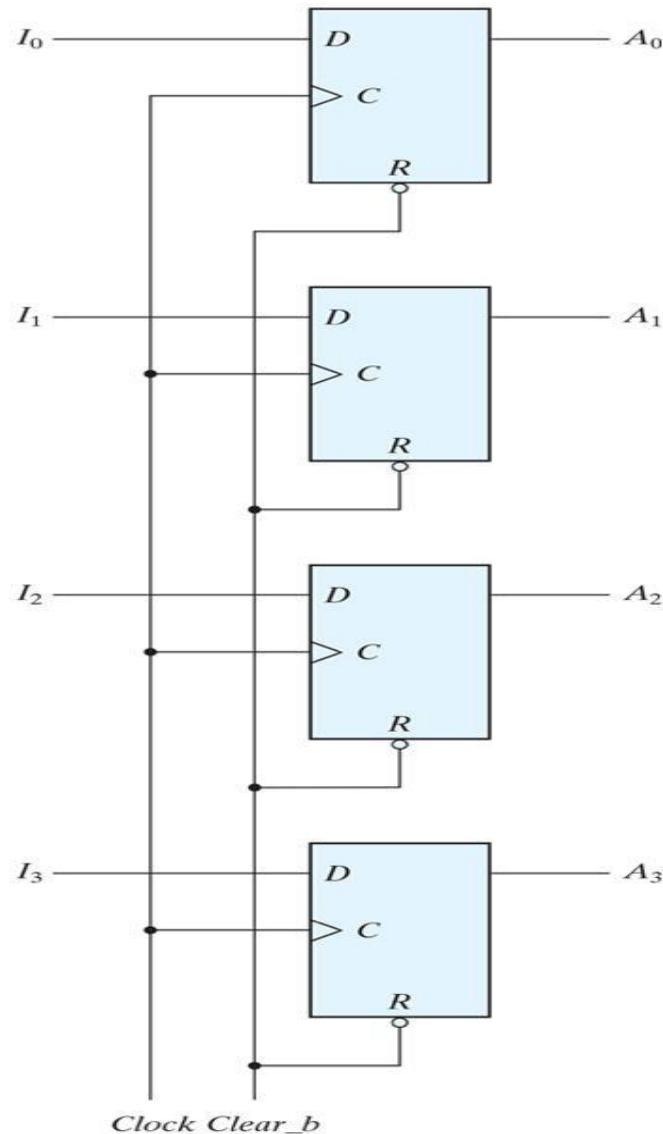
- A register is a **group of flip-flops**, each one of which shares a common clock and is capable of storing one bit of information.
- An n -bit register consists of a group of n flip-flops capable of **storing n bits of binary information**.
- In addition to the flip-flops, **a register may have combinational gates** that perform certain data-processing tasks.
- In its broadest definition, a register consists of a group of flip-flops together with gates that affect their operation.
- **The flip-flops hold the binary information, and the gates determine how the information is transferred into the register.**

# Registers

- A counter is essentially a register that goes through a predetermined sequence of binary states.
- The gates in the counter are connected in such a way as to produce the prescribed sequence of states.
- **The simplest register is one that consists of only flip-flops, without any gates**
- Figure 6.1 shows such a register constructed with four D-type flip-flops to form a four-bit data storage register. **The common clock input triggers all flip-flops on the positive edge of each pulse**, and the binary data available at the four inputs are transferred into the register. The value of (I<sub>3</sub>, I<sub>2</sub>, I<sub>1</sub>, I<sub>0</sub>) immediately before the clock edge determines the value of (A<sub>3</sub>, A<sub>2</sub>, A<sub>1</sub>, A<sub>0</sub>) after the clock edge.
- The input **Clear\_b** goes to the active-low R (reset) input of all four flip-flops. When this input goes to 0, **all flip-flops are reset asynchronously**. The Clear<sub>b</sub> input is useful for clearing the register to all 0's prior to its clocked operation. **The R inputs must be maintained at 1 for normal operation**.

# Registers

**Figure 6.1**  
**Four-bit register.**



# Registers

---

## Registers with Parallel Load:

- **The pulses are applied to all flip-flops and registers in the system.** The master clock acts like a drum that supplies a constant beat to all parts of the system.
- The transfer of new information into a register is referred to as **loading** or **updating** the register. If all the bits of the register are **loaded simultaneously with a common clock pulse**, we say that the loading is done in parallel .
- A clock edge applied to the C inputs of the register of Fig. 6.1 will load all four inputs in parallel. In this configuration, if the contents of the register must be left unchanged, the inputs must be held constant or the clock must be inhibited from the circuit

# Registers

---

## Registers with Parallel Load:

- **Parallel Load Mechanism:**
  - Registers with parallel load **update all bits simultaneously** using a common clock edge.
  - A control signal (**Load**) decides whether **new data** is written or **existing data** is retained.
  - This ensures fast and synchronized data transfer across all bits.
- **Clock Control and Synchronization:**
  - Instead of gating the clock, data flow is controlled at the register inputs using multiplexers.
  - This avoids timing issues caused by variable delays in clock paths.
  - Flip-flops retain or load new data based on the Load signal during each clock pulse.

## Clock Gating (Old Method)

In **clock gating**, you literally **stop or pass** the clock to flip-flops depending on an **Enable** signal.

Example: If  $\text{Enable}=0$ , the clock pulse is blocked; if  $\text{Enable}=1$ , the clock pulse reaches the flip-flops.

### Problem:

The clock has to travel through extra logic (AND gates, buffers, etc.).

These extra delays can differ slightly for different flip-flops.

Result: Not all flip-flops see the clock edge at exactly the same time → **timing skew**.  
Skew can cause some flip-flops to update while others don't, leading to corrupted register values.

## Input Control with Multiplexers (Better Method)

Instead of touching the clock, keep the **clock clean and uniform** for all flip-flops.

Use a **multiplexer (MUX)** in front of each flip-flop's **D** input:

One input of the MUX is the **new data (D\_in)**.

The other input is the **old output (Q)** fed back.

A **Load (Enable)** signal selects which one to send to the flip-flop's D input.

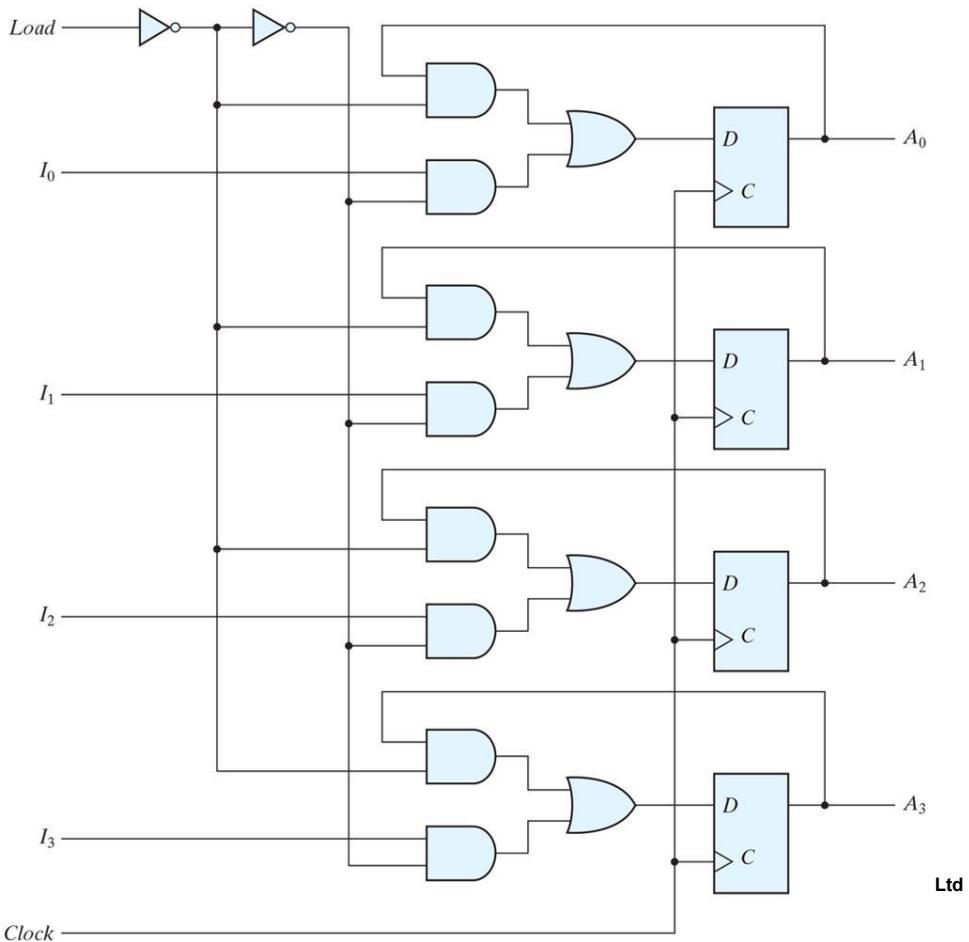
# Registers with Parallel Load:

The transfer of information from the data inputs or the outputs of the register is done simultaneously with all four bits in response to a clock edge.

**Figure 6.2**  
**Four-bit register**  
**with parallel load.**

Load=0 retains data

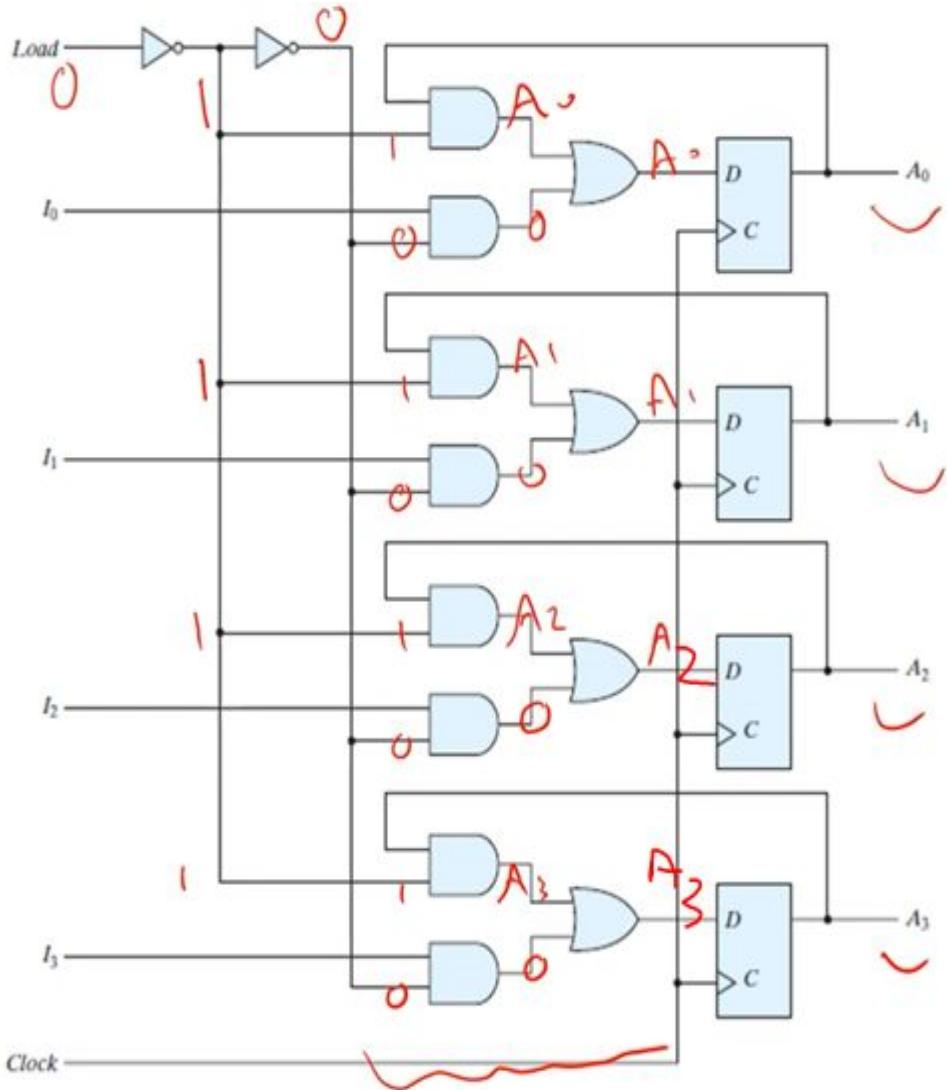
Load =1  
Register accepts new data



# Registers with Parallel Load:

**Figure 6.2**  
**Four-bit register**  
**with parallel load.**

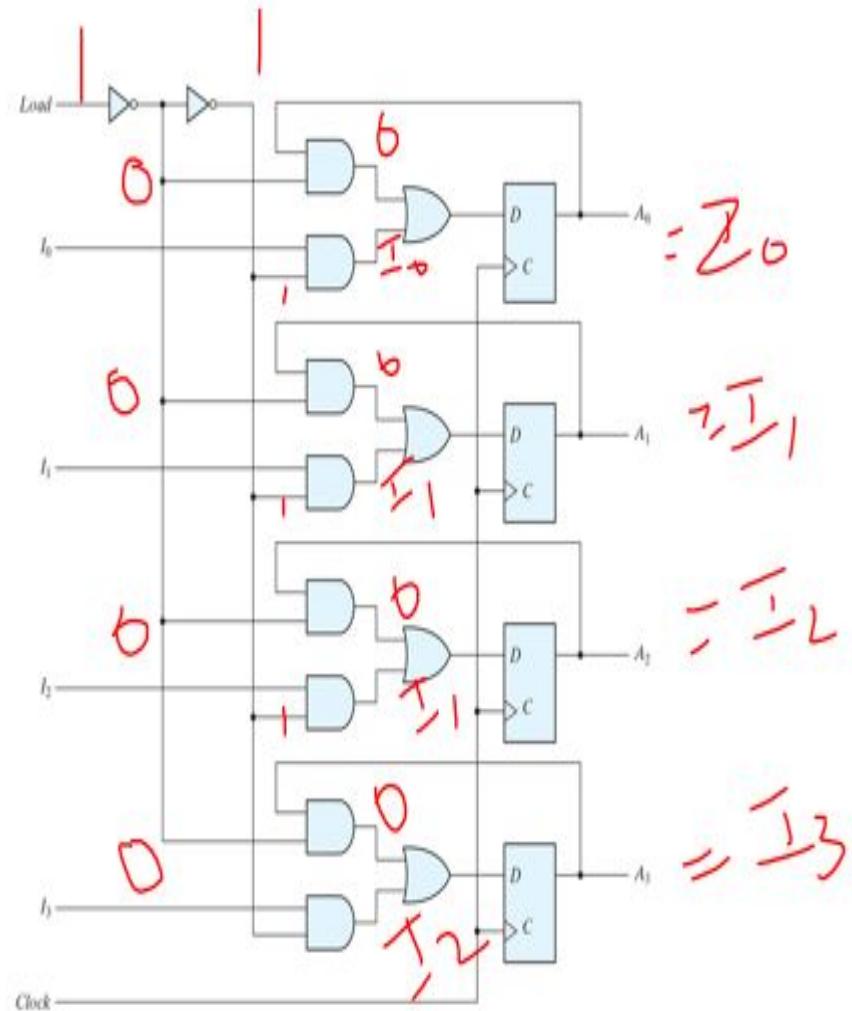
Load=0 retains data



# Registers with Parallel Load:

**Figure 6.2**  
**Four-bit register**  
**with parallel load.**

Load = 1  
 Register accepts new data



# Registers

---

## Shift Registers: section 6.2(T1)

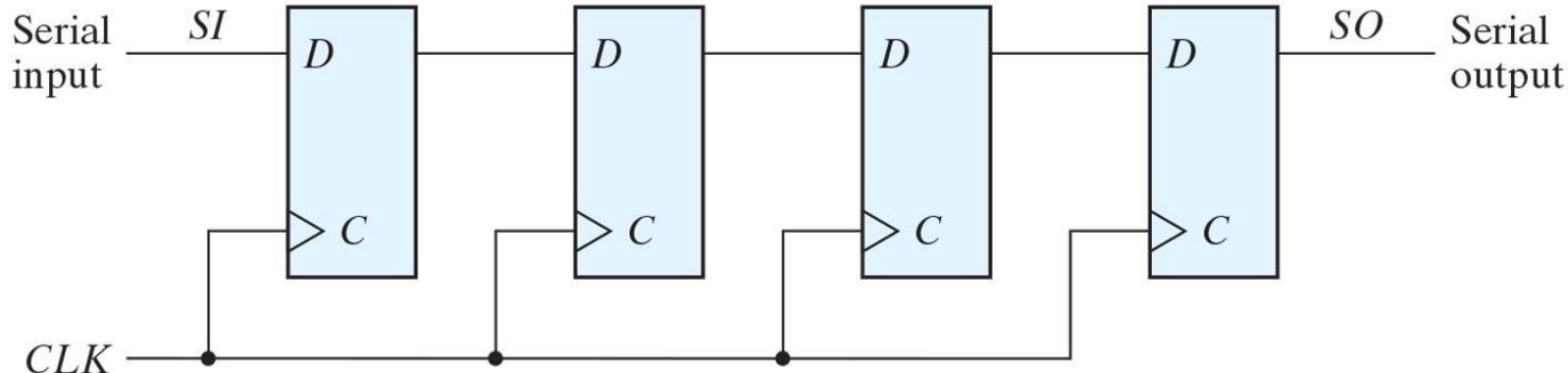
**A register capable of shifting the binary information held in each cell to its neighboring cell, in a selected direction, is called a shift register.**

The logical configuration of a shift register consists of a chain of flip-flops in cascade, **with the output of one flip-flop connected to the input of the next flip-flop.**

All flip-flops receive common clock pulses, which activate the shift of data from one stage to the next.

# Shift Registers

**Figure 6.3**  
**Four-bit shift register.**



The output of a given flip-flop is connected to the D input of the flip-flop at its right.

This shift register is **unidirectional** (left-to-right).

The serial input determines what goes into the leftmost flip-flop during the shift.

The serial output is taken from the output of the rightmost flip-flop.

# Shift Registers

---

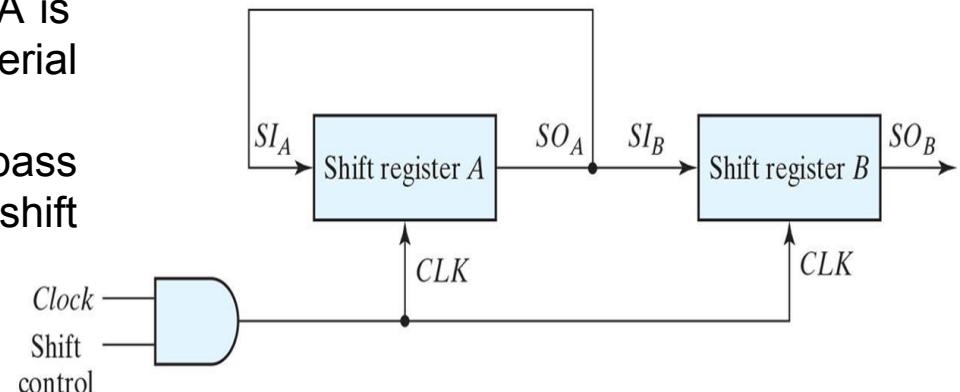
## Serial Transfer:

- Serial transfer occurs **one bit at a time**, unlike parallel transfer where **all bits move together**.
- Data is shifted from **register A to register B**, using **shift registers** connected in series.
- To prevent data loss, register A is made to **circulate its own contents**, while register B shifts in the data.
- A **control signal** (Shift Control) enables shifting for **a fixed number of clock cycles**.
- When active, the signal allows **4 clock pulses (T1–T4)** through AND gates to both registers.
- Each **rising clock edge** shifts data one step, syncing both registers with each clock pulse.

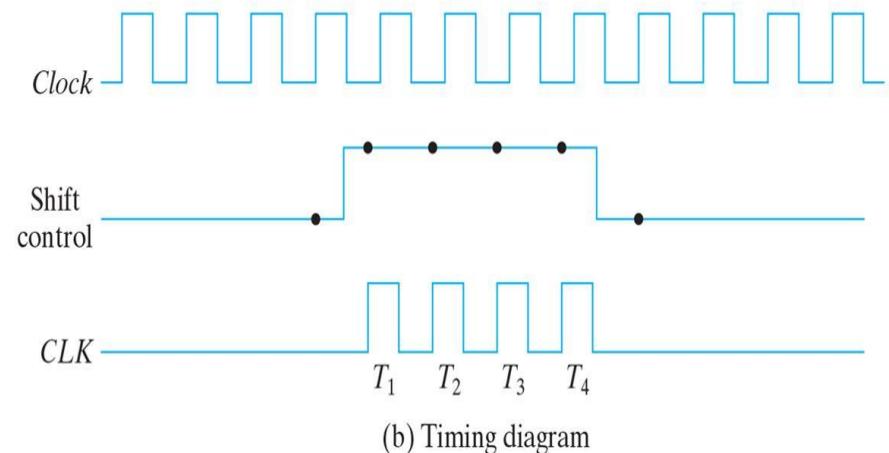
# Serial Transfer

The serial output ( SO ) of register A is connected to the serial input ( SI ) of register B. To prevent the loss of information stored in the source register, the information in register A is made to circulate by connecting the serial output to its serial input.

AND gate that allows clock pulses to pass into the CLK terminals only when the shift control is active.



(a) Block diagram



(b) Timing diagram

**Figure 6.4**  
**Serial transfer from register A to register B.**

# Serial Transfer

---

## Summary:

Two shift registers: **A** (source) and **B** (destination).

Data is transferred **serially** bit by bit from register A → register B.

A clock signal drives both registers.

But the transfer should **not happen all the time** → only when we want.

That's where **Shift Control** comes in.

The **Shift Control** signal is ANDed with the clock before reaching the registers.

If **Shift Control = 1**:

The AND gate passes the clock pulses.

Registers A and B **shift on each clock edge**.

Data moves serially from A → B.

If **Shift Control = 0**:

Clock pulses are blocked.

Registers **hold their data** (no shifting).

So **Shift Control acts like an enable signal for shifting**.

# Serial Transfer

---

With the first pulse,  $T_1$ , the rightmost bit of A is shifted into the leftmost bit of B and is also circulated into the leftmost position of A. At the same time, all bits of A and B are shifted one position to the right. The previous serial output from B in the rightmost position is lost, and its value changes from 0 to 1.

<b>Timing Pulse</b>	<b>Shift Register A</b>				<b>Shift Register B</b>			
Initial value	1	0	1	1	0	0	1	0
After $T_1$	1	1	0	1	1	0	0	1
After $T_2$	1	1	1	0	1	1	0	0
After $T_3$	0	1	1	1	0	1	1	0
After $T_4$	1	0	1	1	1	0	1	1

**Table 6.1 Serial-Transfer Example.**

Thus, the contents of A are copied into B, so that the contents of A remain unchanged i.e., the contents of A are restored to their original value.

# Serial Transfer

---

- In the parallel mode, information is available from all bits of a register and all bits can be transferred simultaneously during one clock pulse.
- In the serial mode mode, the registers have a single serial input and a single serial output. The information is transferred one bit at a time while the registers are shifted in the same direction.

# Shift Registers

## Serial Addition:

- Operations in digital computers are usually done in **parallel because that is a faster mode of operation**. Serial operations are slower because a datapath operation takes several clock cycles, but **serial operations have the advantage of requiring fewer hard ware components**. In VLSI circuits, they require less silicon area on a chip
- The two binary numbers to be added serially are stored in two shift registers. Two binary numbers are stored in shift registers A (augend) and B (addend). A single full adder adds one bit pair at a time, starting from the least significant bit (LSB).
- The carry-out is stored in a D flip-flop, which feeds into the carry-in of the next bit pair in the next clock cycle.
- With each clock pulse, both registers shift right, a new sum is generated, and carry is updated.
- This continues for  $n$  cycles ( $n = \text{number of bits}$ ) until the shift control is disabled, completing the addition process.



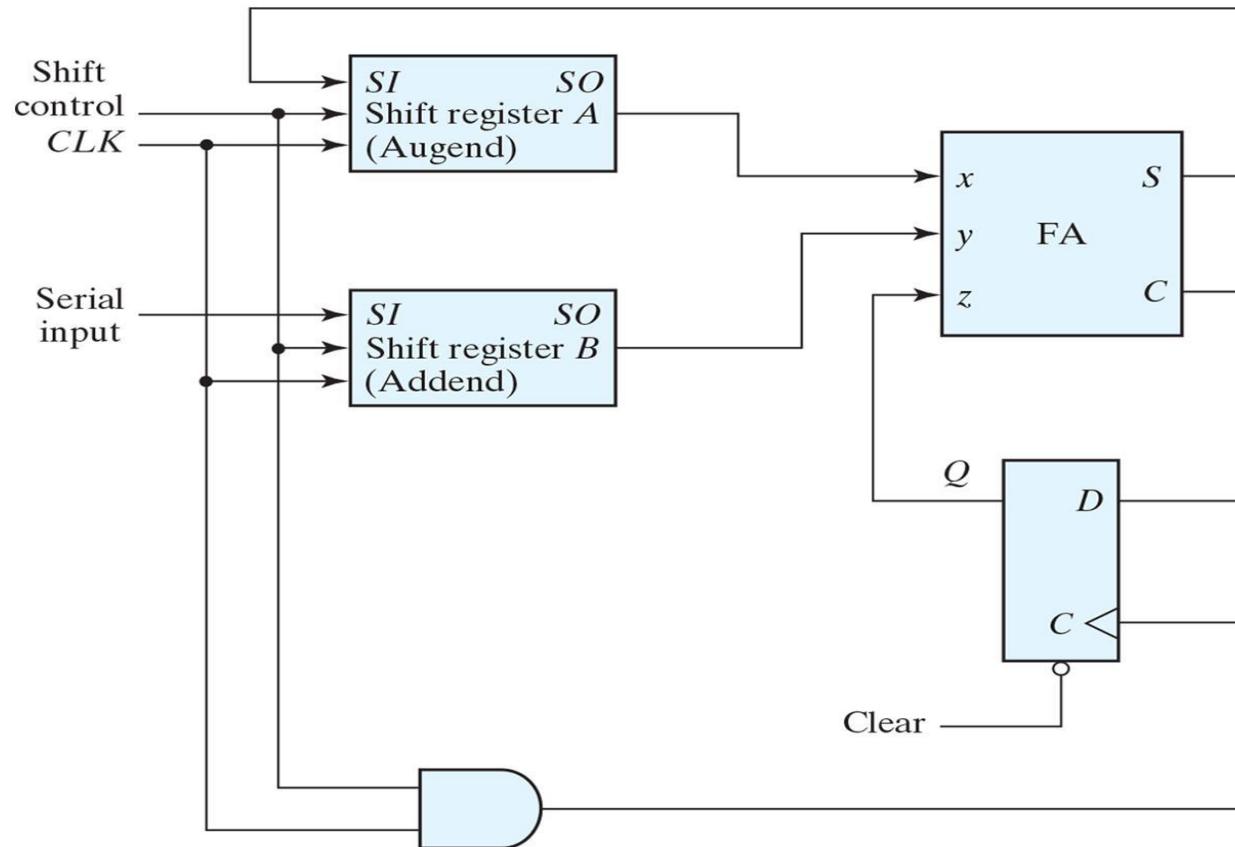
# Serial Addition

---

- It is used perform bit by bit addition in serial form
- It is done using flipflop and adder
- D flipflop- stores carry output after addition
- Right shit register A-A
- Right shift register-B-B
- Add A+B bit by bit using Full adder

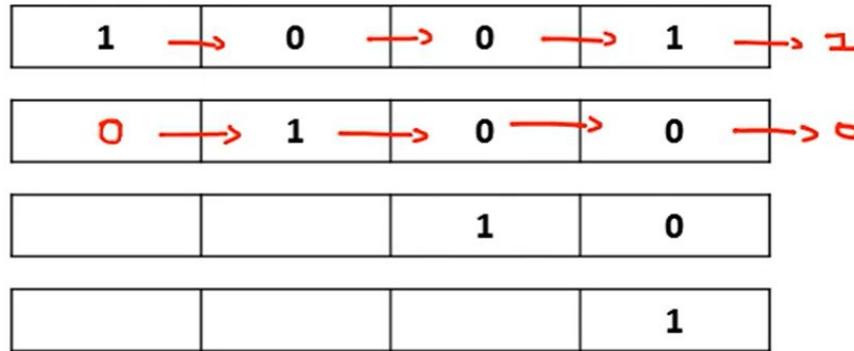
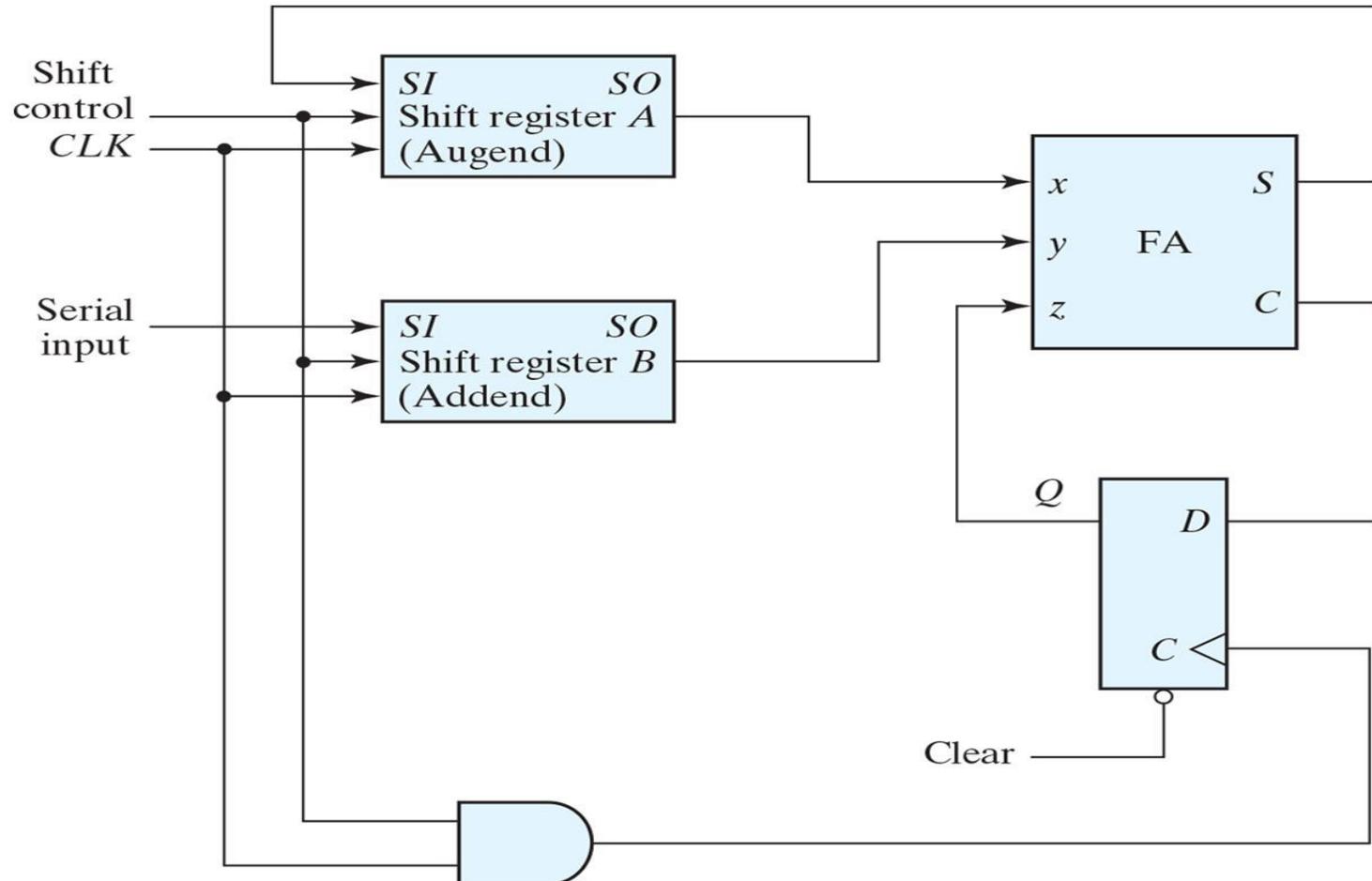
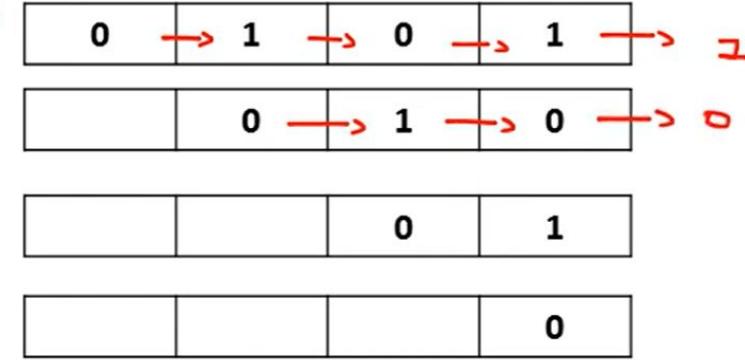
Initially, register A holds the augend, register B holds the addend, and the **carry flip-flop is cleared to 0**. The outputs ( SO ) of A and B provide a pair of significant bits for the full adder at x and y. Output Q of the flip-flop provides the input carry at z. The shift control enables both registers and the carry flip-flop, so at the next clock pulse, both registers are shifted once to the right, the sum bit from S enters the leftmost flip-flop of A, and the output carry is transferred into flip-flop Q. The shift control enables the registers for a number of clock pulses equal to the number of bits in the registers. For each succeeding clock pulse, a new sum bit is transferred to A, a new carry is transferred to Q, and both registers are shifted once to the right

# Serial Addition



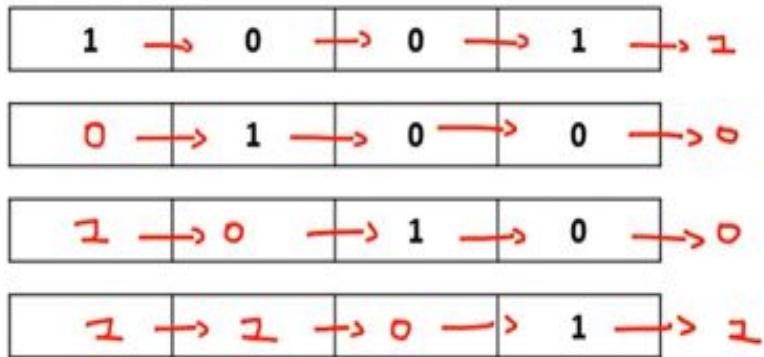
**Figure 6.5: Serial adder.**

Z is Cin  
C is C out  
Q=0 initially

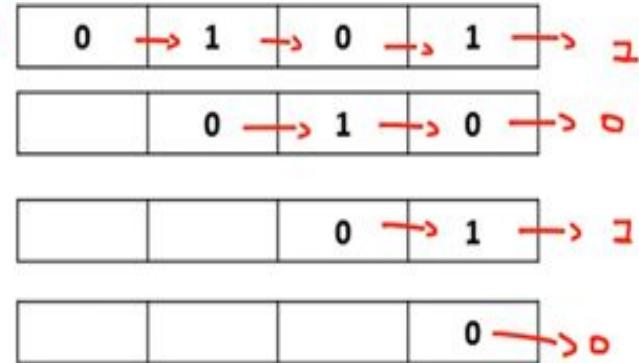
**A****B**

# Serial Addition

A

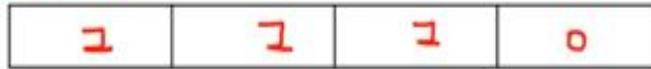


B



$$\begin{array}{r} A = 1001 \\ + B = 0101 \\ \hline 1110 \end{array}$$

A-Registers =



# Serial Addition

---

Comparing the serial adder with the parallel adder described in, we note several differences.

- The parallel adder uses **registers** with a parallel load, whereas the serial adder uses shift registers.
- The number of **full-adder circuits** in the parallel adder is equal to the number of bits in the binary numbers, whereas the serial adder requires only one full-adder circuit and a carry flip-flop.
- Excluding the registers, the parallel adder is a **combinational** circuit, whereas the serial adder is a **sequential circuit** which consists of a full adder and a flip-flop that stores the output carry.
- This design is typical in serial operations because the result of a bit-time operation may depend not only on the present inputs, but also on previous inputs that must be stored in flip-flops

# Serial Addition

---

To show that serial operations can be designed by means of **sequential circuit procedure**, we will **redesign** the serial adder with the use of a state table.

First, we assume that two shift registers are available to store the binary numbers to be added serially. The serial outputs from the registers are designated by  $x$  and  $y$ .

# Serial Addition

**Table 6.2**  
**State Table for Serial Adder.**

The next state of Q is equal to the output carry.

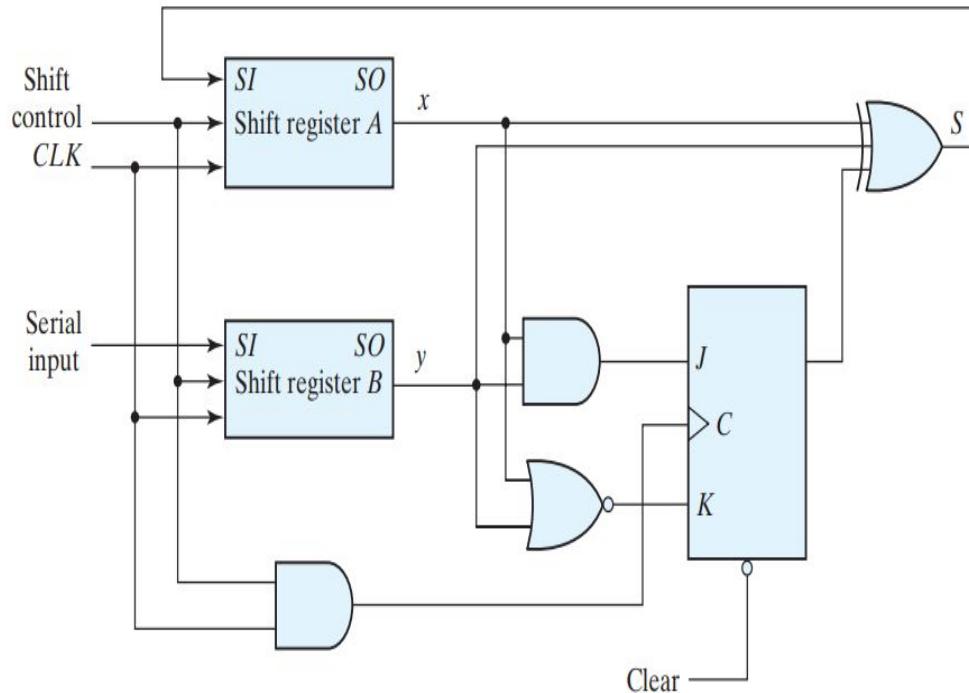
<b>Present State</b>	<b>Inputs</b>		<b>Next State</b>	<b>Output</b>	<b>Flip-Flop Inputs</b>			
	<b>Q</b>	<b>x</b>	<b>y</b>		<b>Q</b>	<b>S</b>	<b>J<sub>Q</sub></b>	<b>K<sub>Q</sub></b>
0	0	0		0	0		0	X
0	0	1		0	1		0	X
0	1	0		0	1		0	X
0	1	1		1	0		1	X
1	0	0		0	1		X	1
1	0	1		1	0		X	0
1	1	0		1	0		X	0
1	1	1		1	1		X	0

# Serial Addition

$$J_Q = xy$$

$$K_Q = x'y' = (x + y)'$$

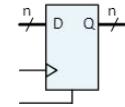
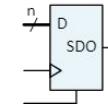
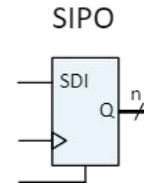
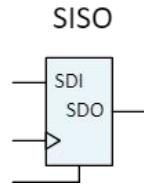
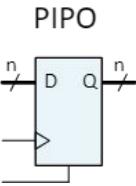
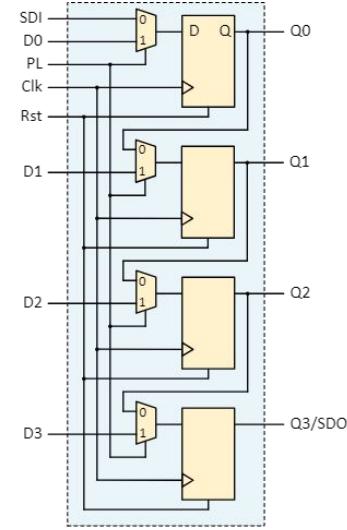
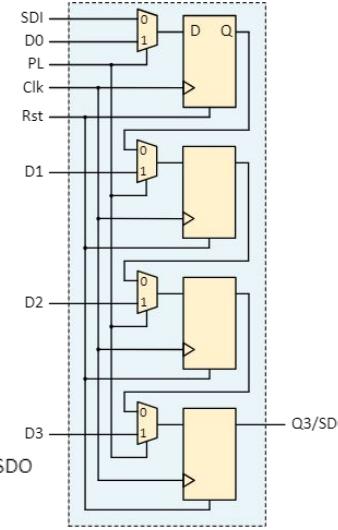
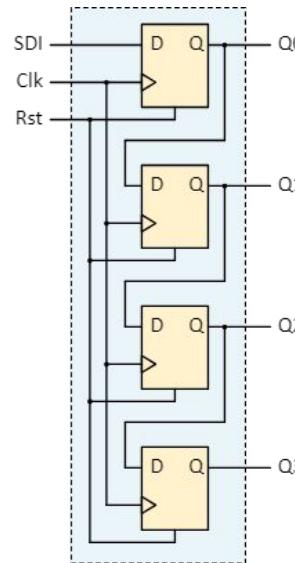
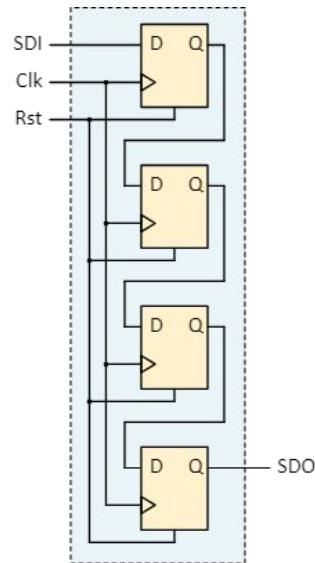
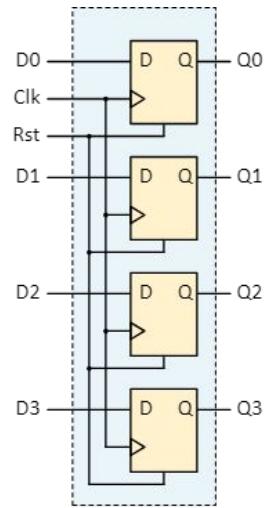
$$S = x \oplus y \oplus Q$$



**FIGURE 6.6**  
Second form of serial adder

# Types of Representation of Registers

## Universal register



The universal register, as the name implies, can be used as a PIPO, SISO, SIPO, or PISO register.

# Applications

---

## Parallel Data Transfer

Where used: Communication between processor and peripherals.

Example: In a keyboard interface, parallel registers capture all key signals at once for quick transfer to the processor.

Why important: Transfers multiple bits simultaneously, ideal for high-speed operations.

## Serial Data transfer:

USB as Serial Communication

Every time you connect a pen drive to your computer, the data is transferred serially (one bit at a time)

uses high-speed serial communication over just a few wires instead of large parallel buses.

# Applications

---

## Universal Shift Registers

Where used: Versatile data handling.

Example: In data encryption hardware, universal shift registers can load, shift, and output data in multiple formats.

Why important: Provides flexibility for complex digital operations.

In a 4-bit parallel load register, what happens when the Load signal is 0 during a clock pulse?

- A) All bits are shifted to the right
- B) The register is cleared to 0000
- C) The existing data is retained
- D) The register loads new data from inputs

Which of following is true about serial Transfer?

- A. In the serial mode, the registers have a single serial input and a single serial output
- B. If Shift Control = 0: The AND gate passes the clock pulses.  
Registers A and B shift on each clock edge.
- C. In the serial mode, information is available from all bits of a register and all bits can be transferred simultaneously during one clock pulse.
- D. all statements are true

In a 4-bit parallel load register, what happens when the Load signal is 0 during a clock pulse?

- A) All bits are shifted to the right
- B) The register is cleared to 0000
- C) The existing data is retained
- D) The register loads new data from inputs

Answer: C – The existing data is retained when Load = 0, since no new parallel input is latched.

Which of following is true about serial Transfer?

- A. In the serial mode, the registers have a single serial input and a single serial output
- B. If Shift Control = 0: The AND gate passes the clock pulses. Registers A and B shift on each clock edge.
- C. In the serial mode, information is available from all bits of a register and all bits can be transferred simultaneously during one clock pulse.
- D. all statements are true

ans: A



**PES**  
UNIVERSITY

CELEBRATING 50 YEARS

**THANK YOU**

---

**Team DDCO  
Department of Computer Science and Engineering**



**PES**

UNIVERSITY

CELEBRATING 50 YEARS

# DIGITAL DESIGN AND COMPUTER ORGANIZATION

## Ripple counters

---

**Team DDCO**

**Department of Computer Science and Engineering**

# Ripple Counters(T1- section 6.3)

---

In digital logic and computing, a counter is a device which stores (and sometimes displays) the **number of times a particular event or process** has occurred, often in relationship to a clock

A register that goes through a **prescribed sequence of states upon the application of input pulses is called a counter**. The input pulses may be clock pulses, or they may originate from some external source and may occur at a fixed interval of time or at random.

The sequence of states may follow the binary number sequence or any other sequence of states. A counter that follows the binary number sequence is called a **binary counter** . An **n -bit binary counter consists of n flip-flops and can count in binary from 0 through  $2^n - 1$** .

EX: 3 bit counter- consists of 3 flipflops, counts from 0-7 (000-111)

# Ripple Counters

---

Counters are available in two categories: **ripple counters and synchronous counters.**

In a ripple counter, a flip-flop output transition serves as a source for triggering other flip-flops. In other words, the C input of some or all flip-flops are triggered, not by the common clock pulses, but rather by the transition that occurs in other flip-flop outputs.

In a synchronous counter, the C inputs of all flip-flops receive the common clock

Counters (Ripple, synchronous)- **upcounter, down counter, updown counter**

# Ripple Counters

<b>Asynchronous / Ripple Counter</b>	<b>Synchronous Counter</b>
Output of one flip-flop drives the clock of the next flip-flop.	No connection between output of one flip-flop and clock of next flip-flop.
Flip-flops are not clocked simultaneously.	Flip-flops are clocked simultaneously.
Circuit is simple even for more number of states.	Circuit becomes complicated as number of states increases.
Speed is slow since clock is propagated through multiple stages.	Speed is high as clock is given to all flip-flops at the same time.

# Binary Ripple Counters

---

## Binary Ripple Counter:

A binary ripple counter consists of a series connection of complementing flip-flops, with the output of each flip-flop connected to the C input of the next higher order flip-flop.

Can be obtained by:

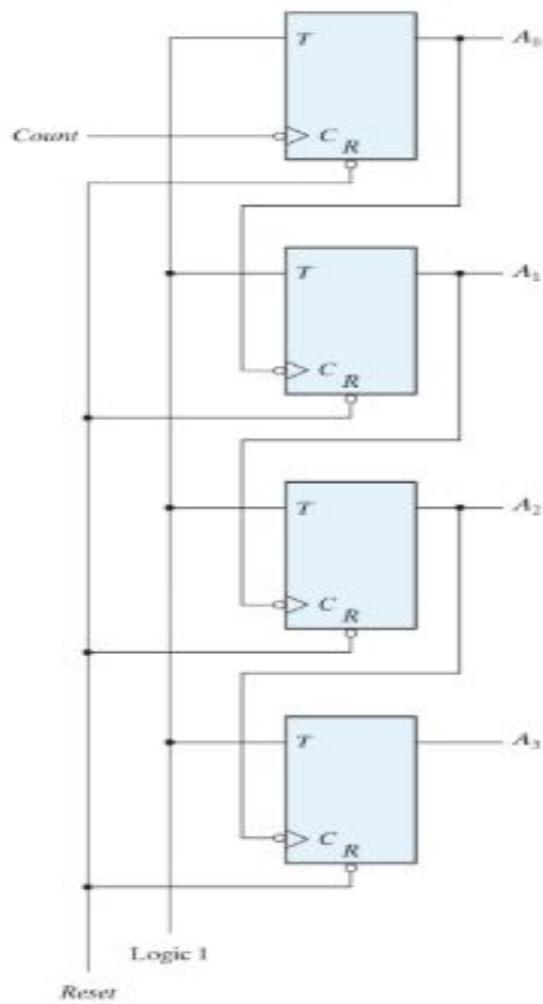
JK connected to same input and converting it into toggle (T flipflop)

D flipflop- D input is always the complement of the present state, and the next clock pulse will cause the flip-flop to complement.

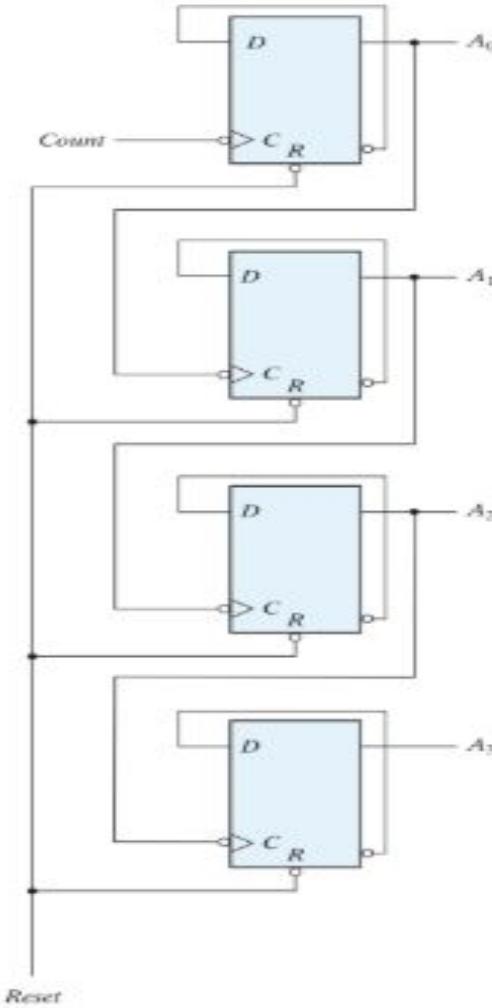
The bubble in front of the dynamic indicator symbol next to C indicates that the flip-flops respond to the negative-edge transition of the input. (transition from 1 to 0)

# Binary Ripple Counters-

Figure 6.8  
Four-bit binary ripple counter.



(a) With *T* flip-flops



(b) With *D* flip-flops

# Binary Ripple Counters

---

Binary Ripple counter:

The flip-flop holding the least significant bit receives the incoming count pulses.

The T inputs of all the flip-flops in

(a) are connected to a permanent logic 1, making each flip-flop complement if the signal in its C input goes through a negative transition. The bubble in front of the dynamic indicator symbol next to C indicates that the flip-flops respond to the negative-edge transition of the input. The negative transition occurs when the output of the previous flip-flop to which C is connected goes from 1 to 0.

# Binary Ripple Counters

## Binary Ripple Counter:

**Table 6.4**  
*Binary Count Sequence*

<b><math>A_3</math></b>	<b><math>A_2</math></b>	<b><math>A_1</math></b>	<b><math>A_0</math></b>
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0

# Binary Ripple Counters

## Four-bit Binary Ripple Counter.

Binary Ripple counter : four-bit binary ripple counter

4 flipflop

$2^4 = 16$  states

0-15 sequence(0000-1111)

The count starts with binary 0 and increments by 1 with each count pulse input. After the count of 15, the counter goes back to 0 to repeat the count

Working-

The least significant bit, A0, is complemented with each count pulse input.

Every time that A0 goes from 1 to 0, it complements A1

Every time that A1 goes from 1 to 0, it complements A2.

Every time that A2 goes from 1 to 0, it complements A3

# Binary Ripple Counters

---

- A binary counter with a reverse count is called a **binary countdown counter**. In a countdown counter, the binary count is decremented by 1 with every input count pulse.
- The count of a four-bit countdown counter starts from binary 15 and continues to binary counts 14, 13, 12, . . . , 0 and then back to 15
- A list of the count sequence of a binary countdown counter shows that the least significant bit is complemented with every count pulse. Any other bit in the sequence is complemented if its previous least significant bit goes from **0 to 1**.
- **all flip-flops trigger on the positive edge of the clock**
- If negative-edge-triggered flip-flops are used, then the C input of each flip-flop must be connected to the complemented output of the previous flip-flop. Then, when the true output goes from 0 to 1, the complement will go from 1 to 0 and complement the next flip-flop as required.



# BCD Ripple Counters

---

A decimal counter follows a sequence of 10 states and returns to 0 after the count of 9.

Such a counter must have at least four flip-flops to represent each decimal digit, since a decimal digit is represented by a binary code with at least four bits

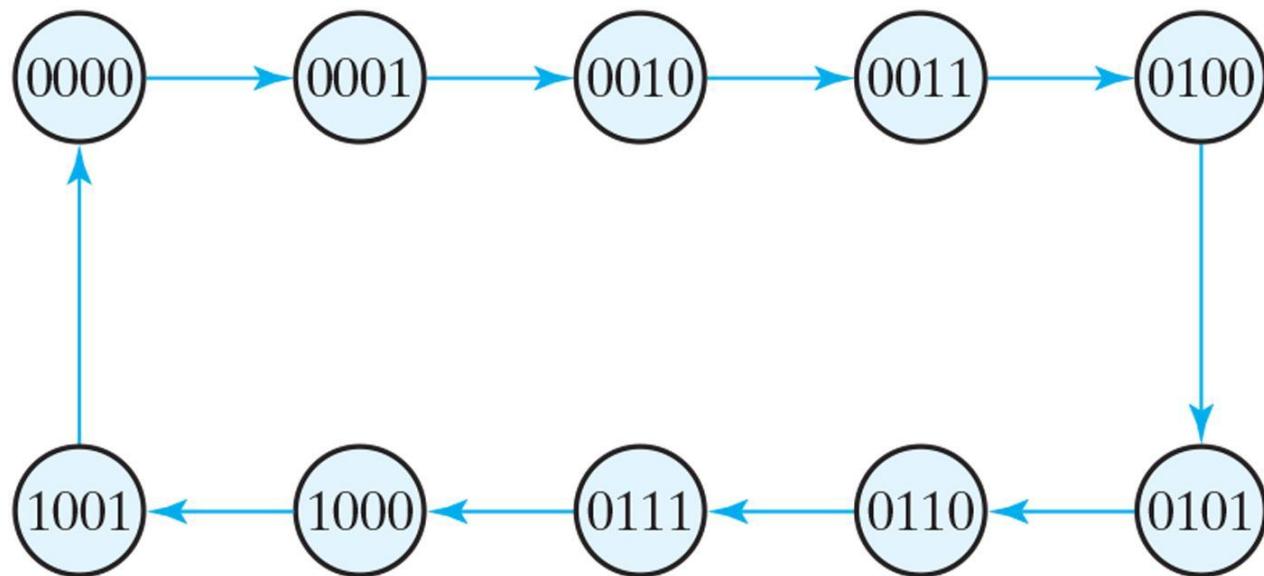
Also called Mod 10 counter.

$$2^n = 10$$

$$N=4$$

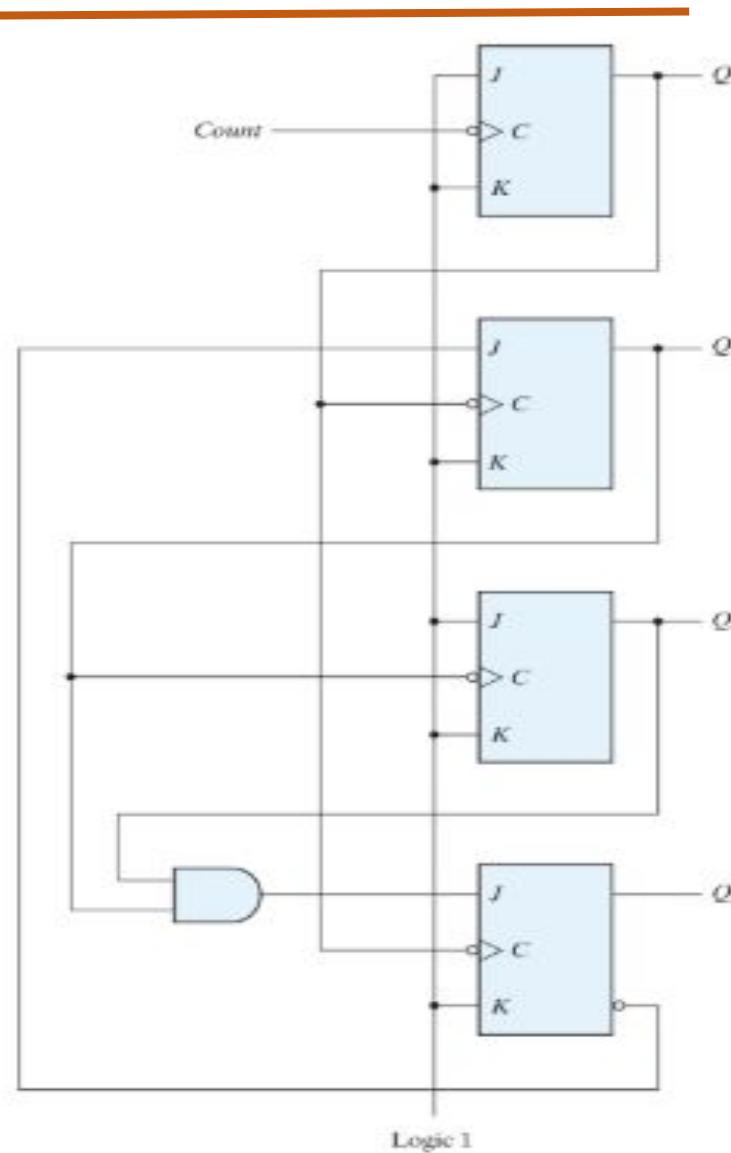
# BCD Ripple Counters

**Figure 6.9**  
**State diagram of a decimal BCD counter.**



# BCD Ripple Counters

**Figure 6.10**  
**BCD ripple counter.**

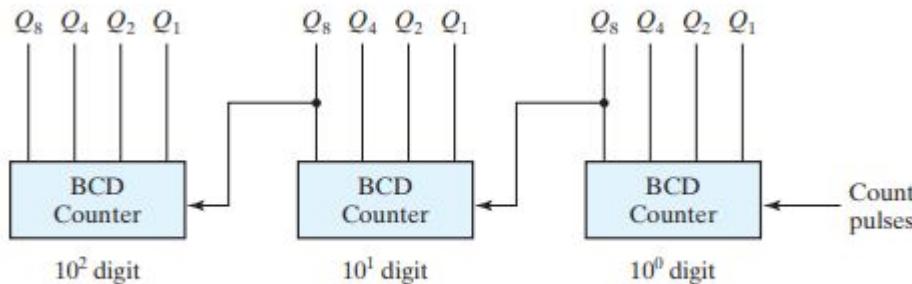


# BCD Ripple Counters

---

- A ripple counter is an asynchronous sequential circuit. Signals that affect the flip-flop transition depend on the way they change from 1 to 0
- when the C input goes from 1 to 0, the flip-flop is set if  $J = 1$ , is cleared if  $K = 1$ , is complemented if  $J = K = 1$ , and is left unchanged if  $J = K = 0$ .
- Q1 changes state after each clock pulse. Q2 complements every time Q1 goes from 1 to 0, as long as  $Q_8 = 0$ . When  $Q_8$  becomes 1, Q2 remains at 0. Q4 complements every time Q2 goes from 1 to 0.  $Q_8$  remains at 0 as long as Q2 or Q4 is 0. When both Q2 and Q4 become 1,  $Q_8$  complements when Q1 goes from 1 to 0.  $Q_8$  is cleared on the next transition of Q1.

# BCD Ripple Counters



**FIGURE 6.11**  
 Block diagram of a three-decade decimal BCD counter

The BCD counter of Fig. 6.10 is a decade counter, since it counts from 0 to 9. To count in decimal from 0 to 99, we need a two-decade counter. To count from 0 to 999, we need a three-decade counter. Multiple decade counters can be constructed by connecting BCD counters in cascade, one for each decade. A three-decade counter is shown in Fig. 6.11 . The inputs to the second and third decades come from Q8 of the previous decade. When Q8 in one decade goes from 1 to 0, it triggers the count for the next higher order decade while its own decade goes from 9 to 0.

# Applications

---

In automated industries like **bottling plants, packaging units, and textile mills**, machines need to **count repetitive events** — e.g., bottles filled, packets sealed, or cloth rolls measured. A **counter** circuit is ideal because it can automatically keep track of such pulses and trigger control actions.

**Ripple Counter Stage :** The sensor pulses are fed into a binary ripple counter.

Example: a 4-bit ripple counter can count from 0000 (0) to 1111 (15).

Each flip-flop toggles on the falling edge of the previous flip-flop's output → “ripple effect.”

## Count Expansion (Cascading)

If we need to count higher numbers (e.g., 1000 bottles), multiple counters are cascaded (seconds → minutes analogy in clocks).

For example, 3 cascaded BCD ripple counters give **000–999** counts.

### Step-by-Step Ripple Counter Operation

Bottle 1 passes → Counter = 0001

Bottle 2 passes → Counter = 0010...

Bottle 15 passes → Counter = 1111 (overflow → resets back to 0)

Cascading ensures higher count capacity (e.g., up to 9999).

# MCQ

---

In a 4-bit down ripple counter, the initial state is 1111. After 5 clock pulses, the state will be:

- A) 1010
- B) 1011
- C) 1101
- D) 1001

The modulus of a BCD counter is:

- A) 8
- B) 10
- C) 12
- D) 16

# MCQ

---

In a 4-bit down ripple counter, the initial state is 1111. After 5 clock pulses, the state will be:

- A) 1010
- B) 1011
- C) 1101
- D) 1001

**Answer:** B) 1011

**Explanation:** Sequence goes 1111 (15) → 1110 (14) → 1101 (13) → 1100 (12) → 1011 (11).

The modulus of a BCD counter is:

- A) 8
- B) 10
- C) 12
- D) 16

**Answer:** B) 10

**Explanation:** BCD counter counts 10 states (0–9). After 1001, it resets to 0000.

# MCQ

---

If a BCD counter is implemented with 4 flip-flops, how many states are unused?

- A) 2
- B) 4
- C) 6
- D) 10

In a 2-digit decimal counter, how many flip-flops are required?

- A) 4
- B) 6
- C) 7
- D) 8

# MCQ

---

If a BCD counter is implemented with 4 flip-flops, how many states are unused?

- A) 2
- B) 4
- C) 6
- D) 10

**Answer:** C) 6

**Explanation:** 4 flip-flops = 16 possible states (0000–1111). Only 10 (0000–1001) are used. Remaining 6 (1010–1111) are **unused**.

In a 2-digit decimal counter, how many flip-flops are required?

- A) 4
- B) 6
- C) 7
- D) 8

**Answer:** D) 8

**Explanation:** One BCD counter needs 4 flip-flops. For 2-digit (00–99), need 2 counters → total 8 flip-flops.



**PES**  
UNIVERSITY

CELEBRATING 50 YEARS

**THANK YOU**

---

**Team DDCO  
Department of Computer Science and Engineering**



**PES**

UNIVERSITY

CELEBRATING 50 YEARS

# DIGITAL DESIGN AND COMPUTER ORGANIZATION

## Synchronous Counters

---

**Team DDCO**

**Department of Computer Science and Engineering**

# Synchronous Counter(T1-section 6.4)

---

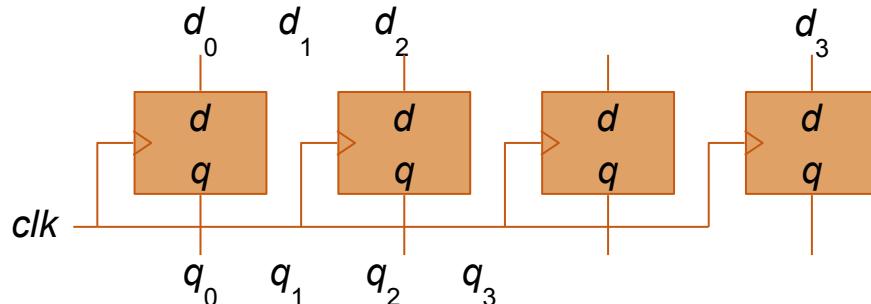
Synchronous counters are different from ripple counters in that clock pulses are applied to the inputs of all flip-flops. **A common clock triggers all flip-flops simultaneously, rather than one at a time in succession as in a ripple counter.**

The decision whether a flip-flop is to be complemented is determined from the values of the data inputs, such as T or J and K at the time of the clock edge. If  $T = 0$  or  $J = K = 0$ , the flip-flop does not change state. If  $T = 1$  or  $J = K = 1$ , the flip-flop complements

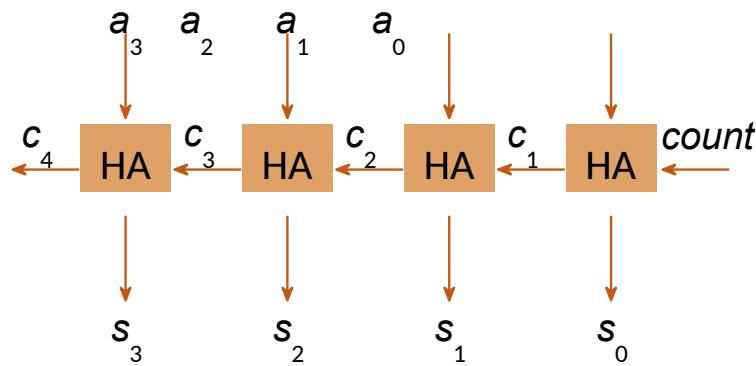
# Counters -Recap

**Incrementing Counters- Example- count number of people entering a mall**

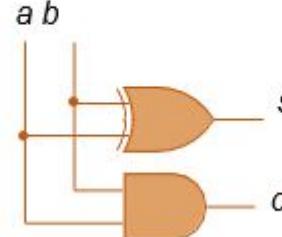
- How to store  $n$ -bits?  $n$ -bit register



- How to increment an  $n$ -bit number?  $n$ -bit incrementer



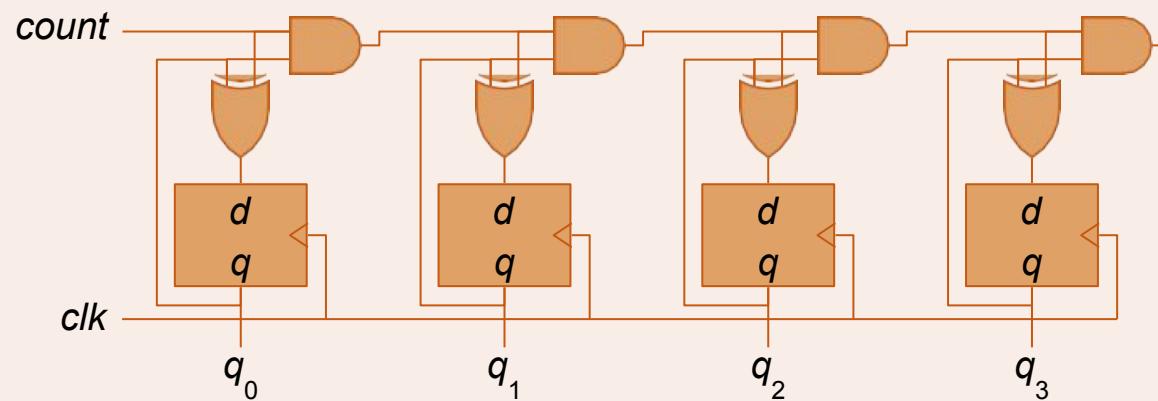
- Half adder logic circuit:



# COUNTERS

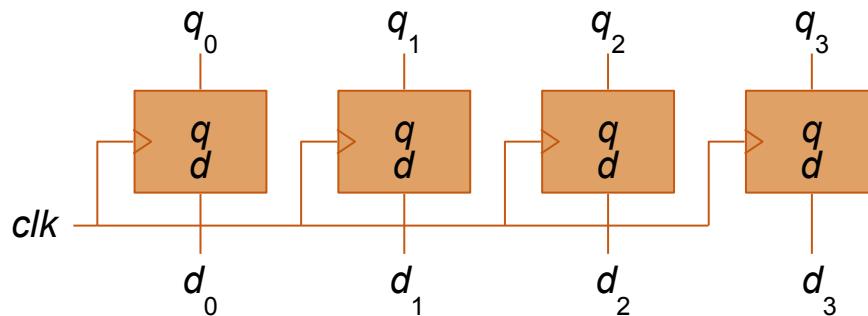
## Incrementing Counters

4-Bit Incrementing Counter



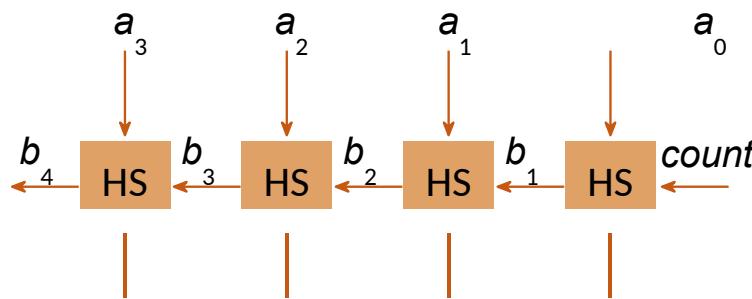
# Counters Recap –Decrementing counters

- An  $n$ -bit counter that counts back from  $2^n - 1$  to 0 (and back to  $2^n - 1$ )
  - .) Ex: a 2-bit counter, counts 11, 10, 01, 00, 11, 10, ....
- How to store  $n$ -bits?  $n$ -bit register

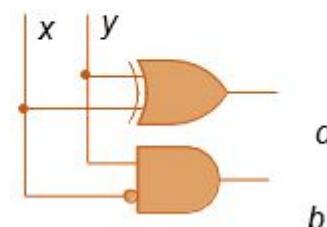


**Parking lot counter:** Starts at 20 and decrements by 1 for each car entering, displaying remaining slots until it reaches 0 → “Parking Full.”

- How to decrement an n-bit number? n-bit decrementer

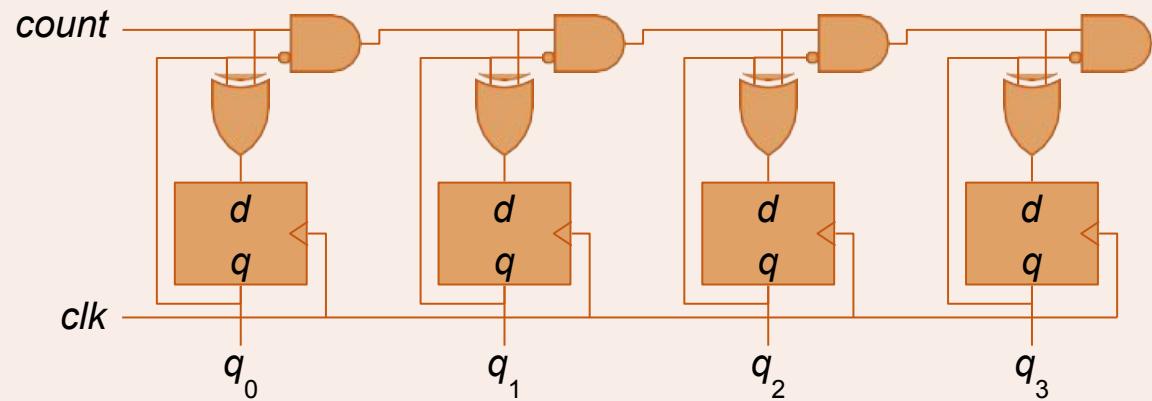


- Half Subtractor Logic Circuit:



## 4-bit Decrementing Counters

4-Bit Decrementing Counter



# Synchronous Counters

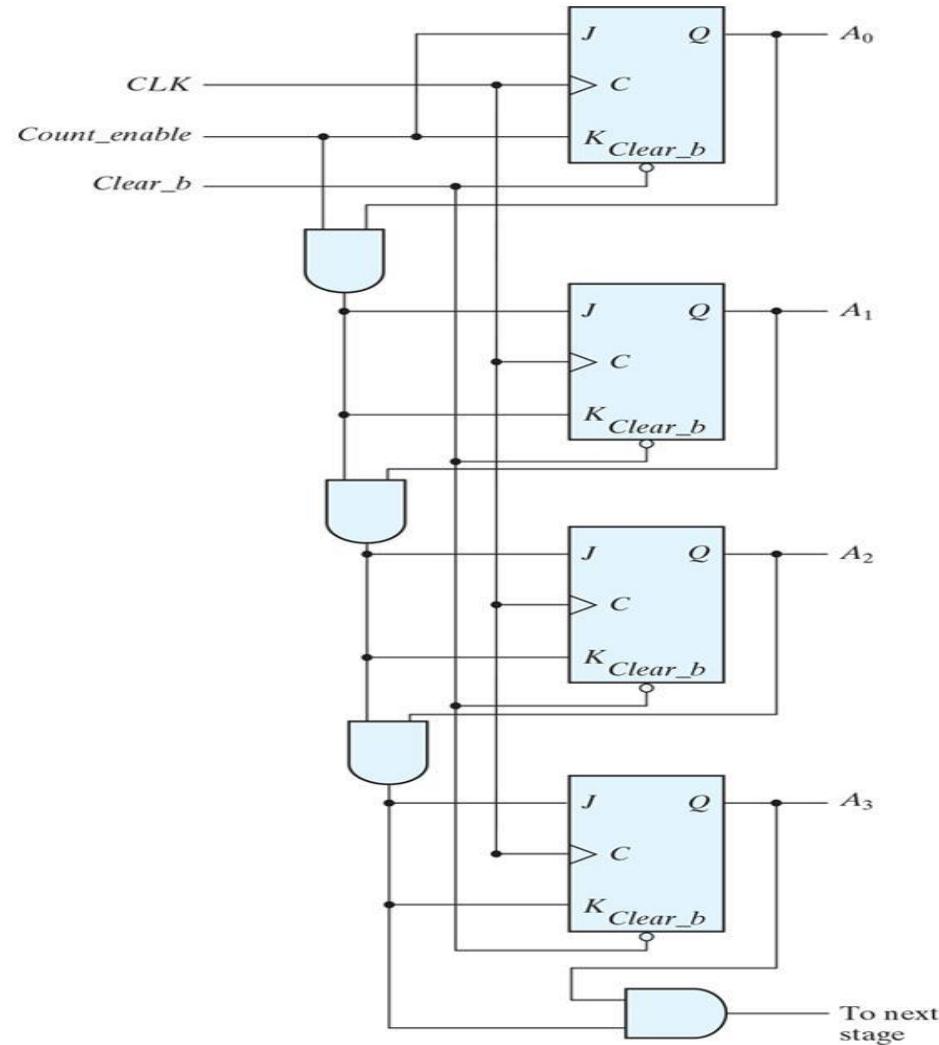
## Binary Counters

In a synchronous binary counter, **the flip-flop in the least significant position is complemented with every pulse. A flip-flop in any other position is complemented when all the bits in the lower significant positions are equal to 1**. For example, if the present state of a four-bit counter is  $A_3A_2A_1A_0 = 0011$ , the next count is 0100.  $A_0$  is always complemented.  $A_1$  is complemented because the present state of  $A_0 = 1$ .  $A_2$  is complemented because the present state of  $A_1A_0 = 11$ . However,  $A_3$  is not complemented, because the present state of  $A_2A_1A_0 = 011$ , which does not give an all-1's condition.

**The synchronous counter can be triggered with either the positive or the negative clock edge.**

# Synchronous Counters

**Figure 6.12**  
**Four-bit**  
**synchronous**  
**binary counter.**



# Synchronous Counters

---

The C inputs of all flip-flops are connected to a common clock. The counter is enabled by Count\_enable. **If the enable input is 0, all J and K inputs are equal to 0 and the clock does not change the state of the counter.** The first stage, A0, has its J and K equal to 1 if the counter is enabled. The other J and K inputs are equal to 1 if all previous least significant stages are equal to 1 and the count is enabled. The chain of AND gates generates the required logic for the J and K inputs in each stage. **The counter can be extended to any number of stages, with each stage having an additional flip-flop and an AND gate that gives an output of 1 if all previous flip-flop outputs are 1.**

# Synchronous Counters

---

Steps to design the counter using flipflops:

1. Decide the number of flipflop
2. Use excitation of the flipflop
3. State table and ckt excitation table
4. Obtain simplified equation using K map
5. Draw the sequential ckt

[https://www.tutorialspoint.com/digital-electronics/design\\_of\\_synchronous\\_counter.htm](https://www.tutorialspoint.com/digital-electronics/design_of_synchronous_counter.htm)

# Synchronous Counters

---

Design 2 bit up synchronous counter using JK flipflop:

Also called Mod 4 counter

# Synchronous Counters

Present (Q1 Q0)	Next (Q1 <sup>+</sup> Q0 <sup>+</sup> )
00	01
01	10
10	11
11	00

Excitation table

Q(n)	Q(n+1)	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

# Synchronous Counters

Present (Q1 Q0)	Next (Q1 <sup>+</sup> Q0 <sup>+</sup> )	inputs			
		J1	K1	J0	k0
00	01	0	X	1	X
01	10	1	x	x	1
10	11	x	0	1	X
11	00	x	1	x	1

**Excitation table**

Q(n)	Q(n+1)	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

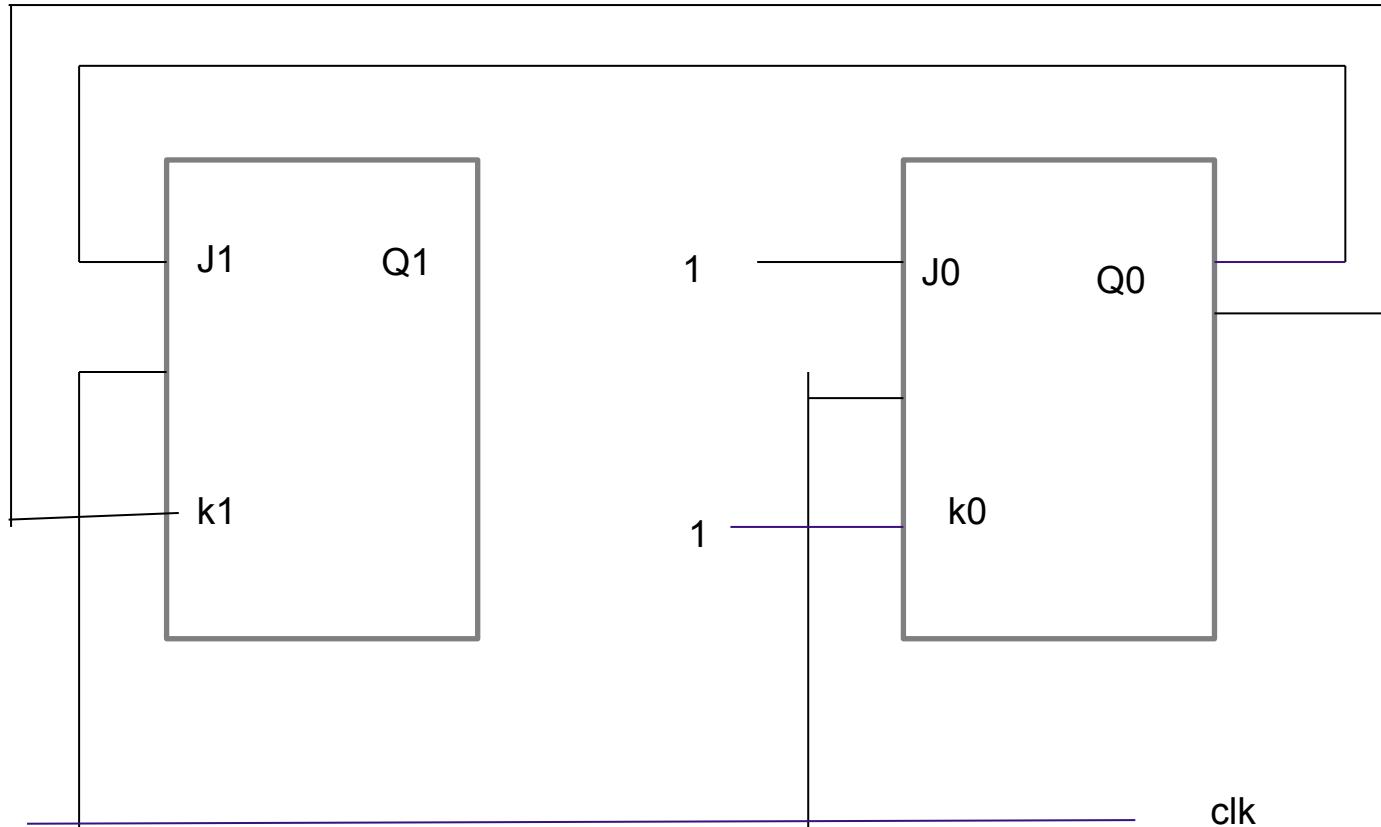
Solving using 2 variable K map of present state

$$J0 = 1, K0 = 1$$

$$J1 = Q0, K1 = Q0$$

# Synchronous Counters

$J_0 = 1, K_0 = 1$   
 $J_1 = Q_0, K_1 = Q_0$



# Synchronous Counters

---

Additional questions/assignment:

1. Design 2 bit synchronous down counter using JK flipflop
2. Design 3 bit synchronous down counter using T flipflop

# Synchronous Counters

---

## BCD Counter.

A BCD counter counts in binary-coded decimal from 0000 to 1001 and back to 0000. Because of the return to 0 after a count of 9, a BCD counter does not have a regular pattern, unlike a straight binary count. To derive the circuit of a BCD synchronous counter, it is necessary to go through a sequential circuit design procedure.

# BCD Counters o decade counter

Present State				Next State				Output	Flip-Flop Inputs			
$Q_8$	$Q_4$	$Q_2$	$Q_1$	$Q_8$	$Q_4$	$Q_2$	$Q_1$	$y$	$T_{Q8}$	$T_{Q4}$	$T_{Q2}$	$T_{Q1}$
0	0	0	0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	1	0	0	0	0	1	1
0	0	1	0	0	0	1	1	0	0	0	0	1
0	0	1	1	0	1	0	0	0	0	1	1	1
0	1	0	0	0	1	0	1	0	0	0	0	1
0	1	0	1	0	1	1	0	0	0	0	1	1
0	1	1	0	0	1	1	1	0	0	0	0	1
0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	0	0	1	0	0	0	0	1
1	0	0	1	0	0	0	0	1	1	0	0	1

**Table 6.5**  
**State Table for BCD Counter.**

$$T_{Q1} = 1$$

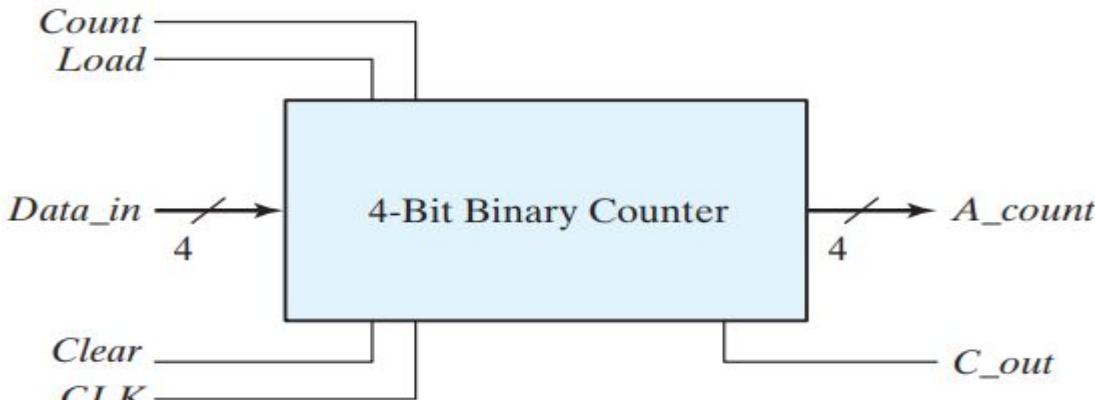
$$T_{Q2} = Q'_8 Q_1$$

$$T_{Q4} = Q_2 Q_1$$

$$T_{Q8} = Q_8 Q_1 + Q_4 Q_2 Q_1$$

$$y = Q_8 Q_1$$

# Binary Counter with Parallel Load



(a)

**Count/Load:** A control input that selects whether the counter should count or load data. If Count is active, the counter increments based on the clock. If Load is active, a 4-bit value is loaded into the counter.

**Data\_in (4 lines):** This provides the data to be loaded into the 4-bit counter when the Load signal is active.

**A\_count (4 lines):** The 4-bit output that holds the current count of the counter.

**Clear:** Resets the counter to zero.

**CLK:** The clock signal that controls the counting operation.

**C\_out:** A carry-out signal that goes high when the counter overflows (from 15 back to 0).

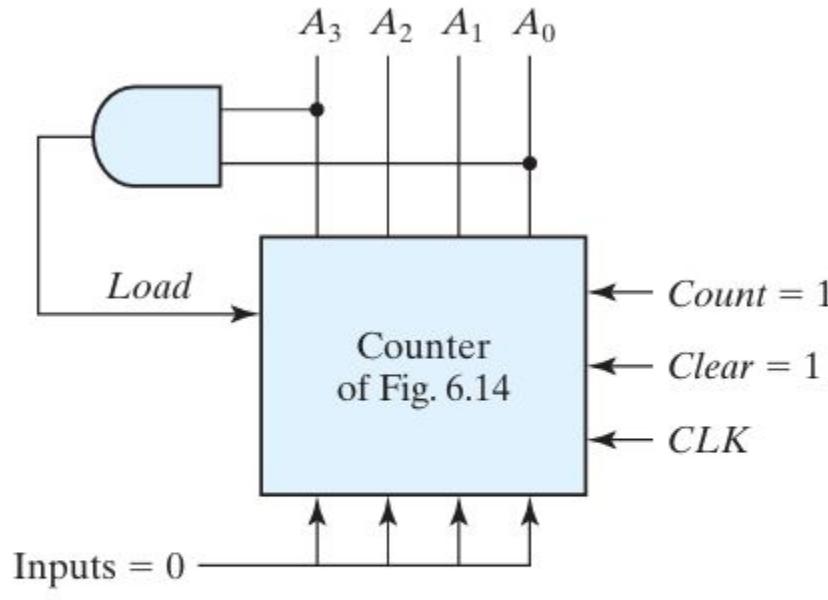
# Binary Counter with Parallel Load

---

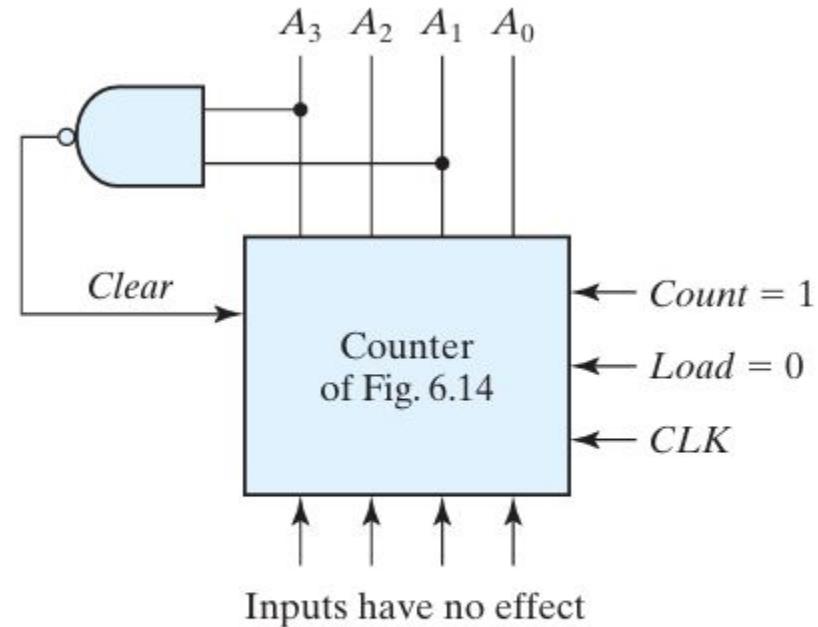
**Table 6.6**  
**Function Table for the Counter of Fig. 6.14.**

<b>Clear_b</b>	<b>CLK</b>	<b>Load</b>	<b>Count</b>	<b>Function</b>
0	X	X	X	Clear to 0
1	↑	1	X	Load inputs
1	↑	0	1	Count next binary state
1	↑	0	0	No change

# Binary Counter with Parallel Load



(a) Using the load input



(b) Using the clear input

**FIGURE 6.15**

Two ways to achieve a BCD counter using a counter with parallel load

# Synchronous Counters application

Example: Image Processing with Synchronous Counters

- ◆ **Scenario**

Suppose we have a grayscale image of  $256 \times 256$  pixels stored in memory.

Each pixel = 1 byte, and all pixels are stored sequentially in RAM.

Pixel(0,0) → Address 0000h

Pixel(0,1) → Address 0001h...

Pixel(255,255) → Address FFFFh

## How Synchronous Counter Helps?

A 16-bit synchronous counter acts as the address generator.

On each clock pulse, the counter increments → next memory address is produced instantly.

The memory outputs the pixel value at that address.

## Why Synchronous Counter?

All flip-flops update together → no glitching on the address bus.

Essential for high-speed video/image systems (e.g., cameras, medical imaging, graphics cards).

A MOD-10 synchronous counter is implemented. What will be the count sequence of the counter?

- (A) 0000 to 1111
- (B) 0000 to 1001
- (C) 0001 to 1110
- (D) 0000 to 1010

A synchronous counter is designed to count in the sequence: 000 → 010 → 011 → 101 → 110 → 000. What is the modulus of this counter?

- (A) 3
- (B) 5
- (C) 6
- (D) 7

A MOD-10 synchronous counter is implemented. What will be the count sequence of the counter?

- (A) 0000 to 1111
- (B) 0000 to 1001
- (C) 0001 to 1110
- (D) 0000 to 1010

Answer: (B) 0000 to 1001

A synchronous counter is designed to count in the sequence: 000 → 010 → 011 → 101 → 110 → 000. What is the modulus of this counter?

- (A) 3
- (B) 5
- (C) 6
- (D) 7

Answer: (B) 5, The modulus (MOD-N) of a counter = number of unique states it cycles through before repeating.



Pearson



**PES**  
UNIVERSITY

CELEBRATING 50 YEARS

**THANK YOU**

---

**Team DDCO  
Department of Computer Science and Engineering**



**PES**

UNIVERSITY

CELEBRATING 50 YEARS

# DIGITAL DESIGN AND COMPUTER ORGANIZATION

## Other counters

---

**Team DDCO**

**Department of Computer Science and Engineering**

# Other Counters(T1 section 6.5)

---

Counters can be designed to generate any desired sequence of states. A divide-by- N counter (also known as a modulo- N counter) is a counter that goes through a repeated sequence of N states.

A circuit with n flip-flops has  $2^n$  binary states. There are occasions when a sequential circuit uses fewer than this maximum possible number of states. **States that are not used in specifying the sequential circuit are not listed in the state table. In simplifying the input equations, the unused states may be treated as don't-care conditions or may be assigned specific next states.**

it is necessary to ensure that the circuit eventually goes into one of the valid states so that it can resume normal operation. Otherwise, if the sequential circuit circulates among unused states, there will be no way to bring it back to its intended sequence of state transitions

# Counter with Unused States

---

A circuit with  $n$  flip-flops has  $2^n$  binary states. There are occasions when a sequential circuit uses fewer than this maximum possible number of states. States that are not used in sequential circuit.

The count has a repeated sequence of six states, with flip-flops B and C repeating the binary count 00, 01, 10, and flip-flop A alternating between 0 and 1 every three counts.

# Other Counters with Unused States

---

**Table 6.7**  
**State Table for Counter.**

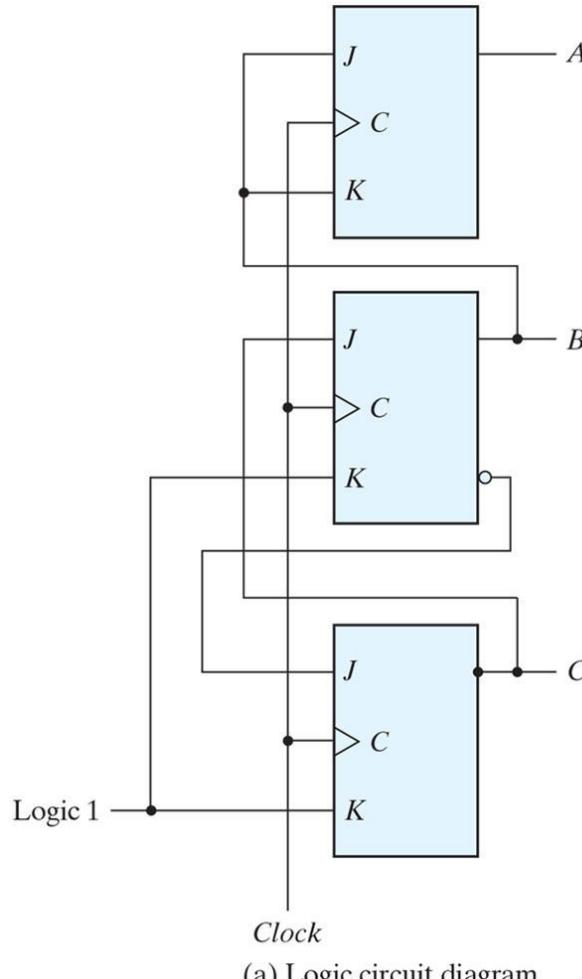
$$\begin{array}{ll}
 J_A = B & K_A = B \\
 J_B = C & K_B = 1 \\
 J_C = B' & K_C = 1
 \end{array}$$

Present State			Next State			Flip-Flop Inputs					
A	B	C	A	B	C	J <sub>A</sub>	K <sub>A</sub>	J <sub>B</sub>	K <sub>B</sub>	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

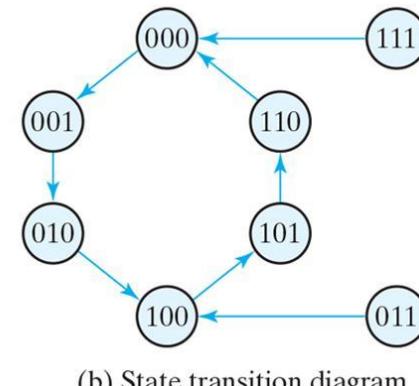
Unused states= 011 and 111

# Other Counters with Unused States

**Figure 6.16**  
**Counter with**  
**unused states.**



Thus, the counter is **self-correcting**. In a self-correcting counter, if the counter happens to be in one of the unused states, it eventually reaches the normal count sequence after one or more clock pulses



# Ring Counters

---

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

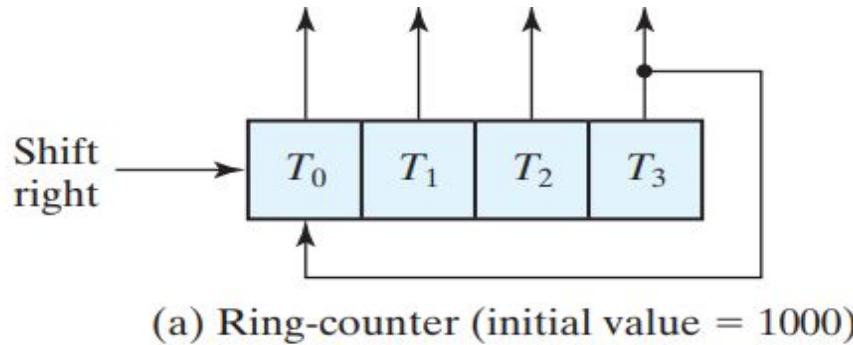
The initial value of the register is 1000 and requires Preset/Clear flip-flops.

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.

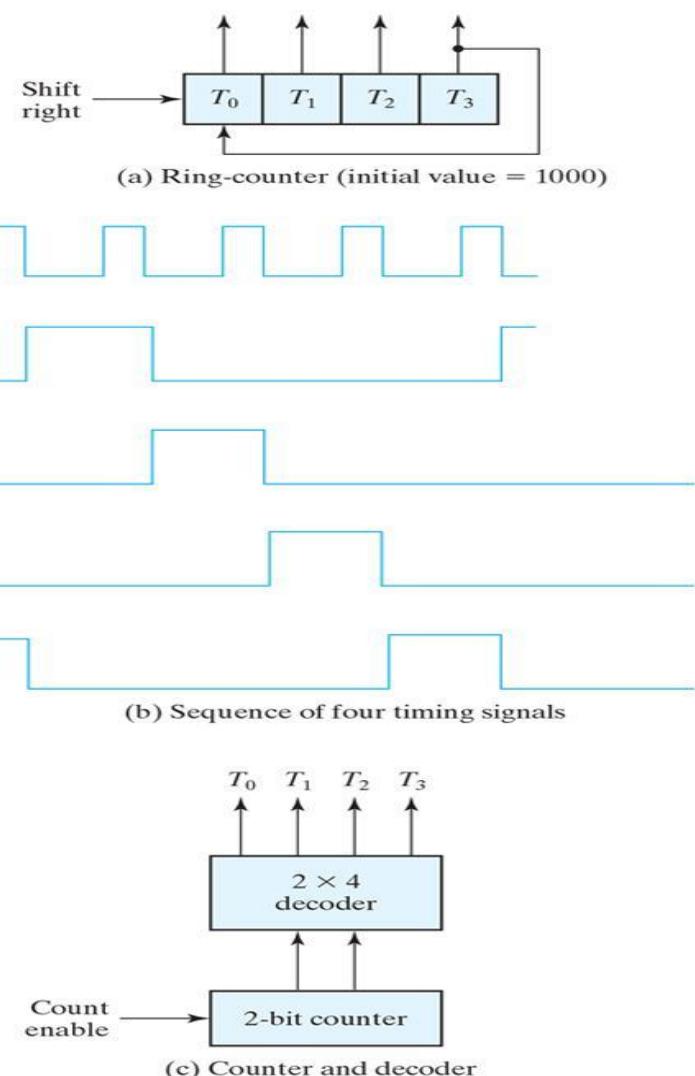
# Ring Counters

- Single bit shifted right with every clock pulse and circulates back from T3 to T0. Each flip-flop is in the 1 state once every four clock cycles



- It is negative edge triggered flipflop state changes on negative edge
- Initial value is always 1000

# Ring Counters



**Figure 6.17**  
**Generation of timing signals.**

# Ring Counters

---

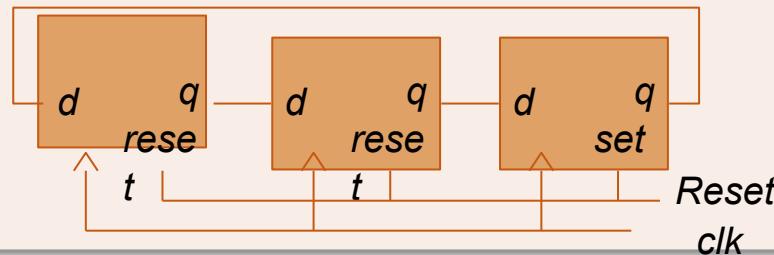
To generate  $2^n$  timing signals, we need either a shift register with  $2^n$  flip-flops or an  $n$ -bit binary counter together with an  $n$ -to- $2^n$ -line decoder.

For example, 16 timing signals can be generated with a 16-bit shift register connected as a ring counter or with a 4-bit binary counter and a 4-to-16-line decoder. In the first case, we need 16 flip-flops. In the second, we need 4 flip-flops and 16 four-input AND gates for the decoder. It is also possible to generate the timing signals with a combination of a shift register and a decoder. That way, the number of flip-flops is less than that in a ring counter, and the decoder requires only two-input gates. This combination is called a Johnson counter .

# Ring Counters –Summary

- Counters constructed by connecting together flip-flops in closed loop, typically with a single 1 bit circulating among them, are called **ring counters**
- Modulus of an  $n$ -bit ring counter is  $n$

## 3-bit Ring Counter

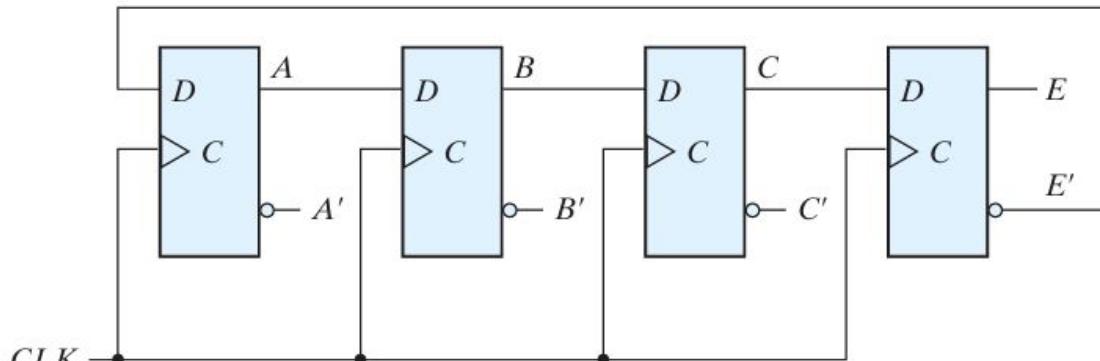


- Also called “one-hot” counters (as in one-hot encoding)
- Number of states= number of flipflop, mod-8 ring counter requires eight flip-flops and a mod-16 ring counter would require sixteen flip-flops.(number of states=number of flipflops) but Other flipflop number of states is  $=2^n$

# Johnson Counter

---

- A  $k$ -bit ring counter circulates a single bit among the flip-flops to provide  $k$  distinguishable states. The number of states can be doubled if the shift register is connected as a **switch-tail ring counter**.
- 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.



(a) Four-stage switch-tail ring counter

# Johnson counter

---

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	$C'E$

(b) Count sequence and required decoding

# Johnson counter

---

The circular connection is made from the complemented output of the rightmost flip-flop to the input of the leftmost flip-flop.

The register shifts its contents once to the right with every clock pulse, and at the same time, the complemented value of the E flip-flop is transferred into the A flip-flop.

In general, a  $k$ -bit switch-tail ring counter will go through a sequence of **2  $k$  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 Johnson counter is a  $k$ -bit switch-tail ring counter with  $2 k$  decoding gates to provide outputs for  $2 k$  timing signals.

The eight AND gates listed in the table, when connected to the circuit, will complete the construction of the Johnson counter.

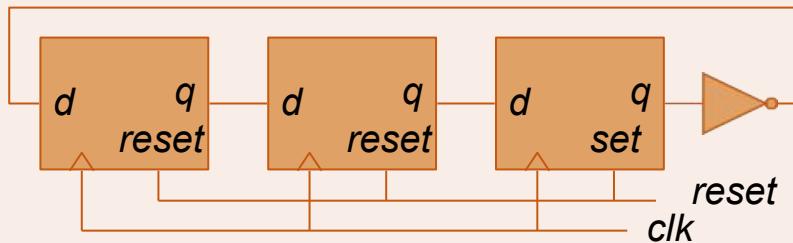
# Johnson Counter

---

- One disadvantage of the circuit in Fig. 6.18 (a) is that if it finds itself in an unused state, it will persist in moving from one invalid state to another and never find its way to a valid state. The difficulty can be corrected by modifying the circuit to avoid this undesirable condition
- One correcting procedure is to disconnect the output from flip-flop B that goes to the D input of flip-flop C and instead enable the input of flip-flop C by the function
- $D_C = (A + C)B$  where  $D_C$  is the flip-flop input equation for the D input of flip-flop C. Johnson counters can be constructed for any number of timing sequences. The number of flip-flops needed is one-half the number of timing signals.
- The number of decoding gates is equal to the number of timing signals, and only two-input gates are needed.

# Johnson counter using D flipflop(negative edge triggered)-summary

3-bit Johnson (Twisted Ring) Counter (switch tail counter)



- What is the modulus of an *n*-bit Johnson counter?

The MOD of the Johnson counter is  $2n$  if  $n$  flip-flops are used. The main advantage of the Johnson counter is that it only needs half the number of flip-flops compared to the standard **ring counter** for the same MOD. (  $4ff=8$  states( $2n$ ))

# Ring Counter applications

---

- **Application: Instruction Pipeline Controller in Microprocessors**
- In certain simple microcontrollers and DSP cores, a ring counter is used as the control unit for the pipeline stages (Fetch → Decode → Execute → Memory → Write-back).
- Why? A ring counter naturally cycles one “hot” bit through its outputs, so each stage gets activated in perfect round-robin order.
- This avoids the need for complex decoders and guarantees non-overlapping control signals, which is critical for synchronous CPU timing.
- *Practical use:* Intel’s early pipeline controllers and some ARM/DSP cores have used ring counters in control sequencing because they are simple, reliable, and glitch-free.

# Johnson Counter applications

---

## Johnson Counter in LED Chaser / Marquee Display

- A Johnson counter (e.g., CD4017 IC) generates 10 sequential outputs (Q0–Q9).
- Each clock pulse activates one output at a time
- Connect 10 LEDs to Q0–Q9. On each clock tick:
- Q0 HIGH → LED1
- ON Next tick → Q1 HIGH → LED2 ON... continues until Q9, then loops back to Q0.
- LEDs light up one after another in a running sequence → “chaser” or “marquee” effect.

## Applications:

- Advertising signboards

## Why Johnson Counter?

- Only 5 flip-flops → 10 outputs
- No extra decoder needed

# MCQ

---

- A **4-bit synchronous counter with unused states** is designed to count only 6 states. If the counter accidentally enters an unused state, it eventually returns to the valid sequence. Such a counter is called:

- (A) Non-deterministic counter
- (B) Self-correcting counter
- (C) Johnson counter
- (D) Binary coded decimal counter

A 4-bit **switch-tail ring counter** (Johnson counter) starts from all 0's. Which of the following will be the **fifth state**?

- (A) 1110
- (B) 1111
- (C) 0111
- (D) 1000

# MCQ

- A **4-bit synchronous counter with unused states** is designed to count only 6 states. If the counter accidentally enters an unused state, it eventually returns to the valid sequence. Such a counter is called:

- (A) Non-deterministic counter
- (B) Self-correcting counter
- (C) Johnson counter
- (D) Binary coded decimal counter

**Answer:** (B) Self-correcting counter

**Explanation:** Self-correcting counters ensure that invalid states eventually lead back into the normal cycle.

- A 4-bit **switch-tail ring counter** (Johnson counter) starts from all 0's. Which of the following will be the **fifth state**?

- (A) 1110
- (B) 1111
- (C) 0111
- (D) 1000

**Answer:** (B) 1111

**Explanation:** Sequence: 0000 → 1000 → 1100 → 1110 → 1111 → ... The 5th state = 1111.

# MCQ

---

- A 3-bit synchronous counter is implemented using T-flip-flops and its present state is 101. What will be the next state?  
A) 110  
B) 011  
C) 010  
D) 001

# MCQ

---

- A 3-bit synchronous counter is implemented using T-flip-flops and its present state is 101. What will be the next state?
- A) 110  
B) 011  
C) 010  
D) 001
- **Answer:** (A) 110  
**Explanation:** In a synchronous up counter using T-FFs, each output toggles if its T input is 1. For a typical design, all T inputs = 1, so the next state from 101 (5) is 110 (6).

$T_0 = 1$  (LSB toggles on every clock).

$T_1 = Q_0$  (next bit toggles when  $Q_0 = 1$ ).

$T_2 = Q_0 \cdot Q_1$  (next bit toggles when  $Q_0$  and  $Q_1 = 1$ ).

In general,  $T_k = Q_0 \cdot Q_1 \cdot \dots \cdot Q_{k-1} \cdot T_k = Q_0 \cdot Q_1 \cdot \dots \cdot Q_{\{k-1\}}$



**PES**  
UNIVERSITY

CELEBRATING 50 YEARS

**THANK YOU**

---

**Team DDCO  
Department of Computer Science and Engineering**

# PES UNIVERSITY, Bangalore

Established under Karnataka Act No. 16 of 2013)

## Department of Computer Science & Engineering

### UE24CS251A: DIGITAL DESIGN AND COMPUTER ORGANIZATION

Unit 1:

- Using Boolean laws, prove that the expression  $F = AB + A'C + BC$  can be simplified to  $F = AB + A'C$ .

*Explain the rationale behind removing the redundant term and the Boolean theorem used.*

Ans:

To simplify:

$$F = AB + A'C + BC$$

Apply Consensus Theorem:

$$\text{Consensus theorem: } AB + A'C + BC = AB + A'C$$

Hence:

$$F = AB + A'C + BC \rightarrow F = AB + A'C$$

- A student proposes a circuit for  $F = AB + A'C$  but implements it as  $(A + B) \cdot (A' + C)$ . Is the implementation logically correct? Justify your answer with a truth table comparison and Boolean reasoning.

Ans:

A	B	C	$AB + A'C$	$(A + B)(A' + C)$
0	0	0	0	0
0	0	1	1	1
0	1	0	0	1 <span style="color: red;">X</span>
0	1	1	1	1
1	0	0	0	1 <span style="color: red;">X</span>
1	0	1	0	1 <span style="color: red;">X</span>
1	1	0	1	1
1	1	1	1	1

Conclusion:

- $F_1 \neq F_2$  (values differ in some cases)
- Student's implementation is incorrect.
- Mistake: Student applied incorrect transformation. (Distributive Law misapplied)

3. You are given a Boolean function implemented in 3 logic levels as  
 $F=AB+C(D+E)$   
 $F = AB + C(D + E)$   
 $F=AB+C(D+E)$
- Why is this not a standard SOP or POS form?
  - Convert this expression into a canonical SOP form.
  - Discuss the impact of logic levels on gate delay and suggest why a two-level form is preferred in digital design.

Ans:

- The function contains nested parentheses and a mix of AND and OR across different layers.  
 Hence, it's not in standard SOP or POS format (which are flat, two-level expressions).
  - Apply distributive law:  $F=AB+C(D+E)=AB+CD+CEF = AB + C(D + E) = AB + CD +$   
 $CEF=AB+C(D+E)=AB+CD+CE$ . This is now in standard SOP form.
  - More logic levels → more delay, as each level adds gate propagation delay. Two-level logic (as in SOP or POS) is preferred for faster computation in combinational circuits.
- 4) In digital logic design, it is often desirable to minimize Boolean functions.
- What is logic minimization, and why is it important in hardware design?
  - What metric is commonly used to measure the efficiency of a minimized Boolean function?
  - Explain how canonical forms help in the process of logic minimization, and state one limitation of using them directly for circuit implementation.

Ans:

- Logic Minimization:**  
 Logic minimization is the process of reducing a Boolean function to its simplest equivalent form without changing its output behavior. This helps to use fewer gates, reduce power consumption, and increase speed in hardware implementations.
- Efficiency Metric:**  
 One commonly used metric is the smallest two-level Sum of Products (SOP) form in terms of:
  - Number of literals
  - Number of gates
  - Depth (levels) of logic
- Role of Canonical Forms & Limitation:**
  - Canonical forms (like sum of minterms or product of maxterms) provide a starting point for minimization because they represent the function exactly and completely.
  - However, they are not minimal — they include all variables in every term, leading to larger and slower circuits if implemented directly.

5. Design a 3-variable logic circuit whose output is 1 for minterms 1, 3, 5, and 6. Draw the K-map, minimize the function, and draw the logic circuit diagram using basic gates.

Ans:

A\BC	00	01	11	10
0	0	1	1	0
1	0	1	0	1

$$f(A,B,C) = A'C + AB'C + ABC'$$

6. Design a 4-variable logic circuit whose output is 1 for minterms 0, 1, 2, 5, 7, 8, 10. The output is don't care (X) for minterms 3 and 11.

- a) Draw the 4-variable Karnaugh Map.
- b) Minimize the function using SOP form (use don't-cares to simplify).
- c) Minimize the same function using POS form (also using don't-cares).
- d) Compare the SOP and POS expressions in terms of gate complexity.
- e) Draw the logic circuit diagram for the simplified SOP expression using basic gates

Ans:

wx\yz	00	01	11	10
00	1	1	X	1
01	0	1	1	0
11	X	0	-	-
10	1	0	1	0

$$f(w,x,y,z) = w'x' + w'xz + wx'z$$

$$f = (w' + x')(w' + z)(x' + y) - \text{de morgan's law}$$

7. Implement the Boolean function  $F = AB + CD$  using:

- (a) AND-OR logic (standard SOP)
- (b) Only NAND gates

Draw both circuits and explain the conversion process from SOP to NAND implementation.

Ans:

Logic Type      Gates Used      Final Output Expression

AND-OR (SOP) 2 AND, 1 OR     $F = AB + CD$

NAND-Only    3 NAND gates  $F = (A \text{ NAND } B)' \text{ NAND } (C \text{ NAND } D)'$

8. Describe the steps involved in converting a Boolean function implemented using AND-OR logic to an implementation using only NOR gates.

Using this method, convert and implement  $F = (A + B)(C + D)$  using NOR gates only.

Ans:

1. Write the expression using AND and OR operations.

2. Apply double negation:

$$F = (A + B)(C + D) = \overline{\overline{(A + B)}(C + D)}$$

3. Use DeMorgan's Theorem to push negation inside:

$$\overline{(A + B)(C + D)} = \overline{A + B} + \overline{C + D}$$

4. Again apply negation to entire expression to restore the original:

$$F = \overline{\overline{A + B} + \overline{C + D}}$$

5. Now this form:

$$F = \overline{(A + B)} + \overline{(C + D)}$$

is suitable for NOR-only implementation.

Logic Type	Original Expression	NOR Equivalent	Gates Used
AND-OR	$F = (A + B)(C + D)$	$F' = [(A+B)(C+D)]'$	3 NOR gates

9. Design a combinational circuit that converts a 4-bit BCD (8421) input into Excess-3 code. Your design should include:

- (a) A complete truth table (consider valid BCD inputs only)
- (b) Boolean expressions for each output bit
- (c) Simplification using Karnaugh Maps (K-maps)
- (d) Final logic diagram using basic gates (AND, OR, NOT)

Ans:

BCD is a 4-bit binary code representing digits 0 to 9.

Excess-3 code = BCD + 0011 (i.e., add 3 to the decimal value).

We ignore BCD inputs from 1010 to 1111 (treated as don't care in logic design).

Decimal BCD (A B C D) Excess-3 (E3 E2 E1 E0)

0	0000	0011
1	0001	0100
2	0010	0101

Decimal BCD (A B C D) Excess-3 (E3 E2 E1 E0)

3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100

(b) Boolean Expressions (Unminimized)

Let inputs be A, B, C, D (MSB to LSB of BCD)

Let outputs be W, X, Y, Z (MSB to LSB of Excess-3)

We'll find expressions for:

- W = E3
- X = E2
- Y = E1
- Z = E0

We'll derive these from the truth table using K-maps in the next step.

A B \ C D 00 01 11 10

00	0	0	0	0
01	0	1	1	1
10	1	1	X	X
11	X	X	X	X

$$W = A + (B \cdot C)$$

$$X = B \oplus C$$

$$Y = (C \oplus D)'$$

$$Z = D'$$

Logic Circuit Diagram

Inputs: A, B, C, D

Outputs: W, X, Y, Z

Gate-level Expressions to implement:

- $W = A + B \cdot C \rightarrow$  Use one AND gate and one OR gate
- $X = B \oplus C \rightarrow$  Use one XOR gate
- $Y = (C \oplus D)' \rightarrow$  Use one XOR gate and a NOT gate
- $Z = D' \rightarrow$  Use one NOT gate

Gate List:

- 1 AND gate (for  $B \cdot C$ )
- 2 XOR gates (for  $X$  and part of  $Y$ )
- 2 NOT gates (for  $Z$  and  $Y$ )
- 1 OR gate (for  $W$ )

10. A combinational circuit has the following gate-level structure:

- Inputs:  $A, B, C$
- Gate 1: Output  $X = A \oplus B$
- Gate 2: Output  $Y = X \cdot C$
- Gate 3: Output  $Z = Y + A'$

Perform the following:

- (a) Derive the Boolean expression for the final output  $Z$
- (b) Simplify the expression using Boolean algebra
- (c) Draw the logic circuit diagram corresponding to the simplified expression

Ans:

Derive the Boolean Expression for  $Z$

We're asked to compute  $Z$  step by step.

Given:

- $X = A \oplus B$
- $Y = X \cdot C = (A \oplus B) \cdot C$
- $Z = Y + A' = ((A \oplus B) \cdot C) + A'$

So the expression for  $Z$  is:

$$Z = ((A \oplus B) \cdot C) + A'$$

Final Simplified Boolean Expression:

$$Z = A' + AB'C$$

Logic Circuit Diagram for  $Z = A' + AB'C$

Required Gates:

- 1 NOT gate: for  $A'$
- 1 AND gate: for  $B' \cdot C$
- 1 NOT gate: for  $B'$
- 1 AND gate: for  $A \cdot B' \cdot C$
- 1 OR gate: for  $A' + AB'C$

Circuit Construction Steps:

1. NOT Gate 1: Input A → Output  $A'$
2. NOT Gate 2: Input B → Output  $B'$
3. AND Gate 1: Inputs  $B'$  and C → Output =  $B' \cdot C$
4. AND Gate 2: Inputs A and  $B' \cdot C$  → Output =  $A \cdot B' \cdot C$
5. OR Gate: Inputs  $A'$  and  $A \cdot B' \cdot C$  → Output Z

This gives you the minimal circuit diagram using 2 NOT gates, 2 AND gates, and 1 OR gate.

11. Design a binary adder-subtractor circuit that uses a common logic to perform both operations.

- Explain how the 2's complement is used in subtraction.
- Show circuit block with controlled inverter using XOR.

Ans:

Subtraction = A + 2's complement of B = A + ( $B' + 1$ )

XOR gate used to invert B based on Add/Sub control signal

12. How is overflow detected in binary addition for both signed and unsigned numbers?

- Provide the condition and example for each case.

Ans:

Unsigned: Overflow if final carry-out from MSB is 1

Signed: Overflow if carry into MSB ≠ carry out of MSB → Overflow =  $C_n \oplus C_{n-1}$

13. What is the need for a BCD adder instead of a binary adder for decimal digit operations?

- Describe the correction logic required when binary sum exceeds 9.
- Derive the correction condition using Boolean logic.

Ans: Binary results >1001 are invalid BCD

Add 0110 (6) if sum > 1001 or carry-out = 1

Correction logic:  $C = K + Z_8 \cdot Z_4 + Z_8 \cdot Z_2$

14. Draw the block diagram of a BCD adder that adds two BCD digits with a carry-in.

- Explain the role of upper and lower 4-bit adders and how the correction is applied.

Ans:

### 1. Upper 4-bit Adder

- Adds:
  - First BCD digit (A3 A2 A1 A0)
  - Second BCD digit (B3 B2 B1 B0)
  - Carry-in (C\_in)
- Produces:
  - Binary sum of 4 bits
  - Carry-out (K) → High if the sum exceeds 15

### 2. Correction Logic

BCD digits range from 0000 to 1001 (i.e., 0 to 9).

So if the binary sum > 1001 (decimal 9) or the carry-out K = 1, it must be corrected.

- Correction condition:

$$C = K + Z_8 \cdot Z_4 + Z_8 \cdot Z_2$$

### Lower 4-bit Adder (Correction Adder)

- Adds binary sum + 0110 (6) only if correction is needed
- Output is the corrected BCD digit
- Carry-out from this stage becomes the carry-in for the next BCD digit position (next stage in multi-digit BCD adders)

15. Explain how a 2-bit binary multiplier works. Draw the circuit and explain the role of AND gates and adders in producing the final product.

Ans: A 2-bit binary multiplier multiplies two binary numbers, say A = A1 A0 and B = B1 B0. The product will have 4 bits: C3 C2 C1 C0.

#### 1. Partial Products:

- Multiply each bit of B with each bit of A using AND gates:
  - PPO = A0 AND B0 → C0
  - PP1 = A0 AND B1, A1 AND B0 → added to form C1
  - PP2 = A1 AND B1 → part of C2 or C3

#### 2. Circuit Components:

- 4 AND gates
- 2 Half Adders or 1 Full Adder depending on implementation.

3. Final Output:

- C0 is directly from the first AND gate.
- C1, C2 are formed using adders on partial products.
- C3 may result from carry.

This structure mimics how we multiply decimal numbers manually—bitwise AND acts like multiplication, and adders handle partial sums.

16. Construct the truth table for a 2-bit magnitude comparator and explain how the circuit distinguishes between A > B, A = B, and A < B conditions.

Ans:

A = A1 A0 and B = B1 B0 be two 2-bit binary numbers.

The comparator outputs are:

- $(A > B) = 1$  if A is greater than B,
- $(A = B) = 1$  if A is equal to B,
- $(A < B) = 1$  if A is less than B.

A1 A0 B1 B0 A > B A = B A < B

0 0 0 0 0 1 0

0 0 0 1 0 0 1

0 0 1 0 0 0 1

0 0 1 1 0 0 1

0 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 1

1 0 0 0 1 0 0

1 0 0 1 1 0 0

1 0 1 0 0 1 0

1 0 1 1 0 0 1

A1 A0 B1 B0 A > B A = B A < B

1 1 0 0 1 0 0

1 1 0 1 1 0 0

1 1 1 0 1 0 0

1 1 1 1 0 1 0

Equality Condition (A = B)

- Use XNOR gates on A1-B1 and A0-B0:
  - $x_1 = A_1 \odot B_1$
  - $x_0 = A_0 \odot B_0$
- Then  $(A = B) = x_1 \cdot x_0$

Greater Than (A > B)

- First compare most significant bits (A1 and B1):
  - If  $A_1 = 1$  and  $B_1 = 0 \rightarrow A > B$
  - If  $A_1 = B_1$ , check A0 vs B0

$$(A > B) = A_1 \cdot B_1' + (A_1 \odot B_1) \cdot A_0 \cdot B_0'$$

Less Than (A < B)

$$(A < B) = A_1' \cdot B_1 + (A_1 \odot B_1) \cdot A_0' \cdot B_0$$

This logic enables a comparator to decide which input is larger or smaller by analyzing from the most significant bit (MSB) down to the least significant bit (LSB), only moving forward when bits are equal.

17. Design a 3-to-8 decoder using only 2-to-4 decoders and logic gates. Explain the logic behind the construction.

Ans:

A 3-to-8 decoder has:

- Inputs: A (MSB), B, C (LSB)
- Outputs: D<sub>0</sub> to D<sub>7</sub> (only one HIGH at a time)

Each output corresponds to one unique combination of inputs A, B, and C.

To construct this using 2-to-4 decoders, observe:

- A 2-to-4 decoder can decode 2 inputs into 4 outputs
- We need to use A as a control signal to enable one decoder at a time
  - When A = 0: lower 4 outputs (D<sub>0</sub>-D<sub>3</sub>) are active
  - When A = 1: upper 4 outputs (D<sub>4</sub>-D<sub>7</sub>) are active

So we use:

- One 2-to-4 decoder to decode inputs B and C when A = 0
- Another 2-to-4 decoder to decode B and C when A = 1

Enable signals are driven by A and A' (NOT A)

A B C Active Decoder Output

0 0 0 Decoder 1       $D_0 = 1$

0 0 1 Decoder 1       $D_1 = 1$

0 1 0 Decoder 1       $D_2 = 1$

0 1 1 Decoder 1       $D_3 = 1$

1 0 0 Decoder 2       $D_4 = 1$

1 0 1 Decoder 2       $D_5 = 1$

1 1 0 Decoder 2       $D_6 = 1$

1 1 1 Decoder 2       $D_7 = 1$

Two 2x4 decoders are used.

A NOT gate generates A' to control one decoder when A = 0.

A directly enables the second decoder when A = 1.

B and C are shared between both decoders.

Outputs are selected using the decoder enabled by A/A'.

18. Explain how complemented minterm expressions can help reduce hardware complexity when implementing functions using a decoder.

- Provide an example where this strategy reduces the number of OR gate inputs.

Ans:

In digital logic design, any Boolean function can be implemented using a decoder and OR gates by selecting the minterms corresponding to the 1s in the truth table.

However, if a function has many 1s (minterms), it may require:

- A decoder with many outputs to be combined using an OR gate with many inputs, which increases hardware complexity.

In such cases, it is more efficient to implement the complement of the function and then invert the result.

This is known as the dual or complemented minterm strategy.

If a function FFF has more than  $2n/2^{2^n} / 2^{2n}/2$  minterms (i.e., more 1s than 0s), it is efficient to:

- Implement  $F'$  ( $F$  complement) using fewer minterms
- Then invert it using a NOT gate (or NOR instead of OR)

This reduces:

- The number of decoder output lines used
- The fan-in (number of inputs) of the OR gate

Using complemented minterm expressions:

- Reduces the number of decoder outputs that need to be connected
- Minimizes the size of OR gates
- Simplifies circuit layout and saves hardware resources

It is a powerful technique especially when number of 1s > number of 0s in the truth table.

19. Design a 4-to-1 multiplexer using only 2-to-1 multiplexers. Show the logic circuit and explain the design.

Ans:

S1 S0 Output (Y)

0 0 D0

0 1 D1

1 0 D2

1 1 D3

$$Y_0 = S_0' \cdot D_0 + S_0 \cdot D_1$$

$$Y_1 = S_0' \cdot D_2 + S_0 \cdot D_3$$

$$Y = S_1' \cdot Y_0 + S_1 \cdot Y_1$$

20. Compare the implementation of Boolean functions using multiplexers vs. decoders. Highlight the trade-offs involved.

Ans:

Feature	Multiplexer (MUX)	Decoder
Selection Logic	Uses select lines to choose one data input	Uses select lines to activate one output line (minterm)
Function Implementation	Implemented by mapping minterms to data inputs ( $D_0$ to $D_n$ )	Implemented by ORing minterm outputs of the decoder

Feature	Multiplexer (MUX)	Decoder
Control	Input variables connected to select lines and data inputs	Input variables connected to decoder inputs
Feature	Multiplexer	Decoder
Gate Count	Fewer gates; compact for certain functions	Higher gate count when multiple minterms are ORed
Wiring	Fewer interconnections	More connections required for OR gates
Feature	Multiplexer	Decoder
Variable Use	One or more variables used on data inputs allows more compact design	All variables used as select inputs, no flexibility
Optimizations	Can reduce logic by using input variables or constants on data lines	Limited to minterm-based implementation unless minimized beforehand
Feature	Multiplexer	Decoder
Propagation Delay	Generally faster; fewer gates in path	May have more delay due to extra logic (e.g., OR gates)
Latency	Lower, especially with tree-based MUX designs	Higher, particularly with multiple OR operations
Trade-off		Explanation
MUX Advantage		More efficient for sparse functions or when logic can be simplified at data inputs
Decoder Advantage		Easier to use for canonical sum-of-minterms representations
MUX Limitation		Can be inefficient if all variables need to be used on data inputs
Decoder Limitation		Requires additional logic (like OR gates), increasing hardware usage

**Additional Questions:**

QUESTIONS AND ANSWERS UNIT 1

COURSE CODE : UE24CS251A

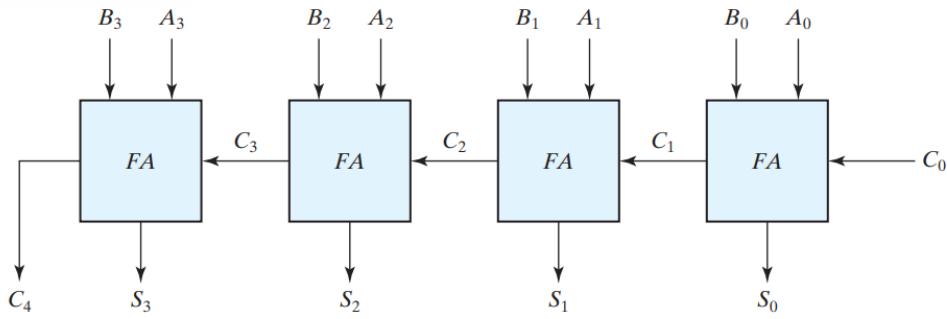
1. Explain the working of a 4-bit binary adder, including the concept of carry propagation and how carry lookahead logic helps in reducing the carry delay. Provide a detailed explanation of the logic diagram for the carry lookahead generator.

**Ans:**

A binary adder is a digital circuit that calculates the arithmetic sum of two binary numbers. It can be built using full adders connected in a cascade, where the output carry from each full adder is passed to the input carry of the next. For n-bit addition, n full adders are used, with the input carry to the least significant bit (LSB) set to 0. In a four-bit binary ripple carry adder, the bits of two binary numbers, A and B, are added starting from the LSB. The output carries ripple through the full adders, producing the sum at the output.

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

For example, adding  $A = 1011$  and  $B = 0011$  results in a sum of  $S = 1110$ . The sum is generated by the full adders as each carry propagates through the chain. This circuit, though simple, would require a complex truth table for a direct design, but cascading standard full adders simplifies the implementation.



**FIGURE 4.9**  
**Four-bit adder**

Carry lookahead logic helps reduce the carry delay in binary addition by addressing the key limitation of ripple carry adders: the time it takes for the carry to propagate through all stages of the adder. In a ripple carry adder, each bit of the sum output depends on the input carry from the previous stage, causing a delay as the carry "ripples" from the least significant bit (LSB) to the most significant bit (MSB). This delay increases with the number of bits, making the addition slower for larger numbers.

Carry lookahead logic reduces this delay by predicting the carry for each bit position without waiting for the actual carry to propagate through all preceding stages. It introduces two new binary variables for each bit position in the adder:

- Generate ( $G_i$ ):** This is generated when both input bits ( $A_i$  and  $B_i$ ) are 1, which guarantees a carry regardless of the input carry ( $C_i$ ). Mathematically,  $G_i = A_i \cdot B_i$ .
- Propagate ( $P_i$ ):** This indicates whether a carry from the previous stage will propagate through this stage. It is true when either  $A_i$  or  $B_i$  is 1 (but not necessarily both). Mathematically,  $P_i = A_i \oplus B_i$  (exclusive OR).

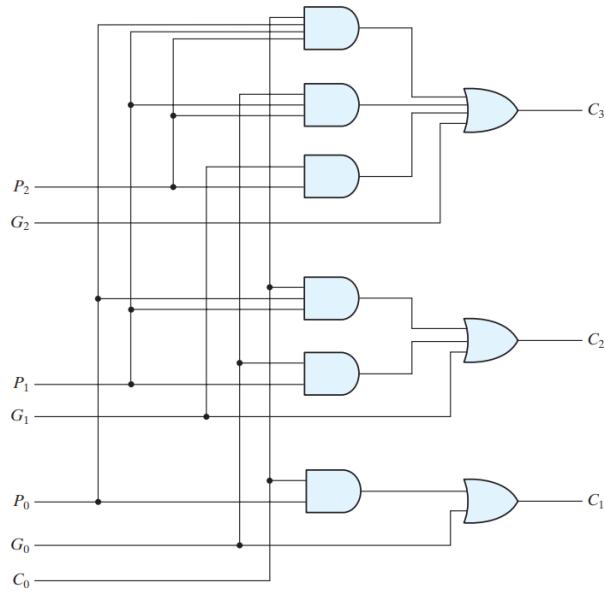
Using these two variables, the carry output ( $C_{i+1}$ ) for each stage can be calculated without waiting for the carry from the previous stage. The carry for each bit is calculated using the following formula:

$$C_1 = G_0 + P_0 C_0$$

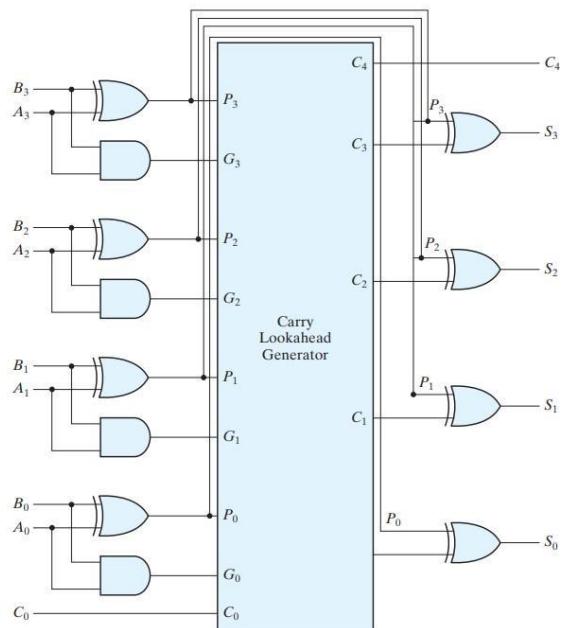
$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 (G_0 + P_0 C_0) = G_1 + P_1 G_0 + P_1 P_0 C_0$$

$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 (G_1 + P_1 G_0 + P_1 P_0 C_0) = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$$

This allows all carry bits to be computed in parallel, rather than sequentially, drastically reducing the propagation delay. The carry lookahead circuit uses a combination of AND and OR gates (or NAND gates) to implement these calculations, and since it calculates the carry for all bit positions simultaneously, it significantly speeds up the addition process. The result is that the carry for any bit position is available almost immediately after the input signals are applied, reducing the overall delay of the adder. However, this speed improvement comes at the cost of increased hardware complexity, as the lookahead logic requires additional gates to compute the carry values in parallel.



**FIGURE 4.11**  
 Logic diagram of carry lookahead generator



**FIGURE 4.12**  
 Four-bit adder with carry lookahead

2. Describe the design and implementation of a full adder circuit. Include the derivation of the Boolean expressions for the sum (S) and carry (C) outputs, and explain how two half adders and an OR gate can be used to implement a full adder.

Ans:

The process of adding n-bit binary numbers involves using a full adder, which adds bits one by one from the least significant bit (LSB) to the most significant bit (MSB), considering any carry

from the previous position. A full adder is a combinational circuit with three inputs—two significant bits ('x' and 'y') and a carry bit ('z')—and two outputs: the sum ('S') and the carry ('C').

The truth table of the full adder shows that the sum output ('S') is '1' when one or all three inputs are '1', while the carry output ('C') is '1' when two or three inputs are '1'. The sum and carry can be expressed as Boolean functions:

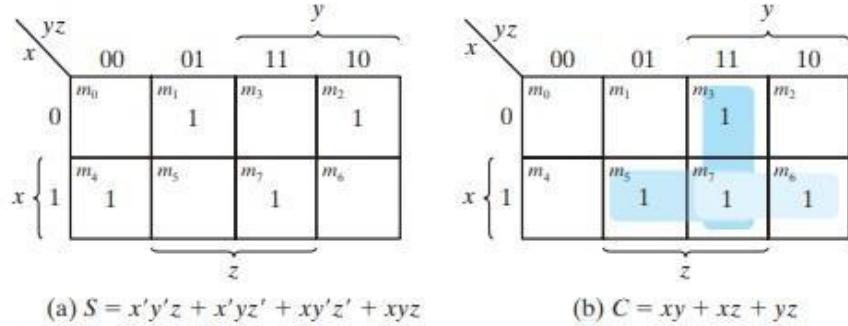
$$S = x'y'z + x'yz' + xy'z' + xyz$$

$$C = xy + xz + yz$$

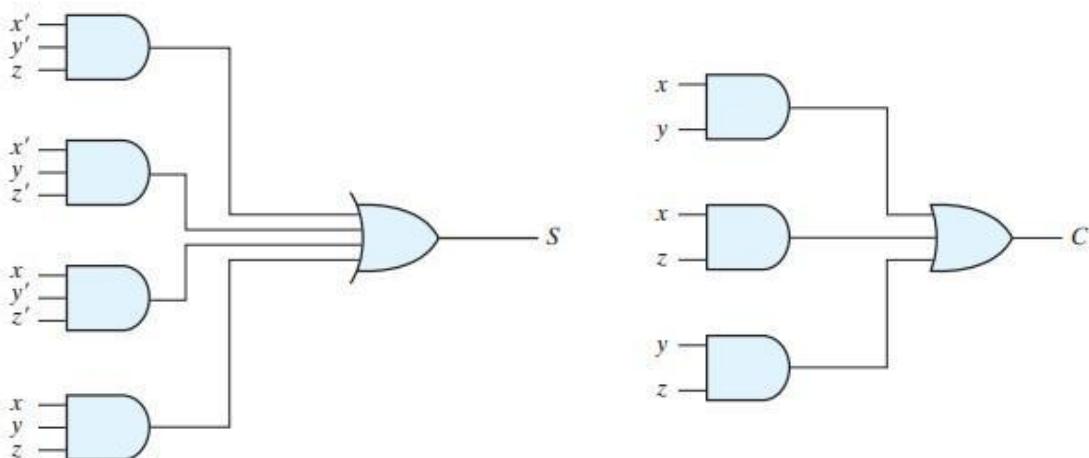
The full adder can be implemented either in sum-of-products form using logic gates or by using two half adders and an OR gate.

**Table 4.4**  
*Full Adder*

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



**FIGURE 4.6**  
K-Maps for full adder



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

in Fig. 4.8. The  $S$  output from the second half adder is the exclusive-OR of  $z$  and the output of the first half adder, giving

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

The carry output is

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

### 3. Explain the working of a 3:8 decoder with a logic diagram and truth table.

**Ans:**

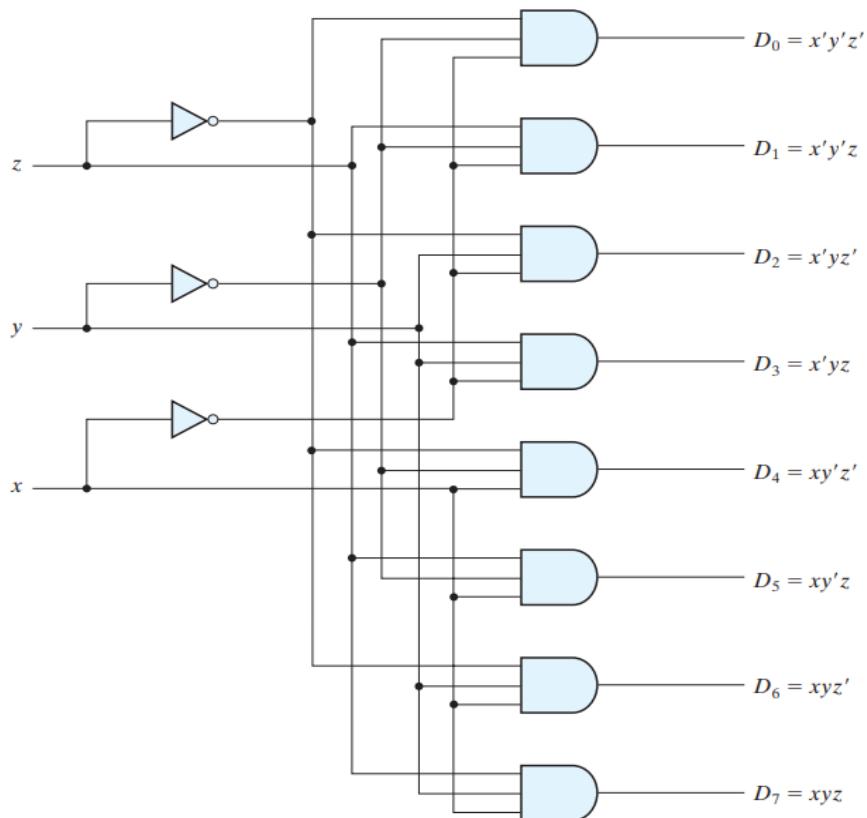
Binary codes represent discrete quantities of information in digital systems, with an  $n$ -bit binary code capable of representing up to  $2^n$  distinct elements. A decoder is a combinational circuit that converts  $n$ -bit binary information into up to  $2^n$  unique outputs, known as  $n$ -to- $m$  line decoders.

These decoders generate the minterms of  $n$  input variables, where each input combination activates a unique output.

A common example is a three-to-eight-line decoder, which converts three inputs into eight outputs, each representing a minterm. This decoder can be used for binary-to-octal conversion. The truth table for the decoder shows that for each input combination, only one output is '1', corresponding to the minterm of the binary number on the input lines.

Decoders can be built using NAND gates, producing minterms in complemented form, and often include enable inputs to control their operation. When the enable input is active, the decoder functions normally; otherwise, it is disabled. A two-to-four-line decoder with an active-low enable input, for example, will only have one output at '0' at any given time, with the others at '1'. This configuration allows the decoder to also function as a demultiplexer, directing information from a single input line to one of several outputs based on selection lines.

Decoders with enable inputs can be connected to create larger decoders, as demonstrated by connecting two 3-to-8-line decoders to form a 4-to-16-line decoder. The enable inputs determine which decoder is active, allowing one to generate minterms while the other outputs zeroes. This feature highlights the versatility of decoders and enable inputs in creating more complex combinational logic circuits by interconnecting standard components.



**FIGURE 4.18**  
**Three-to-eight-line decoder**

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

Inputs			Outputs							
<b>x</b>	<b>y</b>	<b>z</b>	<b>D<sub>0</sub></b>	<b>D<sub>1</sub></b>	<b>D<sub>2</sub></b>	<b>D<sub>3</sub></b>	<b>D<sub>4</sub></b>	<b>D<sub>5</sub></b>	<b>D<sub>6</sub></b>	<b>D<sub>7</sub></b>
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

4. Explain the working of a 4:1 multiplexer using a truth table and logic diagram.

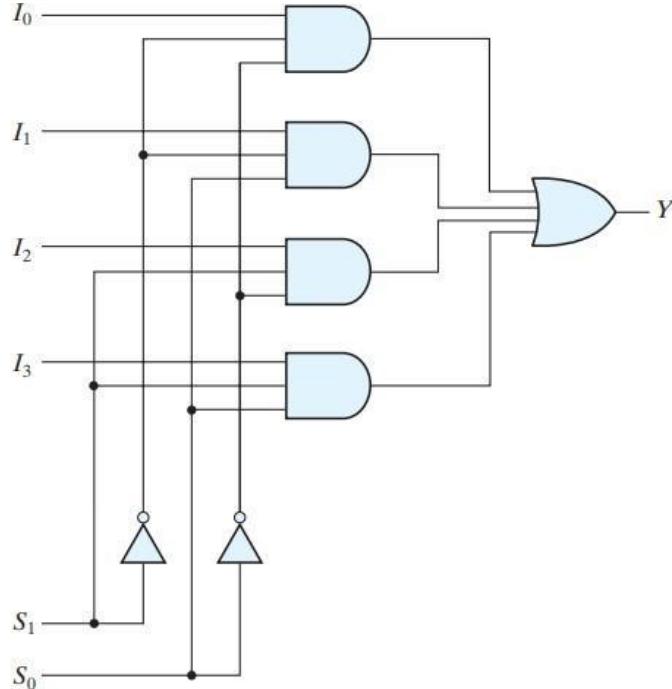
**Ans:**

A four-to-one-line multiplexer (MUX) selects one of four input lines (I0 through I3) to pass to a single output based on the values of two selection lines (S1 and S0). Each input is connected to an AND gate, and the selection lines determine which AND gate is activated. The outputs of the AND gates are then combined using an OR gate, which produces the final output. For example, when S1S0 = 10, the multiplexer selects input I2 to pass to the output, while the other AND gates output zeroes.

A multiplexer functions as a data selector, steering binary information from the selected input to the output. The internal structure resembles a decoder, where the selection lines are decoded to activate one of the AND gates.

Multiplexers can also include an enable input, which controls whether the MUX is active or disabled. When disabled, the output is typically set to zero. Multiplexer circuits can be combined for more complex selection logic, such as a quadruple 2-to-1-line multiplexer, which can select one of two 4-bit sets of data lines based on a single selection input.

The enable input must be active for normal operation, and depending on the selection input, either the A inputs or the B inputs are passed to the outputs. When the enable input is inactive, all outputs are set to zero.



(a) Logic diagram

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

(b) Function table

**FIGURE 4.25**  
**Four-to-one-line multiplexer**

**5. Compare and contrast the Sum of Products (SOP) and Product of Sums (POS) forms of Boolean expressions.**

**Ans:**

The Sum of Products (SOP) and Product of Sums (POS) are two canonical forms used to represent Boolean expressions. They are standard ways of writing logical functions in digital logic design. Here's a comparison and contrast of these two forms:

**Sum of Products (SOP)**

- **Definition:** The SOP form is a Boolean expression where multiple product terms (ANDed terms) are summed (ORED) together. Each product term is composed of literals (variables or their complements) ANDed together.
- **Structure:**
- It has a series of AND operations followed by OR operations.

- Example:  $(A \cdot B) + (A' \cdot C) + (B \cdot C')$

- Here, the product terms are  $(A \cdot B)$ ,  $(A' \cdot C)$ , and  $(B \cdot C')$ , which are then ORed together.

Application:

- It's most commonly used for implementing logic in circuits using AND gates followed by OR gates.
- Often derived directly from a truth table by identifying the rows where the output is 1 (true).
- Minterms:
- Each product term in SOP corresponds to a minterm, which is a unique combination of variables that results in the function output being 1.
- Ease of Use:
- SOP is typically easier to minimize using Karnaugh maps (K-maps) because each 1 in the K-map corresponds directly to a minterm (product term) in the SOP.

Product of Sums (POS)

- Definition: The POS form is a Boolean expression where multiple sum terms (ORed terms) are multiplied (ANDed) together. Each sum term is composed of literals (variables or their complements) ORed together.
- Structure:
- It has a series of OR operations followed by AND operations.

- Example:  $(A+B) \cdot (A'+C) \cdot (B+C')$

- Here, the sum terms are  $(A+B)$ ,  $(A'+C)$ , and  $(B+C')$ , which are then ANDed together.
- Application:
- It's commonly used for implementing logic in circuits using OR gates followed by AND gates.
- Often derived directly from a truth table by identifying the rows where the output is 0 (false). Maxterms:
- Each sum term in POS corresponds to a maxterm, which is a unique combination of variables that results in the function output being 0.
- Ease of Use:

- POS is often less intuitive for simplification compared to SOP, especially when using K-maps, since each 0 in the K-map corresponds to a maxterm (sum term) in the POS.

### Key Differences

#### Derivation:

- SOP is derived from the minterms corresponding to the 1s in a truth table, while POS is derived from the maxterms corresponding to the 0s.
- Gate Implementation:
- SOP is directly implemented using AND gates followed by OR gates.
- POS is implemented using OR gates followed by AND gates.
- Preferred Usage:
- SOP is often preferred in scenarios where the circuit output is predominantly 1, and you want to sum all the cases where the function is true.
- POS is often preferred when the circuit output is predominantly 0, and you want to multiply all the cases where the function is false.

**6. Describe the design and operation of a 4-bit magnitude comparator. Explain how the comparator determines whether one binary number is greater than, less than, or equal to another, using Boolean expressions and logic diagrams.**

#### Ans:

##### Design and Operation of a 4-Bit Magnitude Comparator

A 4-bit magnitude comparator is a combinational circuit that compares two 4-bit binary numbers, A and B, and determines their relative magnitudes. The comparator outputs three binary variables indicating whether A is greater than, less than, or equal to B.

##### Inputs

- Two 4-bit numbers:

$$A = A_3 \ A_2 \ A_1 \ A_0$$

$$B = B_3 \ B_2 \ B_1 \ B_0$$

##### Outputs

##### Outputs

- $A > B$
- $A < B$
- $A = B$

## Design

### 1. Equality Check:

To determine if A and B are equal, compare each pair of bits. Use the exclusive-NOR (XNOR) function for this comparison.

$$x_i = A_i B_i + A'_i B'_i \quad \text{for } i = 0, 1, 2, 3$$

Here,  $x_i = 1$  if  $A_i$  is equal to  $B_i$ , and  $x_i = 0$  otherwise.

The equality output  $A = B$  is given by:

$$A = B = x_3 \cdot x_2 \cdot x_1 \cdot x_0$$

$$(A = B) = x_3 x_2 x_1 x_0$$

The *binary* variable  $(A = B)$  is equal to 1 only if all pairs of digits of the two numbers are equal.

### 2. Greater Than Check:

To determine if A is greater than B, check the most significant bit first, and then proceed to lower bits if necessary. The Boolean function for  $A > B$  is:

$$A > B = A_3 \cdot B'_3 + x_3 \cdot A_2 \cdot B'_2 + x_3 \cdot x_2 \cdot A_1 \cdot B'_1 + x_3 \cdot x_2 \cdot x_1 \cdot A_0 \cdot B'_0$$

Here,  $A > B = 1$  if A is greater than B based on the comparisons of bits from the most significant to the least significant.

### 3. Less Than Check:

To determine if A is less than B, similarly check each bit pair starting from the most significant bit. The Boolean function for  $A < B$  is:

$$A < B = A'_3 \cdot B_3 + x_3 \cdot A'_2 \cdot B_2 + x_3 \cdot x_2 \cdot A'_1 \cdot B_1 + x_3 \cdot x_2 \cdot x_1 \cdot A'_0 \cdot B_0$$

Here,  $A < B = 1$  if  $A$  is less than  $B$  based on the comparisons of bits from the most significant to the least significant.

### Logic Diagram

Equality Check:

Use exclusive-NOR gates to generate  $x_i$  for each bit position.

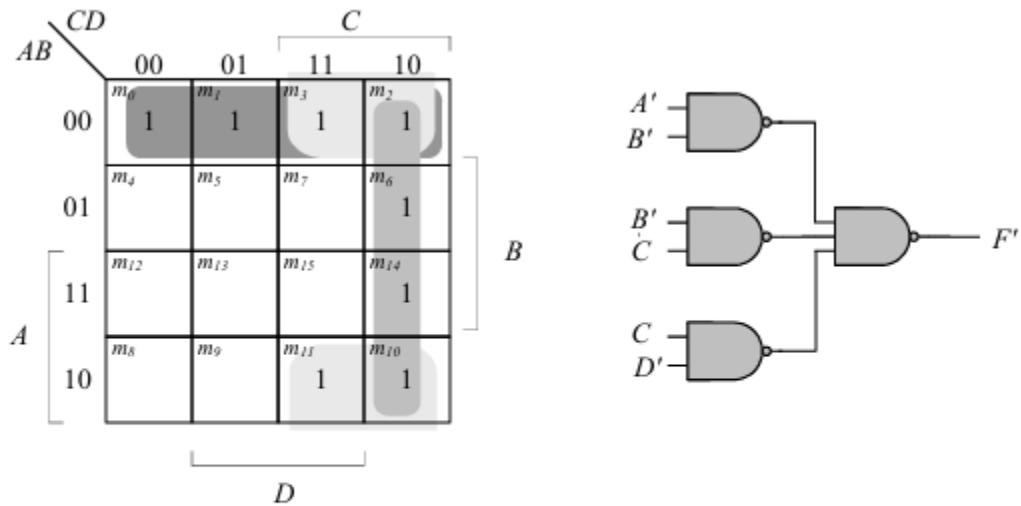
- AND the results of  $x_3, x_2, x_1, x_0$  to get the output  $A = B$ .
2. Greater Than and Less Than Checks:

Use the results of the exclusive-NOR gates to simplify the comparison logic for  $A > B$  and  $A < B$ .

The logic diagram involves connecting the exclusive-NOR gates for each bit comparison, and using AND and OR gates to generate the final outputs for  $A = B$ ,  $A > B$ , and  $A < B$ .

7. Draw a NAND logic diagram that implements the complement of the following function:  
 $F(A,B,C,D) = (0, 1, 2, 3, 6, 10, 11, 14)$  with diagram

Ans:



$$\begin{aligned}
 F &= A'B' + B'C + CD' \\
 F &= ((A+B)(B+C')(C'+D))' \\
 F &= ((A'B')'(B'C)'(CD')')' \\
 F' &= (A'B')'(B'C)'(CD')'
 \end{aligned}$$

8.The truth table for implementing a boolean variable F is given by:

C	B	A	F
0	0	0	d
0	0	1	1
0	1	0	1
0	1	1	d
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

where d represents don't care states. The minimized expression for F is

Ans:

C	BA			
	00	01	11	10
0	(X)	1	(X)	1
1			1	

$$F = \bar{C} + AB$$

9.

The K-map for a Boolean function is shown below, the number of essential prime implicants for this function is (d denotes doesn't care status)

CD	AB			
	00	01	11	10
00	1	0	0	1
01	0	d	0	0
11	0	0	d	1
10	1	0	0	1

Ans:

	AB	CD	00	01	11	10
00	1	0	0	0	(1)	
01	0	d	0	0		
11	0	0	(d)	1		
10	(1)	0	0	0	(1)	

Prime implicants are  $\bar{B} \bar{D} + A \bar{C} \bar{D}$ .

They are also essential prime implicants.

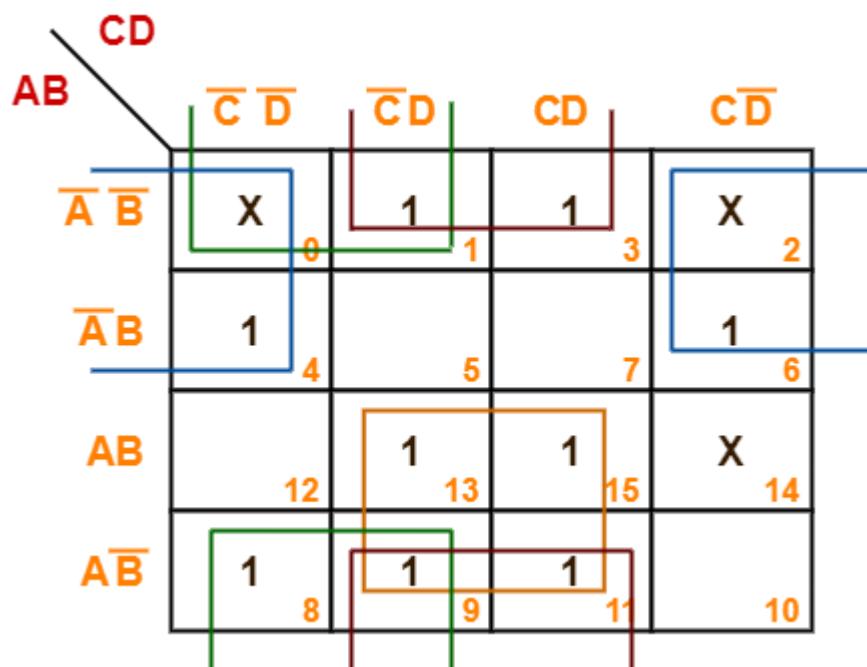
The number of prime implicants is 2

**10.** Minimize the following boolean function-

$$F(A, B, C, D) = \Sigma m(1, 3, 4, 6, 8, 9, 11, 13, 15) + \Sigma d(0, 2, 14)$$

**Ans:**

- Since the given boolean expression has 4 variables, so we draw a  $4 \times 4$  K Map.
- We fill the cells of K Map in accordance with the given boolean function.
- Then, we form the groups in accordance with the above rules.



Now,

$$F(A, B, C, D)$$

$$\begin{aligned}
 &= (AB + AB')(C'D + CD) + (A'B' + AB')(C'D + CD) + (A'B' + AB')(C'D' + C'D) + (A'B' + A'B)(C'D' + CD') \\
 &= AD + B'D + B'C' + A'D'
 \end{aligned}$$

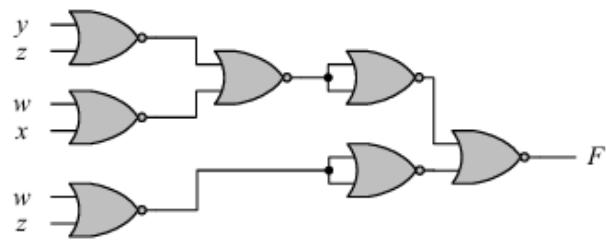
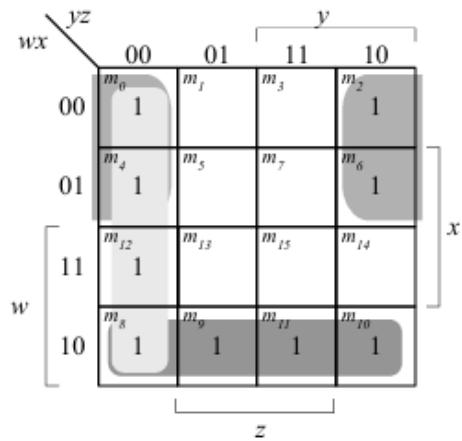
Thus, minimized boolean expression is-

$$F(A, B, C, D) = AD + B'D + B'C' + A'D'$$

**11.** Simplify the following functions, and implement them with two-level NOR gate circuits:

$$(a) F = wx' + y' z' + w'yz'$$

**Ans:**



$$F = y'z' + wx' + w'z'$$

$$F = [(y + z)' + (w' + x)' + (w + z)']'$$

$$F' = [(y + z)' + (w' + x)' + (w + z)']'$$

**Q12.** Simplify the following Boolean expression to its minimal form:

$$F = A \cdot B \cdot C + A \cdot B \cdot C' + A \cdot B' \cdot C + A \cdot B' \cdot C' + A' \cdot B \cdot C$$

Group terms:

- Group 1:  $A \cdot B \cdot C + A \cdot B \cdot C' \rightarrow A \cdot B \cdot (C + C') = A \cdot B$
- Group 2:  $A \cdot B' \cdot C + A \cdot B' \cdot C' \rightarrow A \cdot B' \cdot (C + C') = A \cdot B'$
- Group 3:  $A' \cdot B \cdot C$  remains as-is

Now combine:

$$F = A \cdot B + A \cdot B' + A' \cdot B \cdot C F = A \cdot B + A \cdot B' + A' \cdot B \cdot C$$

Now simplify  $A \cdot B + A \cdot B' = A \cdot (B + B') = A \cdot 1 = A$

$$F = A + A' \cdot B \cdot C F = A + A' \cdot C F = A + A' \cdot B \cdot C$$

This is the minimal expression.

**Q 13- Given the Boolean function:**

$$F = A \cdot B + C \cdot D + E F = A \cdot B + C \cdot D + E$$

**Implement the expression using only NAND gates.**

Use double negation and DeMorgan's Law:

$$F = A \cdot B + C \cdot D + E = \overline{\overline{A \cdot B} + \overline{C \cdot D} + \overline{E}}$$

Apply DeMorgan:

$$F = \overline{(\overline{A \cdot B}) \cdot (\overline{C \cdot D}) \cdot (\overline{E})}$$

**Now using NAND gates:**

- $A \cdot B = (A \text{ NAND } B) \text{ NAND } (A \text{ NAND } B)$
- $C \cdot D = (C \text{ NAND } D) \text{ NAND } (C \text{ NAND } D)$
- $E = E \text{ NAND } E$  (for NOT E)
- The final product is NANDed again with these inversions

This constructs the full NAND-only expression.

**Q 14- Simplify the Boolean expression in Product of Sums (POS) form:**

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

Use distribution and Boolean laws:

Step 1: Multiply first two terms:

$$(A + B + C)(A + B' + C)$$

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

$$= A + (B \cdot B' + B \cdot C + B' \cdot C + C \cdot C)$$

$$= A + (0 + B \cdot C + B' \cdot C + C) = A + C$$

Now multiply with third term:

$$\begin{aligned}
 & (A + C)(A' + B + C) \\
 &= A \cdot A' + A \cdot B + A \cdot C + C \cdot A' + C \cdot B + C \cdot C \\
 &= 0 + A \cdot B + A \cdot C + A' \cdot C + C \cdot B + C \\
 &= A \cdot B + A \cdot C + A' \cdot C + B \cdot C + C
 \end{aligned}$$

Apply Absorption:

- $A \cdot C + A' \cdot C = C$
- $A \cdot B + B \cdot C + C = C + A \cdot B$

Final Simplified Expression:

$$F = C + A \cdot B$$

**Q 15- Using a BCD adder, perform the addition of the decimal numbers 7 and 8. Show all intermediate binary steps and explain how the correction is applied if required.**

Step 1: Convert Decimal to BCD

**Decimal BCD (4-bit)**

7        0111

8        1000

Step 2: Perform Binary Addition

$$0111 \text{ (7)} + 1000 \text{ (8)} = 1111 \text{ (15)}_2$$

So, the binary sum is:

$$\text{Sum} = 1111 \text{ (Decimal 15)}$$

**Step 3: Check for Valid BCD**

- Valid BCD range is **0000 to 1001** (0–9)
- $1111 > 1001 \rightarrow \text{Invalid BCD}$
- Hence, **correction is needed**

**Step 4: Add Correction (Add 0110 = 6)**

$$1111 + 0110 = 10101$$

This is a **5-bit result**:

- $\text{LSB} = 0101 \rightarrow \text{BCD} = 5$
- Carry = 1 → used to represent the **tens digit**

**Step 5: Final Result**

- $\text{LSB} = 0101 \text{ (5)}$
- $\text{Carry} = 1 \rightarrow \text{adds to the tens place: } 1$

$$7+8=15 \rightarrow \text{BCD}=0001\ 0101$$

**BCD Output = 0001 0101 (which represents 15 in BCD)**

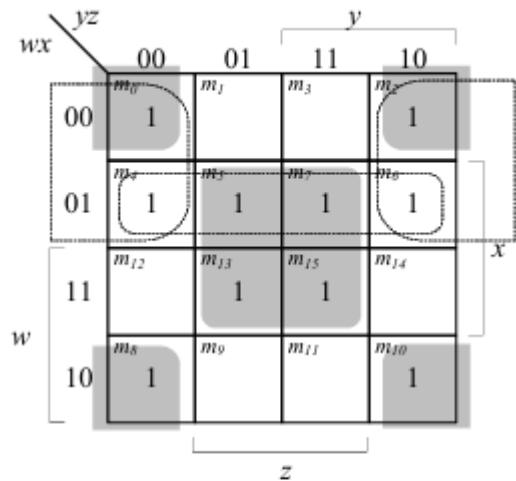
 **Final Answer:**

$$7 + 8 = 15$$

**BCD Result: 0001 0101**

**Q16.** Find all the prime implicants for the following Boolean functions, and determine which are essential:

(a)  $F(w, x, y, z) = \{0, 2, 4, 5, 6, 7, 8, 10, 13, 15\}$



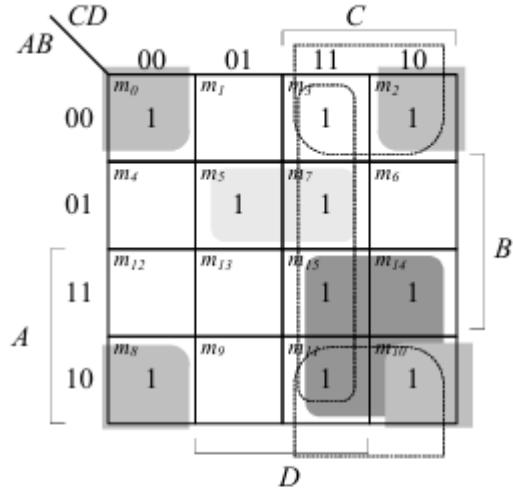
(a)

**Essential:**  $xz, x'z'$

**Non-essential:**  $w'x, w'z'$

$$F = xz + x'z' + (w'x \text{ or } w'z')$$

(b)  $F(A, B, C, D) = \{0, 2, 3, 5, 7, 8, 10, 11, 14, 15\}$



(b)

**Essential:**  $B'D'$ ,  $AC$ ,  $A'BD$

**Non-essential:**  $CD$ ,  $B'C$

$$F = B'D' + AC + A'BD + (CD \text{ OR } B'C)$$

**Q 17- Design an 8x1 multiplexer using only 2x1 multiplexers.**

**Explain the connection hierarchy and calculate the total number of 2x1 MUXes required. Show the step-by-step structure and logic.**

### 1. Understanding the Problem

- An **8x1 MUX** has:
  - **8 data inputs:**  $D_0$  to  $D_7$
  - **3 selection lines:**  $S_2$ ,  $S_1$ ,  $S_0$
  - **1 output**

We want to implement this using only **2x1 multiplexers**, which each:

- Have **2 inputs ( $I_0$ ,  $I_1$ )**
- **1 select line (S)**
- **1 output (Y)**

### 2. Strategy: Hierarchical Design

To reduce from 8 inputs to 1 output using 2-input muxes:

- First level: Select between every **pair of inputs** (i.e.,  $8 \rightarrow 4$  outputs)
- Second level: Select between those  $4 \rightarrow 2$

- Third level: Select between  $2 \rightarrow 1$  final output

This forms a **MUX tree**.

### 3. Step-by-Step Implementation

#### Level 1 (4 multiplexers):

- Use 4 **2x1 MUXes** to select between:
  - MUX1:  $D_0$  vs  $D_1$
  - MUX2:  $D_2$  vs  $D_3$
  - MUX3:  $D_4$  vs  $D_5$
  - MUX4:  $D_6$  vs  $D_7$

Select line used:  **$S_0$**

#### Level 2 (2 multiplexers):

- Use 2 **2x1 MUXes** to select between:
  - MUX5: output of MUX1 vs MUX2
  - MUX6: output of MUX3 vs MUX4

Select line used:  **$S_1$**

#### Level 3 (1 multiplexer):

- Use 1 **2x1 MUX** to select between:
  - MUX7: output of MUX5 vs MUX6

Select line used:  **$S_2$**

### 4. Connection Summary

- Total inputs:  $D_0$  to  $D_7$
- Selection lines:  $S_2$  (MSB),  $S_1$ ,  $S_0$  (LSB)
- Output of final  $2 \times 1$  MUX gives  $Y$

### 5. Logic Representation

#### First Level ( $S_0$ ):

$$\text{MUX1} \rightarrow D_0, D_1 \rightarrow S_0 \rightarrow Y_0$$

$$\text{MUX2} \rightarrow D_2, D_3 \rightarrow S_0 \rightarrow Y_1$$

$$\text{MUX3} \rightarrow D_4, D_5 \rightarrow S_0 \rightarrow Y_2$$

$$\text{MUX4} \rightarrow D_6, D_7 \rightarrow S_0 \rightarrow Y_3$$



Second Level ( $S_1$ ):

MUX5  $\rightarrow Y_0, Y_1 \rightarrow S_1 \rightarrow Y_4$

MUX6  $\rightarrow Y_2, Y_3 \rightarrow S_1 \rightarrow Y_5$

Third Level ( $S_2$ ):

MUX7  $\rightarrow Y_4, Y_5 \rightarrow S_2 \rightarrow$  Final Output

#### **Number of $2 \times 1$ MUXes Required**

**Level MUXes used**

**Level 1 4**

**Level 2 2**

**Level 3 1**

**Total 7**

To implement an  $8 \times 1$  MUX using  $2 \times 1$  MUXes, we need:

- 7 total  $2 \times 1$  multiplexers
- 3 select lines ( $S_2 S_1 S_0$ ) arranged hierarchically
- The implementation follows a tree structure with 3 levels of selection

# PES UNIVERSITY, Bangalore

Established under Karnataka Act No. 16 of 2013)

## Department of Computer Science & Engineering

### UE24CS251A: DIGITAL DESIGN AND COMPUTER ORGANIZATION

1. Explain how a D flip-flop can be constructed using two SR latches connected in a master-slave configuration. Why is this configuration preferred over a single latch for synchronous digital circuits?

Ans:

A D flip-flop is a fundamental memory element used in synchronous digital circuits to store a single bit of data. It captures the input data at a specific clock transition and holds the value stable until the next clock event.

One common way to implement a D flip-flop is by using two SR (Set-Reset) latches connected in a master-slave configuration.

Construction:

The master latch is the first SR latch, which receives the input data (D) and is enabled when the clock (CLK) is LOW. During this time, the master latch is transparent, allowing the input D to pass through to its output.

The slave latch is the second SR latch, connected to the output of the master latch, and is enabled when the clock is HIGH. When enabled, the slave latch captures and holds the state from the master latch.

Operation:

When the clock is LOW, the master latch is enabled and follows the input D. The slave latch is disabled and holds its previous state.

When the clock transitions from LOW to HIGH (rising edge), the master latch is disabled, locking the data present at the instant of transition.

Simultaneously, the slave latch becomes enabled and updates its output to match the master latch's locked data.

This way, the output Q of the flip-flop changes only on the clock edge and remains stable otherwise.

Advantages of Master-Slave Configuration:

The master-slave design eliminates the problem of transparency and race conditions inherent in single latches, which are level-sensitive devices.

It ensures that the output changes only at discrete clock edges (edge-triggered behavior), which is crucial for predictable timing in synchronous circuits.

This configuration prevents the output from responding to changes in the input during the active clock period, providing a clean and stable data storage mechanism.

2. You are designing a digital stopwatch that counts seconds using a binary counter circuit. Explain why a D flip-flop implemented with a master-slave SR latch configuration is preferred for the counter's storage elements over a simple SR latch. How does this choice affect the accuracy and reliability of the stopwatch?

Answer:

In designing a **digital stopwatch** that counts seconds, precise and reliable storage of the current count is critical. The storage elements in the counter circuit must update their states synchronously with the clock pulses to ensure the count increments correctly.

Using a **D flip-flop constructed from two SR latches in a master-slave configuration** is preferred over a simple SR latch for the following reasons:

**1. Edge-Triggered Operation:**

- The master-slave configuration ensures that the flip-flop changes its state **only at the clock's edge** (rising or falling), not while the clock is at a constant level.
- This eliminates unwanted state changes that can occur in simple SR latches, which are **level-sensitive** and may respond to input changes anytime the enable signal is active.

**2. Prevention of Race Conditions and Glitches:**

- Simple latches may become transparent during the clock level and can cause glitches or unintended multiple transitions if the input changes while the latch is enabled.
- Master-slave D flip-flops capture input only once per clock cycle, **avoiding race conditions** that could cause the counter to skip or repeat counts.

**3. Reliable Synchronization:**

- Since the stopwatch counter updates its state only on discrete clock edges, the timing of increments is predictable and stable.
- This synchronization is vital for the stopwatch to maintain **accurate time measurement** without drift or errors due to asynchronous input changes.

**4. Stable Data Storage:**

- The slave latch holds the stored bit steady until the next clock edge, ensuring the output signals remain valid and stable for the rest of the clock period.
- This stability contributes to the **overall reliability** of the stopwatch.

**3. You are designing a binary counter circuit for a digital clock. Explain how using JK flip-flops with the toggle function facilitates the design, and what precautions must be taken to avoid race-around conditions.**

**Answer:**

- JK flip-flops, when both J and K inputs are set to 1, act as **toggle flip-flops**, changing their output on every clock pulse.
- Using JK flip-flops simplifies the design of **binary counters** since each flip-flop toggles its state in response to the previous flip-flop, implementing counting naturally.
- However, **race-around condition** occurs if the clock pulse width is longer than the flip-flop's propagation delay, causing multiple toggles within a single clock cycle, leading to unpredictable output.
- To avoid this, designers use **master-slave flip-flops** or **edge-triggered flip-flops**, which ensure the output changes only once per clock pulse.

- This ensures the binary counter operates correctly and reliably, incrementing the count only once per clock cycle.

4. In an asynchronous sequential circuit, why are latches commonly used instead of flip-flops, and what role does propagation delay play in such circuits?

Answer:

- **Asynchronous sequential circuits** operate without a global clock; their behavior depends on the order and timing of input changes.
- Latches, being **level-sensitive devices**, are preferred in asynchronous circuits because they can change state as soon as the input conditions are met without waiting for a clock edge.
- Propagation delay, which is the time taken for a signal to travel through gates and latches, is critical because it provides the required **timing delay** for correct state transitions.
- The propagation delay can be used to ensure proper sequencing of state changes and to avoid hazards like glitches.
- However, managing delays is challenging, and improper timing can cause instability, which is why asynchronous circuits are less common and more difficult to design reliably than synchronous circuits.

5. A security gate control system opens the gate only when a correct sequence of signals is received: "A" followed by "B". Any incorrect sequence resets the system to its initial state.

The system is implemented using D flip-flops.

- Input  $xxx$  represents the received signal:  $x=0$  for "A",  $x=1$  for "B".
  - Output  $y=1$  only when the sequence is completed correctly; otherwise  $y=0$ .
1. Design the state diagram for this system using the Moore model.
  2. Write the state table and state equations for D flip-flops.
  3. Derive the Boolean expressions for DA and DB (assuming two flip-flops A and B are used to encode states).

### 1. State Diagram (Moore Model)

- **S0 (00)**: Initial state, waiting for "A". If  $x = 0$ , go to S1; else remain in S0.
- **S1 (01)**: "A" received, waiting for "B". If  $x = 1$ , go to S2; else reset to S0.
- **S2 (10)**: Sequence "A" → "B" completed. Output  $y = 1$ . On any next input, return to S0.

### 2. State Table

Present State (A,B)	Input $x$	Next State (A,B)	Output $y$
00 (S0)	0 ("A")	01 (S1)	0
00 (S0)	1 ("B")	00 (S0)	0
01 (S1)	0 ("A")	00 (S0)	0
01 (S1)	1 ("B")	10 (S2)	0
10 (S2)	0 or 1	00 (S0)	1

### 3. State Equations (for D flip-flops)

Let **A** = MSB, **B** = LSB of state encoding.

From the table:

- $D_A = B \cdot x$
- $D_B = \overline{A} \cdot \overline{B} \cdot \overline{x}$
- $y = A \cdot \overline{B}$

6. An elevator control system uses **two JK flip-flops (A,B)** to represent its four states:

- **00**: Ground floor
- **01**: First floor
- **10**: Second floor
- **11**: Maintenance mode

The input  $x = 1$  for **move up**,  $x = 0$  for **move down**. The system ignores all requests when in maintenance mode (11).

1. Draw the **state diagram** for the system.
2. Write the **state table**.
3. Derive the **flip-flop input equations** JA,KA,JB,KB

### 1. State Diagram

- $00 \rightarrow (x=1) \rightarrow 01 \rightarrow (x=1) \rightarrow 10 \rightarrow (x=1) \rightarrow 11$  (Maintenance)
- 11 loops back to itself for both  $x=0$  and  $x=1$ .
- Downward moves happen when  $x=0$  in reverse order ( $10 \rightarrow 01 \rightarrow 00$ ).

### 2. State Table

Present State (A,B)	Input $x$	Next State (A,B)
00	1	01
00	0	00
01	1	10
01	0	00
10	1	11
10	0	01
11	0 or 1	11

### 3. Flip-Flop Input Equations (from JK characteristic table)

From state transitions:

- $J_A = B \cdot x$
- $K_A = \overline{B} \cdot \overline{x}$
- $J_B = \overline{A} \cdot x$
- $K_B = A \cdot \overline{x}$

7. A university wants to design an **automatic gate** at the entrance of the library. The gate opens immediately when a valid student ID card is swiped, provided the gate is currently closed.

1. Identify whether a **Mealy** or **Moore** model is more suitable for this system.
2. Draw a brief state diagram and explain why this model is preferred.

**Answer:**

1. **Mealy model** is preferred because the **output (gate opens)** depends on **both the current state (gate closed)** and the **instantaneous input (valid ID detected)**. This allows an immediate response without waiting for the next clock cycle.
2. **State Diagram (Text Form):**
  - **State S0:** Gate closed. Input = valid ID → Output = open signal; move to S1.
  - **State S1:** Gate open. After a timeout → Output = close signal; move to S0.
3. **Reason:** Faster response since output is generated as soon as the input is detected, making it ideal for real-time entry systems.

8. company wants to design a **coffee vending machine** that dispenses coffee **only after** the user has inserted the correct amount of coins (₹20). The dispensing process should not be interrupted by any new input during dispensing.

1. Identify whether a **Mealy** or **Moore** model is more suitable.
2. Explain how the system would behave in terms of states and outputs.

**Answer:**

1. **Moore model** is preferred because the **output (dispense coffee)** depends only on the current state (Dispense state), not on any new inputs. This ensures stable, glitch-free operation during dispensing.
2. **State Explanation:**
  - **State S0:** Waiting for coins. Output = No coffee.
  - **State S1:** Coins collected = ₹20. Output = Dispense coffee. After dispensing → move to S0.
3. **Reason:** Once the machine enters the Dispense state, it ignores new inputs until the action completes, preventing errors or multiple dispenses.

9. A sequential circuit is used in a vending machine to dispense a product after it detects the sequence 1101 on the input line (overlapping allowed).

The initial state diagram for the Moore machine has 7 states.

- (a) Identify the pairs of equivalent states and perform state reduction to minimize the number of states.
- (b) After reduction, determine the minimum number of flip-flops required if binary state assignment is used.

Ans:

**(a) State Reduction:**

- From the state table, compare states for equivalence:
  - $S_2$  and  $S_5$  are equivalent (same outputs, same/equivalent next states)
  - $S_3$  and  $S_6$  are equivalent
- After merging:  $7 \rightarrow 5$  states

**(b) Minimum flip-flops:**

- For  $m = 5$ ,  $2^n \geq m \rightarrow 2^3 = 8 \geq 5 \rightarrow 3$  flip-flops needed.

10. A traffic light controller has **4 states**:

- S0: Green for North-South
- S1: Yellow for North-South

- S2: Green for East-West
- S3: Yellow for East-West

(a) Assign states using **One-Hot encoding** and draw the state transition table.

(b) Compare the speed advantage of One-Hot over binary encoding for FPGA implementation.

(a) **One-Hot Assignment:**

State	Code	Next State	Output
S0	0001	S1	NS=Green, EW=Red
S1	0010	S2	NS=Yellow, EW=Red
S2	0100	S3	NS=Red, EW=Green
S3	1000	S0	NS=Red, EW=Yellow

(b) **Speed Advantage:**

- **One-Hot Encoding:** No decoding logic needed—each flip-flop directly represents a state. Faster clock speeds achievable in FPGA designs.
- **Binary Encoding:** Requires decoding logic, which increases combinational delay.

11. Explain the operation of a serial adder with a neat diagram.

Ans:

A **serial adder** adds two binary numbers bit-by-bit using:

- Two shift registers (A: augend, B: addend)
- One full adder
- A D flip-flop to store carry

**Working:**

1. Initially, registers A and B hold the two numbers, and carry is set to 0.
2. Each clock pulse shifts bits right from A and B into the full adder.
3. The sum bit is stored back in A, and carry is stored in the flip-flop.
4. After n clock pulses (for n-bit numbers), the addition is complete.

12. In a UART (Universal Asynchronous Receiver-Transmitter) system, shift registers are used to convert data between serial and parallel forms.

Explain how a **Serial-In, Parallel-Out (SIPO)** register can be used to receive data from a serial communication line and make it available to the CPU in parallel.

**Answer:**

In serial communication, data arrives one bit at a time. A SIPO register collects these bits over multiple clock pulses.

- **Step 1:** The first bit enters the first flip-flop on the first clock pulse.
- **Step 2:** Each subsequent clock shifts the previous bits to the next flip-flop, while the new bit enters the first position.
- **Step 3:** After n clock cycles (for an n-bit register), the register holds the full data word.
- **Step 4:** The CPU can then read all bits in parallel, enabling faster processing.

**Example:** In UART reception, an 8-bit SIPO register reconstructs a byte sent serially over a wire.

### 13. Question:

In a cost-sensitive embedded device, a designer chooses a **serial adder** instead of a parallel adder. Explain the trade-offs and why the choice is suitable for this application.

#### Answer:

A serial adder uses only one full adder and a carry flip-flop, along with shift registers to hold operands.

- **Advantages:** Requires fewer gates, reducing cost, power consumption, and chip size.
- **Disadvantages:** Slower since it processes one bit per clock cycle.
- **Suitability:** In low-speed applications (e.g., sensor data processing in a weather station), performance requirements are modest, making the cost savings more valuable than high speed.

14. A digital stopwatch counts **from 00 to 59 seconds** using two cascaded BCD ripple counters. Explain how this can be implemented.

Ans:

#### Answer:

- Stopwatch requires mod-60 counter → decomposed into mod-10 (units) and mod-6 (tens).
- Units (0–9) → 4-bit BCD ripple counter.
- Tens (0–5) → 3-bit counter with reset after 6.
- Units overflow at 9 triggers increment of tens.
- Together, system cycles 00 → 59 → 00.
- Application: Used in watches, timers.

15. A **digital lock** requires a 4-digit sequence (0000 → 1111) to open.

Design a **4-bit ripple up-counter** using **T flip-flops** to generate the 16 states (0–15).

1. Explain how the counter progresses through the binary sequence.
2. Draw the complete **circuit diagram** of the 4-bit ripple counter using T flip-flops.
3. Indicate how a **decoder** can be used to detect the state 1010 (decimal 10), which represents the unlock condition.

Ans:

**Operation:**

- The LSB flip-flop toggles with each clock pulse.
- Each subsequent flip-flop toggles when the previous flip-flop transitions from 1 → 0 (falling edge).
- Sequence: 0000 → 0001 → 0010 → ... → 1111 → back to 0000.

**Circuit Diagram (Expected):**

- 4 T flip-flops connected in series.
- T inputs tied to logic 1 (so each flip-flop toggles when clocked).
- Clock of first FF = external clock.
- Clock of next FF =  $\bar{Q}$  (or Q) of previous FF.
- Outputs: Q0 (LSB), Q1, Q2, Q3 (MSB).

**Decoder Integration:**

- A 4-to-16 decoder connected to Q3 Q2 Q1 Q0.
- When inputs = 1010, the decoder output line 10 goes HIGH.
- This HIGH can trigger a relay to **unlock the system**.

16. Explain the role of enable, clear, and load inputs in a synchronous counter.

Answer:

**Role of Control Inputs in a Synchronous Counter****1. Enable (Count Enable):**

- Determines whether the counter is active or idle.
- If **Enable = 1**, the counter responds to clock pulses (increments/decrements).
- If **Enable = 0**, all J/K (or T) inputs are forced to 0 → flip-flops hold their present state.
- Useful for pausing the counter without losing the current count.

**2. Clear (Reset):**

- Forces the counter output to **0000** irrespective of the clock or enable.
- Can be **synchronous** (takes effect at the next clock edge) or **asynchronous** (takes effect immediately).
- Used to initialize the counter at power-on or to restart the sequence.

**3. Load (Parallel Load):**

- Allows a specific binary number to be loaded into the counter directly from the **Data\_in lines**.

- If **Load = 1**, on the next clock edge the counter ignores counting and loads the given value.
- Useful for programmable counting (e.g., starting from N instead of 0).

**17. What is a MOD-N counter?** Design a MOD-6 synchronous counter using T flip-flops.

Answer:

A **MOD-N counter** is a sequential circuit that cycles through **N distinct states** (counts) and then repeats. For example, a MOD-6 counter goes through 6 states ( $0 \rightarrow 5$ ) before returning to 0. Synchronous counters update **all flip-flops on the same clock edge**, so they are fast and glitch-resistant compared to ripple counters

1) Number of Flip-Flops

- Need 3 flip-flops ( $Q_2, Q_1, Q_0$ ) since  $\lceil \log_2 6 \rceil = 3$ . States 110 and 111 are unused.

2) State Sequence (Valid States)

$000(0) \rightarrow 001(1) \rightarrow 010(2) \rightarrow 011(3) \rightarrow 100(4) \rightarrow 101(5) \rightarrow 000(0)$

3) T-Flip-Flop Excitation

For T-FF:  $Q^+ = Q \oplus T \Rightarrow T = Q \oplus Q^+$

From the sequence above:

- $T_0 = 1$  (Q0 toggles every count)
- $T_1 = \neg Q_2 \cdot Q_0$  (Q1 toggles when  $Q_2=0$  and  $Q_0=1 \Rightarrow$  at states 001 and 011)
- $T_2 = Q_0 \cdot (Q_1 \oplus Q_2)$  (Q2 toggles at states 011 and 101)

With an optional Count Enable E:

- $T_0 = E, T_1 = E \cdot \neg Q_2 \cdot Q_0, T_2 = E \cdot Q_0 \cdot (Q_1 \oplus Q_2)$

**18. Discuss the working of a BCD (MOD-10) synchronous counter.**

Why does it not follow a regular binary sequence? Explain with a state table.

Answer:

A **BCD counter** (Binary Coded Decimal counter) is a **MOD-10 counter**:

- It counts from **0000 (decimal 0)** to **1001 (decimal 9)** in binary.
- After 1001, the counter resets back to 0000.
- Thus, it cycles through **10 states only** (instead of all 16 states of a 4-bit binary counter).

**How it works:**

1. A 4-bit synchronous counter is used.
2. The **Enable** input allows counting on each clock pulse.
3. When the count reaches **1001 (decimal 9)**, detection logic activates the **Clear or Load** input, forcing the next state to **0000**.
4. This ensures the sequence is **0–9 only**, making it suitable for decimal displays (like digital clocks, calculators).

**Why It Does Not Follow a Regular Binary Sequence**

- A 4-bit binary counter would naturally go through **16 states ( $0000 \rightarrow 1111 = 0\text{--}15$ )**.
- But BCD counters are restricted to **decimal digits (0–9)**.
- Therefore, the states **1010 (10) to 1111 (15)** are **unused/invalid** and must be skipped.
- Special reset/load logic truncates the count after 1001

19. Design a **3-bit synchronous up counter** using **T flip-flops**.

1. Derive the T-input equations for each flip-flop.
2. Prepare the state transition table (present state  $\rightarrow$  next state).
3. Explain how the counter avoids glitches compared to a ripple counter.
4. Draw the logic diagram.

**Answer**

- Derivation:  $T_0=1$ ,  $T_1=Q_0$ ,  $T_2=Q_0 \cdot Q_1$ ;  $T_0 = 1$ ;  $T_1 = Q_0$ ;  $T_2 = Q_0 \cdot Q_1$ ;  $T_0=1, T_1=Q_0, T_2=Q_0 \cdot Q_1$ .
- State table: Show transitions  $000 \rightarrow 001 \rightarrow 010 \rightarrow 011 \rightarrow 100 \rightarrow 101 \rightarrow 110 \rightarrow 111 \rightarrow 000$ .
- Explanation: Since all flip-flops are triggered by the same clock, synchronous counters change all bits **at the same time**, avoiding ripple delays.
- Logic diagram with 3 T-FFs and required AND gates.

20. Differentiate between Ring and Johnson counters

Answer:

Feature / Aspect	Ring Counter	Johnson Counter (Twisted Ring)
Basic Operation	A single '1' (or '0') circulates around N flip-flops $\rightarrow$ produces exactly <b>N states</b> .	Inverts last flip-flop output and feeds back $\rightarrow$ produces <b><math>2N</math> states</b> with N flip-flops.
Hardware Need	Requires <b>more flip-flops</b> for same modulus (e.g., 10 states $\rightarrow$ 10 FFs).	More efficient: <b>fewer flip-flops</b> (10 states $\rightarrow$ 5 FFs).
Real-Time Example 1	<b>Instruction Pipeline Controller</b> in CPUs/DSPs $\rightarrow$ Each stage (Fetch, Decode, Execute, Memory, WB) activated sequentially, glitch-free.	<b>Clock/Phase Generator in PLLs</b> $\rightarrow$ Generates equally spaced multi-phase clocks (used in communication systems, VLSI).
Real-Time Example 2	<b>Traffic Light / Sequencer Systems</b> $\rightarrow$ Each output directly drives one lamp or actuator without decoder.	<b>DAC/ADC Control</b> $\rightarrow$ Drives resistor ladders in DACs or timing signals in SAR ADC sampling.

Feature / Aspect	Ring Counter	Johnson Counter (Twisted Ring)
Real-Time Example 3	Time Division Multiplexing (TDM) slot assignment → one channel active at a time.	Keyboard Scanners / Pattern Generators → Johnson counter provides unique non-overlapping codes.
Advantage	Very simple decoding (each flip-flop directly drives one stage).	Efficient state utilization ( $2N$ states with $N$ FFs), great for timing/phase generation.
Disadvantage	Wastes flip-flops for large MOD values.	Requires decoding logic to identify states (less direct than ring counter).